2016

Evidence-enabled verification for the Linux kernel

Ahmed Yousef Tamrawi
Iowa State University

Evidence-enabled verification for the Linux kernel

by

Ahmed Yousef Tamrawi

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Iowa State University

Ames, Iowa

2016

DEDICATION

I would like to dedicate this thesis to my parents, sisters, wife, son Yusuf, and to my daughter Nawwar without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Suraj Kothari for his guidance, patience, inspirational thoughts, and support throughout this research and the writing of this thesis. His insights and words of encouragement have often inspired me and renewed my hopes for completing my graduate education. I would also like to thank Jeremiäs Sauceda, Jon Mathews, and Nikhil Ranade from EnSoft for their guidance and help throughout my graduate study. I would additionally like to thank my committee members for their efforts and contributions to this work: Dr. Srikanta Tirthapura, Dr. Manimaran Govindarasu, Dr. Yong Guan, and Dr. Robyn R. Lutz.

ABSTRACT

Formal verification of large software has been an elusive target, riddled with problems of low accuracy and high computational complexity [22, 26, 82, 95]. With growing dependence on software in embedded and cyber-physical systems where vulnerabilities and malware can lead to disasters, an efficient and accurate verification has become a crucial need. The verification should be rigorous, computationally efficient, and automated enough to keep the human effort within reasonable limits, but it does not have to be completely automated. The automation should actually enable and simplify human cross-checking which is especially important when the stakes are high. Unfortunately, formal verification methods work mostly as automated black boxes with very little support for cross-checking.

This thesis is about a different way to approach the software verification problem. It is about creating a powerful fusion of automation and human intelligence by incorporating algorithmic innovations to address the major challenges to advance the state of the art for accurate and scalable software verification where complete automation has remained intractable. The key is a mathematically rigorous notion of verification-critical evidence that the machine abstracts from software to empower human to reason with. The algorithmic innovation is to discover the patterns the developers have applied to manage complexity and leverage them. A pattern-based verification is crucial because the problem is intractable otherwise. We call the overall approach Evidence-Enabled Verification (EEV).

This thesis presents the EEV with two challenging applications: (1) EEV for **Lock/Unlock Pairing** to verify the correct pairing of mutex lock and spin lock with their corresponding unlocks on all feasible execution paths, and (2) EEV for **Allocation/Deallocation Pairing** to verify the correct pairing of memory allocation with its corresponding deallocations on all feasible execution paths. We applied the EEV approach to verify recent versions of the Linux kernel. The results include a comparison with the state-of-the-art Linux Driver Verification (LDV) tool, effectiveness of the proposed visual models as verification-critical evidence, representative examples of verification, the discovered bugs, and limitations of the proposed approach.

CHAPTER 1. OVERVIEW

With growing dependence on software in embedded and cyber-physical systems where vulnerabilities and malware can lead to disasters, efficient and accurate verification has become a crucial need for safety and cybersecurity. The challenges of verifying our software infrastructure are daunting, in part because of the complexity of the software, but also due to the sheer volume of it. The Linux kernel alone, which provides the basis for so many devices (web servers, routers, smart phones, desktops), is over 12 MLOC.

1.1 State-Of-The-Art Formal Verification Techniques

Formal verification has been the holy grail of software engineering research [44]. 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 [58]. However, there are two fundamental limitations: (A) a completely automated and accurate analysis encounters NP hard problems [32, 79, 87], and (B) formal verification methods work mostly as automated black boxes with little support for cross-checking [1, 2, 5, 39, 41, 83, 89, 98].

The current formal verification approaches are not geared to produce evidence to empower and integrate human reasoning into the verification process. Consider the Matching Pair Verification (MPV) problem: it involves verifying the correct pairing of two events on all feasible execution paths. Specific examples of such events can be: *allocation* and *deallocation* of memory, *locking* and *unlocking* of mutex, or *sensitive source* and *malicious sink* of sensitive information in the context of cybersecurity. MPV is broadly applicable to problems of software safety and cybersecurity. State-of-the-art research and commercial tools for MPV produce either very little evidence, voluminous evidence, or evidence that refers to the intermediate representation for the verification machinery but not to the source code. Such evidence is hard to decipher; it does not simplify cross-checking.

1.2 A Different Approach to Software Verification

This thesis is about a different approach to software verification. It is about targeting automation to amplify human intelligence and solve a hard problem by integrating automation with human reasoning. The verification should be automated wherever possible, and complemented by human reasoning wherever needed. The key is a mathematically rigorous notion of verification-critical evidence that the machine abstracts from software to empower human to: (a) cross-check an automatically verified instance, and (b) complete the verification where automation falls short. The goal is to create a powerful fusion of automated evidence abstraction, evidence-based reasoning, and pattern-based automated verification. The machine and human have different strengths and weaknesses to complement each other. Even for automatically completed verification, we advocate human cross-checking. This is especially important for critical systems where failures of software can be catastrophic. The cross-checking should be simplified by automation to conserve human effort.

We propose an Evidence-Enabled Verification (EEV) approach for the MPV problem. EEV conserves human effort and empowers reasoning by facilitating it with machine-produced evidence which not only makes automation computationally efficient but also simplifies human reasoning. The automation part of the EEV approach partitions the problem into verification instances, produces the verification-critical evidence for each instance, and wherever possible performs the verification automatically. When the automatic verification falls short, the evidence provides significant help to the analyst to understand the complications surrounding the particular instance so that the analyst can decide what additional evidence to gather and what reasoning to apply to complete the verification. With a better understanding, it is sometimes possible to define a pattern to encapsulate a barrier and advance the automation so that it can address the particular barrier.

This thesis presents two EEV tools for the Linux kernel: (1) $\mathcal{L}$-SAP: an EEV tool for **Lock/Unlock Pairing** (Chapter 6) to verify the correct pairing of mutex lock and spin lock with their corresponding unlocks on all feasible execution paths, and (2) $\mathcal{M}$-SAP: an EEV tool for **Allocation/Deallocation Pairing** (Chapter 7) to verify the correct pairing of memory *allocation* with its corresponding *deallocations* on all feasible execution paths. The tools produce three categories of results: ($\mathcal{C}1$) automatically verified instances with correct pairings, ($\mathcal{C}2$) automatically verified instances with violations, i.e. a feasible path with a lock/allocation not followed by unlock/deallocation, and ($\mathcal{C}3$) instances where the automated verification is inconclusive.

$\mathcal{L}$-SAP produces for each verification instance the following verification-critical evidence: the Matching Pair Graph (MPG) (Section 4.1) to assist with the inter-procedural reasoning by identifying relevant functions and their interactions, and the Event Flow Graph (EFG) (Chapter 5) to assist with intra-procedural reasoning for each relevant function. Similarly, $\mathcal{M}$-SAP produces for each verification instance the following evidence: the Memory Taint Graph (MTG) (Section 4.2) to assist with the inter-procedural reasoning by identifying relevant functions and their interactions, the Event Flow Graph (EFG) (Chapter 5) to assist with intra-procedural reasoning for each relevant function, and the Points-To Graph (PtG) (Section 4.3) to assist with data flow reasoning by identifying the points-to relations between the pointers of interest at different locations in source code. In addition, we have developed smart views to enable interactive reasoning where the analyst can interact visually and programmatically to augment or refine the produced evidence.

We applied the $\mathcal{L}$-SAP tool to verify three recent versions of the Linux operating system (i.e., 3.17-rc1, 3.18-rc1, and 3.19-rc1) with altogether 37 MLOC and $66,609$ verification instances. We applied the $\mathcal{M}$-SAP tool to verify version (3.17-rc1) of the Linux operating system with 12 MLOC and $2,963$ verification instances. We developed our EEV tools using Atlas [3, 37], a graph database platform for program analysis and comprehension. The query language and the interactive visualization capabilities of Atlas are crucial to build and apply visual models for software verification.

The results (Sections 6.4 and 7.4) include a comparison with the state-of-the-art Linux Driver Verification (LDV) tool [5], effectiveness of the proposed visual models (MPG, MTG, PtG, and EFG) as verification-critical evidence through representative examples of the three result categories produced by $\mathcal{L}$-SAP and $\mathcal{M}$-SAP, the discovered bugs, and a website [6] for public cross-checking of verification results with evidence for all the verification instances.

1.3 Thesis Contribution

The key contributions are:

1. A rigorous notion of verification-critical evidence using visual models. The evidence enables human-machine collaboration to achieve high efficiency and accuracy without exorbitant manual effort (Chapter 4).

2. A novel linear time control flow graph pruning algorithm to compute a compact derivative of control flow graph (CFG) called Event Flow Graph (EFG) (Chapter 5).

3. A scalable and accurate evidence-enabled verification approach for lock/unlock pairing and its implementation in $\mathcal{L}$-SAP tool (Chapter 6).

4. A scalable and accurate evidence-enabled verification approach for allocation/deallocation pairing and its implementation in $\mathcal{M}$-SAP tool (Chapter 7).

5. A comprehensive empirical study and evaluation on recent versions of the Linux kernel to showcase the scalability and accuracy of the evidence-enabled verification tools: $\mathcal{L}$-SAP (Section 6.4) and $\mathcal{M}$-SAP (Section 7.4).

1.4 Thesis Organization

The remainder of the thesis is organized as follows. We first present the motivation and major challenges of static software verification in Chapter 2. Next, Chapter 3 describes the related work on static verification techniques for lock/unlock pairing and allocation/deallocation pairing. Chapter 4 presents software visual models as verification-critical evidence used in lock/unlock pairing and allocation/deallocation pairing. Chapter 5 describes in details the Event Flow Graph (EFG) and the linear time algorithm for deriving the EFG from its corresponding CFG. Chapters 6 and 7 present the automated verification algorithms used in $\mathcal{L}$-SAP tool for lock/unlock pairing and $\mathcal{M}$-SAP tool for allocation/deallocation pairing, respectively.

Chapters 6 and 7 also present the empirical assessment of $\mathcal{L}$-SAP and $\mathcal{M}$-SAP, the produced evidence, and the discovered bugs. Chapter 8 concludes with an overall summary of the benefits of the EEV approach and a discussion of the extensibility of the EEV approach.

CHAPTER 2. MAJOR CHALLENGES AND MOTIVATION

In this chapter, we discuss the major challenges for lock/unlock pairing and allocation/deallocation pairing analyses that motivated the design of our evidence-enabled verification approaches.

2.1 Challenges Overview

Synchronization problems and memory leaks can be catastrophic - a business-transaction server can crash resulting in a big financial loss, or a safety-critical control system can halt causing loss of lives. With multi-threading and event-driven processing, it is challenging to ensure resilience of software systems to such problems. These problems can elude dynamic analyses and regression testing because their occurrence often depends on intricate sequences of low-probability events [41]. Performing multiple runs of a program to examine all possible behaviors is prohibitively expensive and time-consuming. Thus, automated static analyses are crucial to complement testing and dynamic analyses.

Consider the allocation/deallocation pairing analysis: an allocation event A is paired with a deallocation event D iff D can deallocate the memory block allocated by A. The existence of unpaired allocations on feasible execution paths results in memory leaks. The lock/unlock pairing analysis is similar where the lock event L is paired with an unlock event U iff U can release the lock acquired by L. The existence of unpaired locks on feasible execution paths results in synchronization problems. This pairing of an allocation (lock) with its corresponding deallocations (unlock) on all feasible execution paths requires the following: (a) a *data flow analysis* to *map* each allocation (lock) to corresponding deallocations (unlocks), (b) a *control flow analysis* to *pair* each allocation (lock) with corresponding deallocations (unlocks), and

(c) a *feasibility analysis* to check the feasibility of an execution path on which an allocation (lock) is not ensued by a deallocation (unlock). A general-purpose, completely automated, and accurate analysis is intractable for each of the above three requirements.

The current formal verification approaches are *not* geared to produce evidence to empower and integrate human reasoning into the verification process. State-of-the-art research and commercial tools for MPV problems produce either very little evidence, voluminous evidence, or evidence that refers to the intermediate representation for the verification machinery but not to the source code. Such evidence is hard to decipher; it does not simplify cross-checking. Thus, another challenge is to formulate the evidence and build tools that can produce evidence.

2.2 Path Explosion

A Control Flow Graph (CFG) [14] is a graph representing all paths that the program may traverse during its execution. CFGs are central to any static analysis of the flow of execution and data-relationships in a computer program. The number of control flow paths grows exponentially with non-nested branch nodes. Our empirical study of the Linux kernel shows many functions with very large number of execution paths. For example, function `register_cdrom`[1] has 20 non-nested branch nodes. This path explosion increases with inter-procedural analysis as a path from the caller function splits into multiple paths in the called function.

To address the path-explosion challenge, this thesis presents a novel CFG pruning technique that produces a compact derivative of CFG called Event Flow Graph (EFG) (Chapter 5). This is achieved by introducing an *equivalence relation* on the CFG paths to partition/group them into equivalence classes. It is then sufficient to perform the verification or pairing analysis on these equivalence classes (i.e., paths in EFG) rather than on the individual paths in a CFG. We have adopted the graph algorithm by Tarjan *et al.* [86] to come up with a graph compaction algorithm to form the equivalence classes efficiently. Although the number of paths in a CFG is very large, the number of path equivalence classes is quite small, as seen from our results of the Linux kernel, and that enables an efficient and accurate path-sensitive analysis.

[1] http://lxr.free-electrons.com/source/drivers/cdrom/cdrom.c?v=3.19#L586

2.3 Inter-procedural Analysis

In several lock/allocation instances in the Linux kernel, the corresponding unlocks/deallocations occur across multiple functions, thus requiring inter-procedural analysis. The pairing algorithm must handle the following cases: (1) the locking/allocation function A calls the unlocking/deallocation function D through a call sequence, (2) the locking/allocation function A is called by the unlocking/deallocation function D through a call sequence (the reference to the lock object (allocated memory block) is returned upward the reverse call sequence from A to D), (3) A and D are called by a common parent, or (4) A and D are called asynchronously sharing the lock object (allocated memory block) as a heap object. A combination of these cases can happen where one instance of locking/allocation in A is paired with multiple unlocking/deallocation instances in several multiple functions $D_1, D_2, ..., D_k$ on different paths. The pairing algorithm (Chapters 6 and 7) addresses the inter-procedural analysis challenge by generating *compact* function summaries to enable efficient inter-procedural and context-sensitive analysis.

2.4 Feasibility Analysis

A path on which a lock (allocation) is not paired with an unlock (deallocation) may or may not be an error depending on whether the path is feasible or not. Thus, analyzing feasibility of paths is important to avoid false positives. Analyzing path feasibility can incur exponential computation [23, 39, 64, 67, 88] because it involves checking satisfiability of branch conditions governing a path. Typical complications for path feasibility analysis are correlations between branch conditions, loops, and inter-procedural paths.

To address the feasibility analysis challenge, first, our pairing algorithm (Chapters 6 and 7) minimizes the need for performing feasibility analysis by sequencing the analysis steps to produce at the end, exactly those path equivalence classes on which a lock (allocation) is not paired with an unlock (deallocation). The feasibility analysis is required only for these cases to avoid false positives. Second, the EFG (Chapter 5) optimizes the path feasibility analysis by identifying a small necessary set of branch conditions relevant for path feasibility. Third, the analysis calculates Boolean expressions that express the conditions under which each lock (allocation) is not paired with unlock (deallocation). Finally, our pairing analysis uses BDDs [92] to check the satisfiability of these expressions to determine whether the paths that contain the unpaired locks (allocations) are infeasible. We perform a simple automated check for path feasibility by applying an intra-procedural textual equality based analysis. The cases that are not amenable to the simple check are reported as inconclusive so that the human analyst can examine them further.

2.5 Pointer Analysis

Despite the advances in pointer analysis algorithms, they cannot be completely accurate because of the fundamental limitation. Typical accuracy hurdles for pointer analysis are pointer arithmetic operations on pointers, heap objects, function pointers, aggregate structures, offset-references, complex pointer references. For example, a highly accurate bit representation to track a pointer loses accuracy when a pointer is passed to a linked list. Further complications are library and system calls. Godefroid and Lahiri [46] from Microsoft Research note "Indeed, in practice, symbolic execution of large complex programs is rarely fully precise due to external library or system calls, un-handled program instructions, pointer arithmetic, floating-point computations, etc."

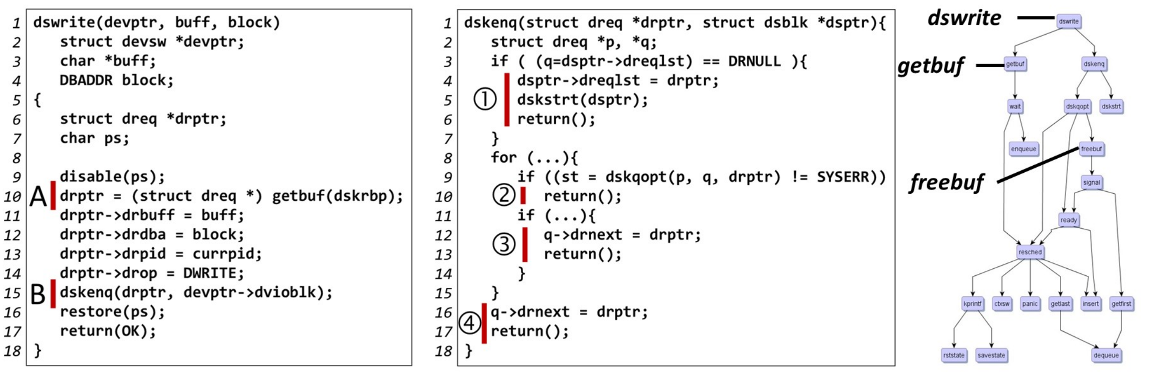
To provide scalable and sound analyses and conservatively mitigate inaccuracies of pointer analyses, the lock/unlock pairing analysis (Chapter 6) uses *type-based* analysis (Section 6.2.1). Our evaluation on multiple versions of the Linux kernel shows that only a tiny percentage (0.2%) of the lock instances cannot be paired using our type-based analysis. To achieve accuracy and scalability of the type-based analysis, the lock/unlock pairing analysis leverages an innovative algorithm [47]) (Section 4.1) to compute the minimum set of functions for the above four variants of inter-procedural analysis.

For allocation/deallocation pairing analysis, our goal is not to advance state-of-the-art pointer analysis, but rather to provide a working analysis that supports our verification-critical evidence framework. Our pairing analysis (Chapter 7) uses a conservative pointer analysis (Section 7.2.1) that is limited to handling cases and implementation patterns found in the

Linux kernel such as the cases where the allocated memory blocks do not escape through function pointers or heap objects.

2.6 A Concrete Example to Illustrate the Need for Verification-Critical Evidence

The current formal verification approaches [1,2,5,39,41,83,89,98] are not geared to produce evidence to empower and integrate human reasoning into the verification process. They work mostly as automated black boxes with very little support for cross-checking. The following example brings out the need for verification-critical evidence to reason about and solve complex software verification problems. The example in Figure 2.1 is from the XINU [12] operating system. XINU is a small system with about five thousand lines of code and 200 functions. However, XINU is complex enough to bring out key verification challenges. This example involves pairing an allocation with corresponding deallocations. In XINU, `getbuf` and `freebuf` are respectively the allocation and the deallocation functions.



```
dswrite(devptr, buff, block)
   struct devsw *devptr;
   char *buff;
   DBADDR block;
{
   struct dreq *drptr;
   char ps;

   disable(ps);
A  drptr = (struct dreq *) getbuf(dskrbp);
   drptr->drbuff = buff;
   drptr->drdba = block;
   drptr->drpid = currpid;
   drptr->drop = DWRITE;
B  dskenq(drptr, devptr->dvioblk);
   restore(ps);
   return(OK);
}
```

```
dskenq(struct dreq *drptr, struct dsblk *dsptr){
   struct dreq *p, *q;
   if ( (q=dsptr->dreqlst) == DRNULL ){
①     dsptr->dreqlst = drptr;
      dskstrt(dsptr);
      return();
   }
   for (...){
      if ((st = dskqopt(p, q, drptr) != SYSERR))
②        return();
      if (...){
③        q->drnext = drptr;
         return();
      }
   }
④  q->drnext = drptr;
   return();
}
```

(a) Function `dswrite` **(b) Function `dskenq`** **(c) Call Graph of `dswrite`**

Figure 2.1 An example to illustrate the need for evidence to reason about the possibility of a memory leak in the function `dswrite`

Consider the following verification instance: function `dswrite` (Figure 2.1(a)) calls `getbuf` but it does not call `freebuf`. The verification problem is to match the `getbuf` function call in `dswrite` with its corresponding `freebuf` function call(s). As seen from the `dswrite` code in Figure 2.1(a), `dswrite` calls and passes the pointer `drptr` to the allocated memory to function `dskenq`. As shown in Figure 2.1(b), function `dskenq` has four paths with different exits from the function.

To the best of our knowledge, all the existing static analysis would either fail or report a memory leak for this example. Moreover, these tools produce very little evidence or voluminous evidence which is hard to decipher. With only one exception [83], the evidence refers to the intermediate representation for the verification machinery but not to the source code. Let us improvise and consider a hypothetical tool that produces the following evidence: (1) a call graph of function `dswrite` (Figure 2.1(c)), (2) markings for the relevant source code line numbers (Figure 2.1(b)) in function (`dskenq`) where the tool detects the leak.

Let us now understand the difficulties the analyst would face even with this improvised evidence. We will discuss the difficulties along each of the four paths as shown in function `dskenq` (Figure 2.1(b)).

1. **Path 1:** `drptr` is passed to function `dskstrt`; it does not deallocate the allocated memory. But, the analyst cannot conclude that it is a memory leak because on the same path `drptr` is assigned to `dsptr->dreqlst` where `dsptr` (passed as a parameter to `dskenq` from `dswrite`). The analyst would then need to go back to `dswrite` and figure out that `dsptr` points to a global data structure. The verification becomes very challenging for the analyst once the pointer is globally accessible. Any function could accesses the pointer to deallocate the memory. The call graph (Figure 2.1(c)) does not help at this point. Without any more evidence, the analyst faces the arduous task of sifting through all functions to complete the verification.

2. **Path 2:** The pointer to the allocated memory `drptr` is passed to function `dskopt`. This tells the analyst that there is a problem in `dskopt`. The corresponding code to `dskopt` is not shown but we point out that it has 7 paths including paths on which the pointer to the allocated memory `drptr` is assigned to a global structure. Thus, the analyst has a lot more work to do to complete the verification. However, there is no additional evidence to help the analyst with the remaining work.

3. **Paths 3 and 4:** The analyst faces similar challenge as in Path 1 because the pointer to the allocated memory `drptr` is assigned to a global structure. The analyst also has to manually trace the data flow through `dskenq` to figure it out.

Summary. This example exhibits multiple challenges due to global accessibility of the pointer to the allocated memory. It poses a monumental task for the analyst to complete the verification.

This is also an example of an interesting pattern that should be leveraged for a tractable verification. On several execution paths in `dskenq`, there is a consistent pattern of the allocated memory pointer `drptr` being assigned to a global structure. Moreover, it is the same global structure on all paths. It looks like that the developer has done so on purpose with a design in mind. The design is not documented and the improvised evidence does not provide a clue. The call graph evidence in Figure 2.1(c) is problematic because: (1) there are functions irrelevant to the verification called by `getbuf` and `freebuf`, and (2) there are functions relevant to the verification but not captured. If the analyst does not figure out that the allocated memory pointer on some paths in `dskenq` becomes globally accessible, the analyst could conclude that it is a memory leak which is incorrect.

To address these difficulties, the evidence must simplify human reasoning: (1) by identifying relevant functions and their interactions, (2) by assisting with intra-procedural reasoning for each relevant function, and (3) by assisting with data flow reasoning by identifying the points-to relations between the pointers of interest at different locations in source code. In Chapter 4, we revisit the motivation example and present multiple visual models as inter-procedural, intra-procedural, and data flow evidence to help the analyst reason, complete, and cross-check the verification instance.

CHAPTER 3. LITERATURE REVIEW

In this chapter, we first survey previous static verification approaches for lock/unlock and allocation/deallocation pairing analyses and reflect on the shortcoming of their results, scalability, and accuracy, then we discuss their produced evidence. Next, we present previous work on Linux kernel verification.

3.1 Data Races and Deadlocks Detection

Detecting data races and deadlocks is a well-known challenging problem [41]. Although not directly related, there are many papers on dynamic verification techniques to detect race conditions and deadlocks [27, 31, 40, 49, 50, 61, 62, 71, 75, 80, 96]. However, Performing multiple runs of a program to examine all possible behaviors is prohibitively expensive and time-consuming. Thus, automated static verification techniques are crucial to complement testing and dynamic verification techniques.

Warlock [81], Extended Static Checking [38] (ESC) and ESC/Java [57] are static race detection approaches for C, Modula-3 and Java respectively. These approaches utilize a theorem prover to find race conditions. However, these approaches require annotations to inject knowledge into the analysis and to reduce the number of false positives. Thus, applying these approaches to large code bases such as the Linux kernel would require time-consuming and tedious annotation effort and at the end annotations can be erroneous and checking their correctness would be a daunting task for millions of lines of complex code.

RacerX [41] and Relay [89] are static lockset based analyses for C code that scale to large real world programs including the Linux kernel. RacerX [41] uses a top-down approach to compute *absolute locksets* (the set of locks held by the program) at each program point. RacerX is able to run on an older version of the Linux kernel (v2.5.62 - 1.8 MLOC) in tens of minutes. In contrast, Relay [89] uses a bottom-up approach to compute *relative locksets*, which describes the changes in the locks being held relative to the function entry point, at each program point. Relay was able to analyze an older version of the Linux kernel (v 2.6.15 - 4.5 MLOC) in 72 hours. RacerX reports that in order to scale, the analysis discards valuable information, such as truncating the function summaries. Consequently, discarding possible races. Moreover, both approaches (RacerX and Relay) require significant post-processing of large volume of warnings/false positives.

Saturn [39] is considered a scalable static analysis engine that is both sound and complete with respect to the user-provided analysis script (abstraction), written in its Calypso language. ESP [36] is a path-sensitive analysis tool that scales to large programs by merging superfluous branches leading to the same analysis state. However, the lock analysis script bundled with Saturn and ESP is neither sound nor complete, most notably because of its lack of global alias analysis and incomplete function summaries for inter-procedural analyses.

Locksmith [73,74] is a sound static race detector for C. It uses a constraint based technique to infer the correlation of memory locations to the locks that protect them. If a shared location is not consistently protected by the same lock, a race is reported. Locksmith analyzes source code fast. However, it finds superficial errors and it produces a number of false alarms, which is about 90% on the device drivers of an older version of the Linux kernel and about 98% on some POSIX applications.

Cho *et al.* [29] proposed a lock/unlock pairing analysis that combines an inter-procedural analysis and dynamic checking for better detection of races and deadlocks. Their analysis uses dynamic checking to compensate for imperfections in their static analysis. However, the proposed dynamic analysis introduces an overhead for the overall analysis. Moreover, the lock/unlock pairing mechanism has been only applied to code orders of magnitude smaller than the Linux kernel.

The Linux Driver Verification tool (LDV) [5] is currently the top-rated tool in the software verification competition (SV-COMP) [18–20] in the Linux device drivers verification category. LDV uses Blast [21] model checker to check for memory leaks and synchronization problems. However, LDV takes 60 hours for lock/unlock pairing in a recent version of the Linux kernel.

3.2 Static Memory Leak Detection

Memory leak detection is a well-known challenging problem. Although not directly related, there are number of reported attempts on memory leak detection using dynamic analysis [24,33, 66,69,100,101]. There has been a lot of research devoted to checking memory leaks statically [1, 5, 52, 55, 83, 97].

Saturn [97] models the memory-leak detection problem as a boolean satisfiability problem, then uses a SAT solver to identify memory-leaks. Its analysis is context-sensitive and intra-procedurally path-sensitive. Saturn checks for specific type of memory leaks: a memory block that is allocated in function p and is never escaped and deallocated in p is considered a memory leak. Thus, Saturn handles a very small subset of the allocations in the Linux kernel. Saturn is able to run on an older version of the Linux kernel (v2.6.10 - 5MLOC) in 23 hours without parallelization.

FastCheck [28] uses guarded value-flow analysis to detect memory leaks. It tracks the flow of values through top-level pointers only. It is fast but limited to analyzing allocation sites whose values escapes to (flow into) top-level pointers only. Through our study in the Linux kernel, most of the memory leak cases reported in this thesis will go undetected with FastCheck.

Athena [55] finds inter-procedural path-sensitive faults (including memory leaks) guided by user specifications. Sparrow [52] is a static analyzer that relies on abstract interpretation to detect memory leaks in C programs. It separately analyzes each function's memory behavior into a parameterized summary that is used in analyzing its call-sites. Both approaches (Athena and Sparrow) have been applied to programs order of magnitude smaller than the Linux kernel and it is not clear how scalable these tools are to larger code bases.

Clang [1] finds memory leaks in C and Objective-C programs based on symbolic execution. Being intra-procedural, it assumes unknown or symbolic values for the formal parameters of a function and the returned values from its call-sites. Therefore, all inter-procedural leaks can go undetected. Clang does not scale-well to large code bases. Moreover, LLVM [7], its core compiler, does not succeed in compiling the recent versions of the Linux kernel.

Saber [83] is a static detector for memory leaks in C programs. It performs a sparse value-flow analysis to track flow of values (escapes), from allocation to deallocation sites, through top-level and address-taken pointers. Saber does not track escapes (values-flows) to global variables. Although, Saber is the most recent memory leak detector, it cannot be applied to large code bases as the Linux kernel due to the following challenges: (1) Saber uses Open64 [8] compiler which cannot link the whole program static single assignment intermediate representation of the Linux kernel in its inter-procedural optimization module due to some non-trivial compilation issues (e.g., assembly language mixed with C), and (2) Saber uses Andersen's pointer analysis [15] for building value-flow dependencies. Although Andersen's analysis is well-studied algorithm, it is hard to scale to whole pointer analysis of large code bases as the Linux kernel (12 MLOC). Some heuristics may be applied to speed up and scale Saber's pointer analysis by performing field-insensitive analysis or mixing with unification-based [35] analysis, however, this will drastically affect the detection accuracy.

The Linux Driver Verification tool (LDV) [5] is also used to check for memory leaks. However, LDV takes 30 hours for allocation/deallocation pairing in a recent version of the Linux kernel.

3.3 Evidence-Enabled Verification

Saturn [39, 98] provides a minimalist evidence that consists of the source correspondence (i.e., file path and line numbers) for the checked instances. For example, if a memory leak is detected, Saturn will point-out the source correspondence for the allocation site only. This would require a huge effort of cross-verification, especially, if the corresponding deallocation calls for the allocated memory occur many levels down the call chain from the allocation function.

RacerX [41] and Relay [89] require significant post-processing of large volume of proofs/reports to produce comprehensible evidence for manual validation.

The Linux Driver Verification tool (LDV) [5] produces a monumental amount of non-visual complex evidence that require significant processing to produce the source correspondence of where the problem occurs.

Clang [1] finds memory leaks in C and for each detected memory leak, Clang generates an evidence that consists of multiple HTML pages. Each HTML page corresponds to the source code of a function related to the detected memory leak. Clang also draws segments between the different statements corresponding to the execution path where the detected memory leak occurs. In this case, the cross-verification process is tedious and error-prone, especially with inter-procedural memory leaks given that the analyst is limited to one function at a time.

Saber [83] produces a set of graphs , for each detected memory leak, that includes: program assignment graph, constraint graph, and value-flow graphs. However, this evidence corresponds to the intermediate representation of source code (i.e., static single assignment). Moreover, the evidence is difficult to read given that it has no source correspondence to the actual source code where the memory leak occurs.

Coverity [2] is a well-known commercial tool that offers static source code analysis technology that finds critical defects and security vulnerabilities in C/C++ and Java source code. For detected memory leaks and synchronization problems, Coverity annotates the source code with the events leading to the error. This produced evidence poses a usability barrier to the analyst, especially for inter-procedural errors the occur across multiple source files.

To the best of our knowledge, we are not aware of any evidence-based verification platform that is geared to produce comprehensible, compact, and visual evidence to empower and integrate human reasoning into the verification process.

3.4 Linux Kernel Verification

Linux dominates the server operating systems market. Many embedded devices such as smart phones run Linux as kernel. There is an increasing need for automatic verification of Linux components.

Microsoft had identified the device drivers as the most important source of failures in Windows. Therefore, the company has integrated the Static Driver Verifier (SDV) tool in the Windows Driver Development Kit that uses the SLAM [17] verification engine. For Linux, an industry-funded verification project of the size of SDV does not exist. Linux verification has become an important research topic using program analysis, SMT solvers, model checking, and other software verification techniques [43, 48, 53, 54, 63, 70, 72, 94]. Despite many new advances, the recent competitions on software verification (SV-COMP) [18–20] show that state-of-the-art tool implementations have problems analyzing Linux.

This Thesis proposes EEV as an alternative approach to software verification of MPV problems. Our comparison results on recent versions of the Linux kernel (Sections 6.4 and 7.4) with the top-rated Linux Driver Verification (LDV) [5] tool proves the practicality, accuracy, and scalability of EEV.

CHAPTER 4. VISUAL MODELS AS VERIFICATION-CRITICAL EVIDENCE

The complexity of software is rooted in its own version of the *butterfly effect* [45, 59]. 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. In addition, the current formal verification approaches work mostly as automated black boxes with very little support for cross-checking [1, 2, 5, 39, 41, 83, 89, 98].

In this chapter, we propose *visual software models* as the key enablers for integrating automation with human intelligence to solve software verification problems where complete automation has remained intractable. The key innovation is a mathematically rigorous notion of verification-critical evidence that the machine abstracts from software to empower human to reason with. The machine and human have different strengths and weaknesses to complement each other. Even for automatically completed verification, we advocate human cross-checking. This is especially important for critical systems where failures of software can be catastrophic. The cross-checking should be simplified by automation to conserve human effort.

Specifically, we will present four visual software models:

1. The Matching Pair Graph (MPG) (Section 4.1) and Memory Taint Graph (MTG) (Section 4.2) for managing inter-procedural complexity due to impact propagation.
2. The Points-To Graph (PtG) (Section 4.3) to assist with data flow reasoning by identifying the points-to relations between the pointers of interest at different source code locations.
3. The Event Flow Graph (EFG) (Section 4.4) (derived from the Control Flow Graph (CFG)) for managing intra-procedural complexity of control paths explosion.

4.1 Matching Pair Graph (MPG)

By design, the matching pair graph (MPG) [47] is a directed graph with edges representing function calls and the roots of MPG are asynchronous functions; they either belong to different threads or some of them may be called through interrupts. The MPG captures the functions that are needed for verifying the correct pairing of lock/unlock and allocation/deallocation. The functions in the MPG are sufficient for the pairing analysis if the type of the pointer referencing the lock object or the allocated memory block is preserved and never changes (i.e., not casted to a different type). Now, let us discuss how the MPG can serve as valuable evidence and significantly simplify work for the analyst by revisiting the motivation example in Figure 2.1. Figure 4.1 shows the MPG corresponding to the allocated memory in function `dswrite`. From the several hundred XINU functions, the MPG has narrowed down the relevant functions to 6.

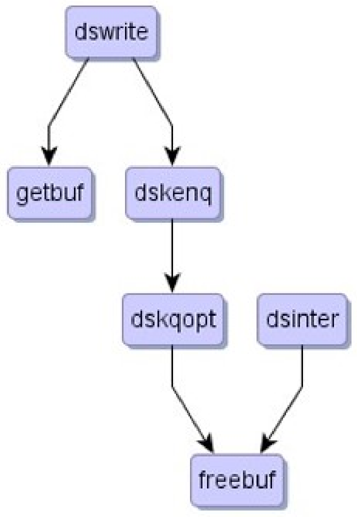


Figure 4.1 MPG evidence for the allocated memory in `dswrite`

Besides producing a small set of functions, the MPG provides other very valuable pieces of evidence. The MPG shows a call chain from `dswrite` to `freebuf` which indicates the possibility of execution paths on which the pointer to the allocated memory is passed as a parameter, eventually to function `dskopt` which deallocates the memory. More importantly, the MPG includes function `dsinter` which turns out to be a crucial clue. Function `dsinter` is not connected to `dswrite` by forward or reverse call chains. Strangely, `dsinter` calls `freebuf` but not `getbuf`. It is actually a critical clue.

The analyst can hypothesize that `freebuf` in `dsinter` can potentially pair with the `getbuf` in `dswrite` and since `dsinter` and `dswrite` are roots of the MPG, they work asynchronously and communicate with each other by sharing a global data structure `D`. To prove the hypothesis the analyst must locate `D` and complete the verification. The analyst has a good suspect for `D`, namely the data structure to which the allocated memory pointer `drptr` is assigned. This makes it quite easy for the analyst to complete the verification.

This example of the memory allocation instance in `dswrite` illustrates how the evidence from MPG can be of tremendous assistance to the analyst. It presents a case not amenable to automation because of the invisible control due to asynchronous processing. The data flow is also invisible because the allocated memory pointer `drptr` is inserted in a global linked list `D`. Function `dsinter` draws the pointers from the list and deallocates the memory for each pointer. And the fact that it does so until the list becomes empty shows that it is not a memory leak.

This is not a wayward example. It stems from the well-known *producer-consumer* pattern [85], a classic example of a multi-process synchronization. In Section 6.2.2, the MPG is used in our lock/unlock pairing analysis.

4.2 Memory Taint Graph (MTG)

The Memory Taint Graph (MTG) is a directed graph where the nodes correspond to functions and the edges are call relations. The MTG captures the set of functions where the object of interest associated with the MPV instance has been referenced, communicated or escaped. There are three communication ways: passing a reference through a parameter, returning a reference to the parent via a return statement or through a parameter, or referencing it though a global variable. Unlike the MPG, the MTG works with the MPV instances where the type of the pointer referencing the object of interest is not preserved while communicating or escaping the object. That means, the object is referenced by pointers of different types than the initial type. The MTG for allocation/deallocation pairing of object o contains the set of functions visited/analyzed while tracking references to o via pointer analysis. Nevertheless, the MTG will not be able to capture functions that are involved in obscure flows through interrupt driven.

The MTG can serve as valuable evidence and significantly simplify work for the analyst by capturing how the object of interest has been communicated/escaped through the program. Figure 4.2 shows the MTG corresponding to the allocated memory in function `dswrite`. The analyst can easily see that the allocated memory block in `dswrite` has been passed as a parameter to `dskenq` and then to `dskqopt` and finally to `freebuf`. In Section 7.2.1.13, we will use the concept of MTG in our allocation/deallocation pairing analysis.

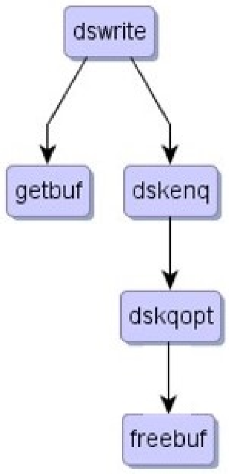


Figure 4.2 MTG evidence for the allocated memory in `dswrite`

4.3 Points-To Graph (PtG)

The Points-To Graph (PtG) is a directed graph where the nodes and edges are as follow: The nodes are:

- Pointer node that represents a program pointer. In our node representation, the pointer node can be one of the following:
  - An *id pointer* (IDP) node denotes a directly accessed pointer with no field references. For example, the pointers `ptr` and `q` are IDPs in the expression `ptr = q`.
  - A *field-reference pointer* (FRP) denotes a pointer that is being referenced/accessed through its containing structure using the pointer/structure de-reference operations: → or •. For example, the pointer `a->b->c` is a FRP and its containing structure is `a->b`. In turn, `a->b` is a FRP pointer and its containing structure is `a` which is an IDP.

- Memory Location (MLoc) node that represents a location in memory that can be referenced or pointed-to by pointer nodes. The MLoc node can also represent an Allocated Memory Block (AMB) that corresponds to an allocated memory block via an allocation call. The AMB node is associated with an address corresponding to the source code location of the allocation callsite.

There are two kinds of directed edges:

- Points-to Edge: A points-to edge ($a \rightarrow b$) from node a to node b represents that the pointer represented by a *points-to* the pointer represented by b.

- Field Edge: A field edge ($C \dashrightarrow k$) from node C to node k denotes that the pointer represented by k is a field in the structure represented by the pointer C.

Section 7.2.1 describes the details of mining, building, and the transformation of points-to graphs. Now, let us discuss how the PtG can serve as valuable evidence and significantly simplify work for the analyst by revisiting the motivation example in Figure 2.1. Figure 4.3 shows the PtG after processing line 4 in function `dskenq` with respect to the allocated memory block AMB at line 10 in function `dswrite`.

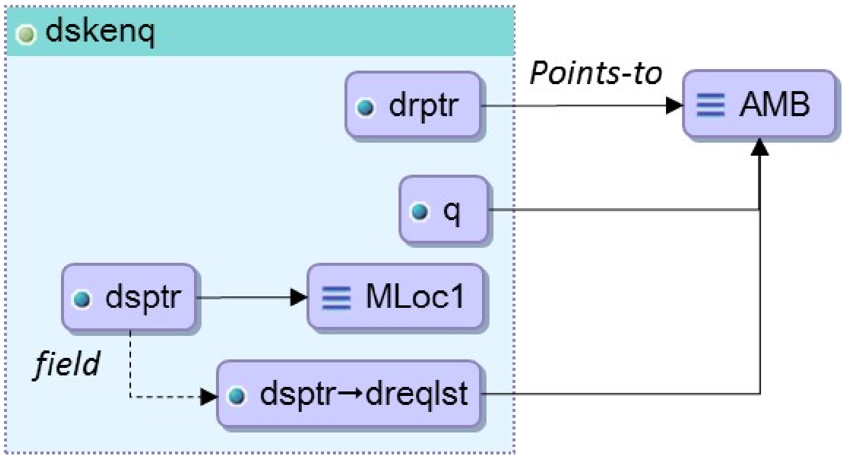


Figure 4.3 PtG evidence for the allocated memory in `dswrite` at line 4 of function `dskenq`

The PtG provides valuable pieces of evidence. The PtG shows the allocated memory block at line 10 of `dswrite` represented by the AMB node. It also shows all the pointers that directly and indirectly point to that block. The analyst can check whether any of the pointers at line 4 of function `dskenq` points-to the AMB. For example, the analyst can see that pointers `q` and `dsptr->dreqlst` point-to the allocated memory block and may be able to deallocate it. The analyst can also see that `dsptr` can be used to deallocate the memory block through the field `dreqlst`. This makes it quite easy for the analyst to complete or validate the verification. This example illustrates how the evidence from PtG can be of tremendous assistance to the analyst.

4.4 Event Flow Graph (EFG)

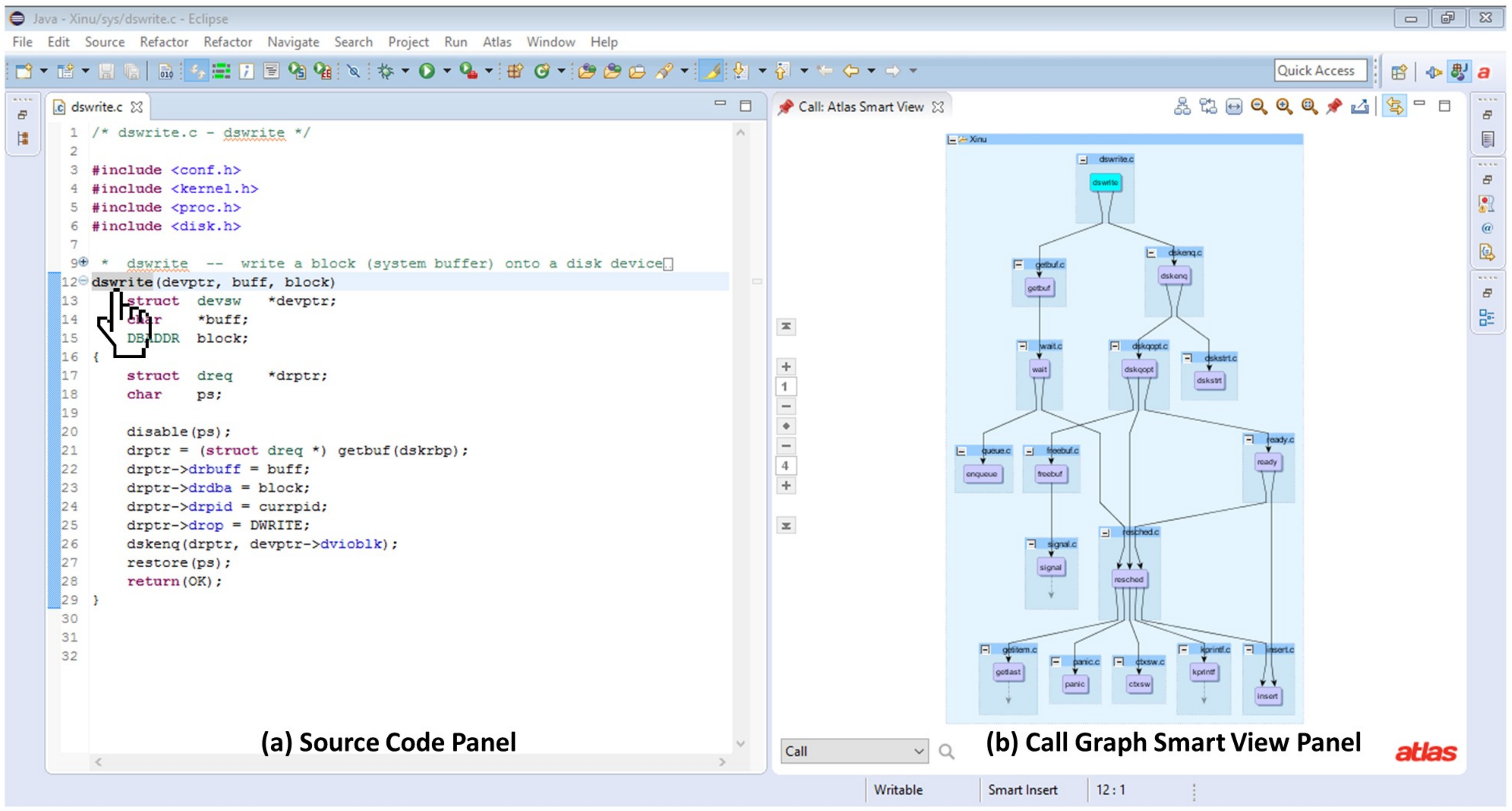
The CFG is valuable for understanding program behaviors. However, in real-world software the CFG can be very large and complex. This thesis presents a compact derivative of the CFG, called the EFG. Mathematically, the EFG defines an *equivalence relation* on CFG paths, where each path in the EFG corresponds to a multitude of CFG paths with identical traces of events of interest. The EFG has two major advantages over CFG: (a) although the number of paths in a CFG can be exponentially large, the essential information to be analyzed is captured by a small number of equivalence classes, and (b) checking path feasibility becomes simpler.

In Chapter 5, we present two important applications to show the use of the EFG as a powerful program comprehension tool: (a) a verification problem for the Linux kernel, and (b) a side-channel vulnerability detection problem for Java bytecode. We have built an interactive tool for creating EFGs for C, Java, and Java bytecode. While CFGs can be complex with exponentially many paths, we have developed an efficient algorithm to transform a CFG to an EFG. The algorithm is linear with respect to the number of CFG nodes and edges.

4.5 Enabling Technology & Interactive Reasoning

Interactive reasoning can be performed using visual models and their source correspondence. The interactions can go from one visual model to another one, go from a visual model node to its corresponding source code, or go from source code to a corresponding visual model. One can click on a function node in a visual model (e.g., MPG) to open up the EFG for the selected function in MPG and observe the control flow paths within the function. The EFG nodes correspond to statements in the source code. One can click on a EFG node to observe the corresponding source code. Similarly, one can click on a pointer in source code to show the corresponding PtG. Clicking a points-to relation takes the user to the source code expression where the points-to relation has been created.

To enable such interactive reasoning, we built our EEV machinery on top of Atlas [3, 37] which is a graph database platform for program analysis and comprehension. It provides a shell and one can interactively (i.e., visually and/or programmatically) query and mine programs to gather additional evidence. Atlas also provides *Smart Views* to provide instant feedback and interactive software graph visual models as the analyst clicks on code artifacts or other visual models. A number of out-of-the-box smart views are provided for common queries for building call graphs, data flow graphs, type hierarchies, dependency graphs, and many other useful results. For example, when the analyst clicks on a function either in a visual model or source code, the call graph smart view (if selected) will instantly produce the call graph for the selected function. The produced smart view appears on a side tab that does not interfere with the analyst and keeps him focused on the current task. Figure 4.4(b) shows the call graph smart view for the selected function `dswrite` in the source code panel (Figure 4.4(a)).



```
/* dswrite.c - dswrite */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>

 *  dswrite  --  write a block (system buffer) onto a disk device
dswrite(devptr, buff, block)
    struct  devsw   *devptr;
    char    *buff;
    DBADDR  block;
{
    struct  dreq    *drptr;
    char    ps;

    disable(ps);
    drptr = (struct dreq *) getbuf(dskrbp);
    drptr->drbuff = buff;
    drptr->drdba = block;
    drptr->drpid = currpid;
    drptr->drop = DWRITE;
    dskenq(drptr, devptr->dvioblk);
    restore(ps);
    return(OK);
}
```

Figure 4.4 Call graph smart view for selected function `dswrite`

CHAPTER 5. EVENT FLOW GRAPH FOR PROGRAM COMPREHENSION

The Control Flow Graph (CFG) is valuable for understanding program behaviors. However, in real-world software the CFG can be very large and complex. This chapter presents a compact derivative of the CFG, called the Event Flow Graph (EFG). Mathematically, the EFG defines an *equivalence relation* on CFG paths, where each path in the EFG corresponds to a multitude of CFG paths with identical traces of events of interest. We present two important applications to show the use of the EFG as a powerful program comprehension tool: (a) a verification problem for the Linux kernel, and (b) a side-channel vulnerability detection problem for Java bytecode. Using the Atlas platform, we have built an interactive tool for creating EFGs for C, Java, and Java bytecode. While CFGs can be complex with exponentially many paths, we have developed an efficient algorithm to transform a CFG to an EFG. The algorithm is linear with respect to the number of CFG nodes and edges.

5.1 Introduction

The CFG [14] is a graph representing all paths that the program may traverse during its execution. CFGs are central to any static analysis of the flow of execution and data-relationships in a computer program. The number of control flow paths grows exponentially with non-nested branch nodes. Our empirical study of the Linux kernel shows many functions with very large number of execution paths. For example, function `register_cdrom`[1] has 20 non-nested branch nodes. This path explosion increases with inter-procedural analysis as a path from the caller function splits into multiple paths in the called function.

[1]http://lxr.free-electrons.com/source/drivers/cdrom/cdrom.c?v=3.19#L586

We define a compact derivative of the CFG called the Event Flow Graph (EFG). The purpose of the EFG is to simplify the corresponding CFG as much as possible while retaining all relevant execution traces for a given problem. We call a relevant execution trace an *event trace*. Defined for each CFG path, it is the sequence of nodes corresponding to events relevant for a given problem. Two CFG paths are *equivalent* if they have the same event trace. The EFG retains all event traces from the CFG but it has only one path for each distinct event trace. The non-branch nodes not relevant to the problem and the branch nodes not leading to distinct event traces are elided by the CFG to EFG transformation. We describe an efficient *linear-time* algorithm for this transformation.

We present two applications to show the fundamental importance of EFGs for program comprehension. The first application is lock/unlock pairing: verify that each lock event L is paired on all feasible execution paths with unlock events U that release the lock acquired by L. For this application, the relevant events are `Lock`, `Unlock`, and other program statements that pass the pointer to the lock object. The CFGs and EFGs, for pairing analysis of $66,609$ `Lock` instances in three recent Linux versions, are posted on a website [6]. The second application is detecting sophisticated vulnerabilities that can lead to algorithmic complexity (AC) or side-channel (SC) attacks [10]. For this application, the relevant events are loops with differential time or space behaviors.

The program comprehension scenarios for the two applications are as follows. For verifying the pairing between `Lock` and `Unlock`, program comprehension support is important for: (a) cross-checking results of automated pairing, and (b) human-in-the-loop pairing for difficult cases where automated pairing is intractable. For detecting AC or SC vulnerabilities, program comprehension support is important for: (i) hypothesizing potential AC or SC attacks, and (ii) proving or refuting each attack hypothesis. Apart from being a compact graph, the EFG is important for focusing on critical branch nodes. For the first application, it brings out the branch nodes critical for determining the feasibility of paths where a `Lock` is not followed by a corresponding `Unlock`. For the second application, it brings out the branch nodes critical for selecting paths for AC or SC attacks.

This chapter has the following key research contributions:

- A rigorous formulation of the EFG as a fundamental concept for reducing the cognitive burden by simplifying the CFG while retaining problem-specific execution behaviors.
- Two important applications to show the use of the EFG as a powerful program comprehension tool.
- A quantitative study of reductions afforded by the CFG to EFG transformation for real-world software.

The remainder of the chapter is organized as follows. We first present the motivation in Section 5.2. Next, Sections 5.3 and 5.4 present two applications to show the fundamental importance of EFGs for program comprehension. Section 5.5 describes the tool capabilities for constructing EFGs. Section 5.6 presents the linear-time algorithm for constructing the EFG from its corresponding CFG. Our empirical assessment of EFGs using three recent versions of the Linux kernel is presented in Section 5.7. Section 5.8 describes the related work. Finally, we conclude in Section 5.9.

5.2 Motivation

We present two examples to show the simplifications achieved by EFGs.

5.2.1 A Linux Example

The following example from the Linux kernel (v3.19) illustrates how an EFG greatly simplifies the verification of pairing between `Lock` and `Unlock`. Figure 5.1 shows an example of a CFG and its corresponding EFG. The EFG is constructed with respect to `Lock` and `Unlock` events. 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 EFG and the 7 paths are represented by a single path in the EFG. Thus, the EFG simplifies the verification task by compacting the CFG.

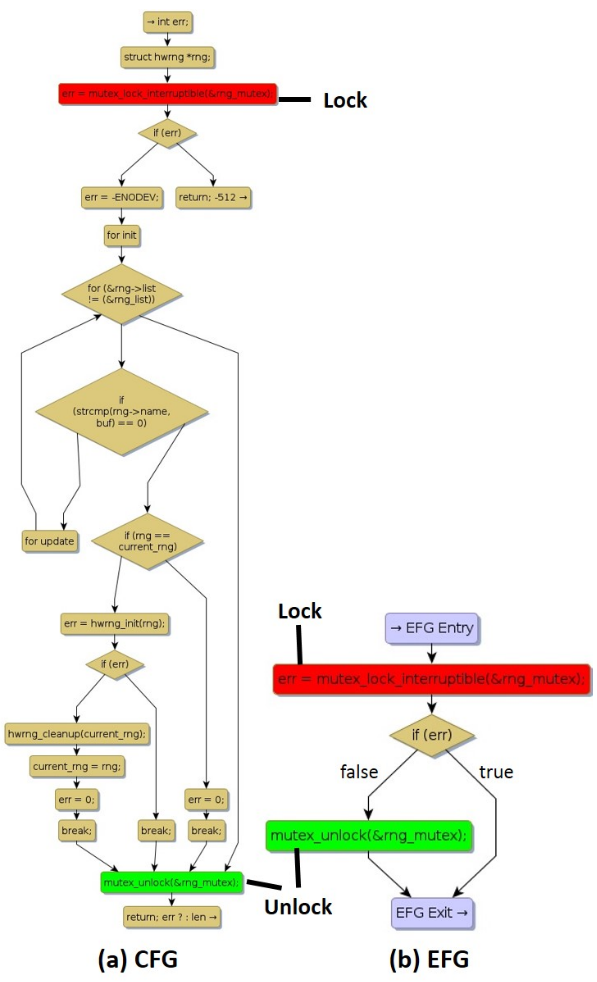


Figure 5.1 CFG and EFG for the function `hwrng_attr_current_store`

The EFG also simplifies the path feasibility check. As seen from the EFG in Figure 5.1(b), 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 EFG has retained only the condition that is necessary to verify the path feasibility, the other 4 conditions from the CFG are not retained in the EFG. If the lock is granted, then the particular condition is `false`. So, the `true` path in Figure 5.1(b) is not feasible and thus the pairing is correct.

Similar to this example, our empirical observation on real-world software shows that the set of problem-specific events is usually sparse and consequently EFGs are typically significantly smaller compared to the corresponding CFGs. A quantitative study of the Linux kernel is presented later in Section 5.7.

5.2.2 Loop Call Graph

The following example illustrates how EFGs can be used to focus on loops. As mentioned earlier, loops are important for detecting AC and SC vulnerabilities.

The *Loop Call Graph* (LCG) is a directed graph whose nodes are methods that contain intra-procedural loops and methods that reach loops through call chains. There is an edge from method m_1 to m_2 in the LCG if method m_1 calls method m_2. Further, an edge is colored if the callsite of method m_2 in method m_1 is located within an intra-procedural loop.

The LCG is an example of extending the EFG to capture relevant behaviors across methods. Within each method, we can have the EFG that captures loops by treating loop headers and callsites which may reach a method with a loop. The LCG enables us to explore and understand loops nested across methods. We have developed interactive tool support so that one can navigate through inter-procedural loops using the LCG and click on a method to view the intra-procedural EFG.

Figure 5.2 shows the LCG for TimSort [11], a defacto sorting algorithm implemented in several languages including Java and Python. In Java bytecode, TimSort contains 20 loops that are distributed among 11 different methods. When a method in the LCG is selected, the intra-procedural EFG shows loops shaded darker for each level of nesting (shown within a dashed callout in Figure 5.2. To recover loops in bytecode, we leverage the Decompiled Loop Identification algorithm presented by Wei *et al.* [91].

TimSort has many pieces to it with variants of sorting algorithms working together to achieve highly optimal performance depending on the size and ordering of the data to be sorted. The complex internal structure of TimSort is revealed by the EFG that focuses on loops.

5.2.2.1 Loop-based Intra-procedural EFG

An LCG helps to navigate through inter-procedural loops. By clicking on a LCG node, we can view the intra-procedural EFG. Figure 5.3 shows the CFG and Figure 5.4 shows the loop-based EFG. The highlighted branch points are the loop headers. The EFG retains three

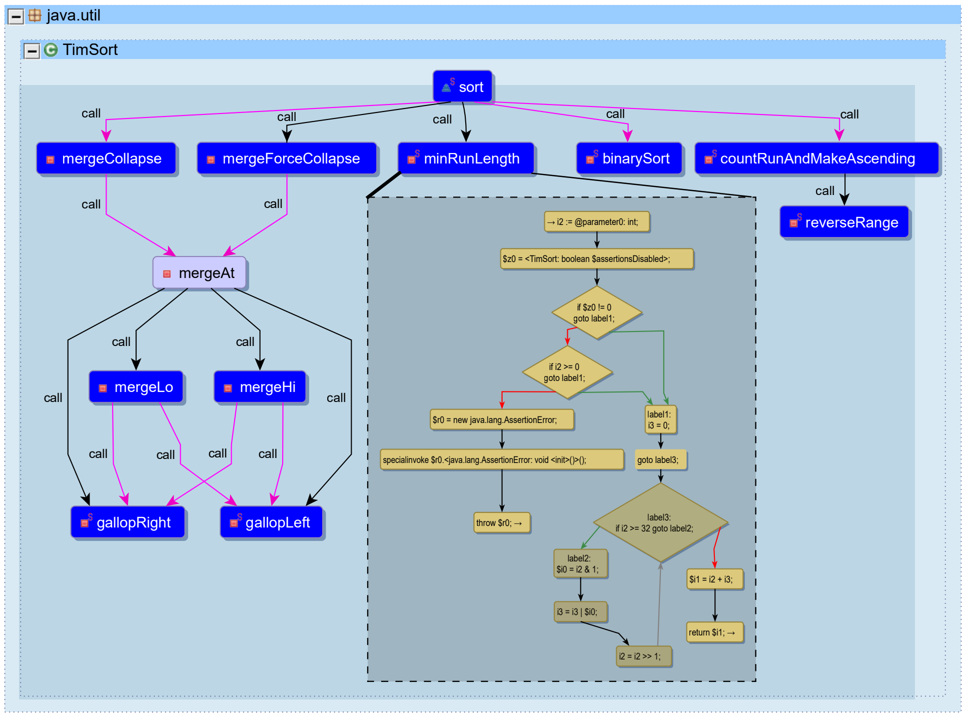


Figure 5.2 Loop Call Graph for TimSort Algorithm

other branch points because they are *differential branch points* with branches that are distinct with respect to loops. As discussed later, loops are important for detecting SC vulnerabilities. In a SC attack, the attacker controls a differential branch point using inputs to observe the difference in space or time behavior and learns a secret (e.g., a password).

5.3 Application: Verify Pairing

We present four instances to show different program comprehension scenarios. The first scenario is to verify correct pairing of `Lock` and `Unlock`. The second scenario is to argue that the pairing is not correct and thus it is a bug. The third scenario is to reason about a case when the `Lock` happens inside a loop, looks like a bug at a cursory glance but it is not a bug on a thoughtful review of the EFG. The fourth scenario is an EFG quirk due to a peculiar programming pattern. These instances are drawn from the *Matching Pair Graphs* (MPGs) [47] and the EFGs posted on our website [6] for each of the 66, 609 `Lock` instances from three recent versions (3.17-rc1, 3.18-rc1, and 3.19-rc1) of the Linux kernel. The MPG, defined for each `Lock`

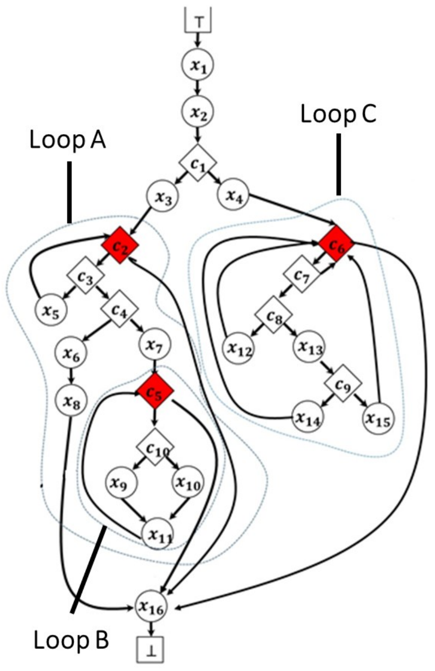


Figure 5.3 CFG with marked loop headers

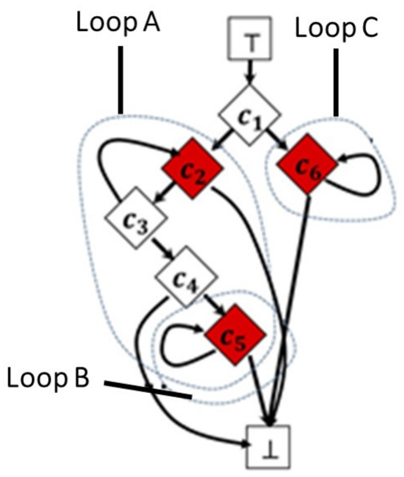


Figure 5.4 Loop-based EFG

instance, provides the call graph of relevant functions for the `Lock`. The EFGs are given for all the relevant functions.

5.3.1 Correct Lock-Unlock Pairing

Figure 5.5(a) shows the MPG for the lock in function `hso_free_serial_device`. Figures 5.5(b) and 5.5(c) show the EFGs for the MPG functions `hso_free_shared_int` and `hso_free_serial_device`, respectively.

In this example, it is easy to observe from the EFG of function `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 EFG of the called function `hso_free_shared_int` shows a matching unlock on all paths within the called function.

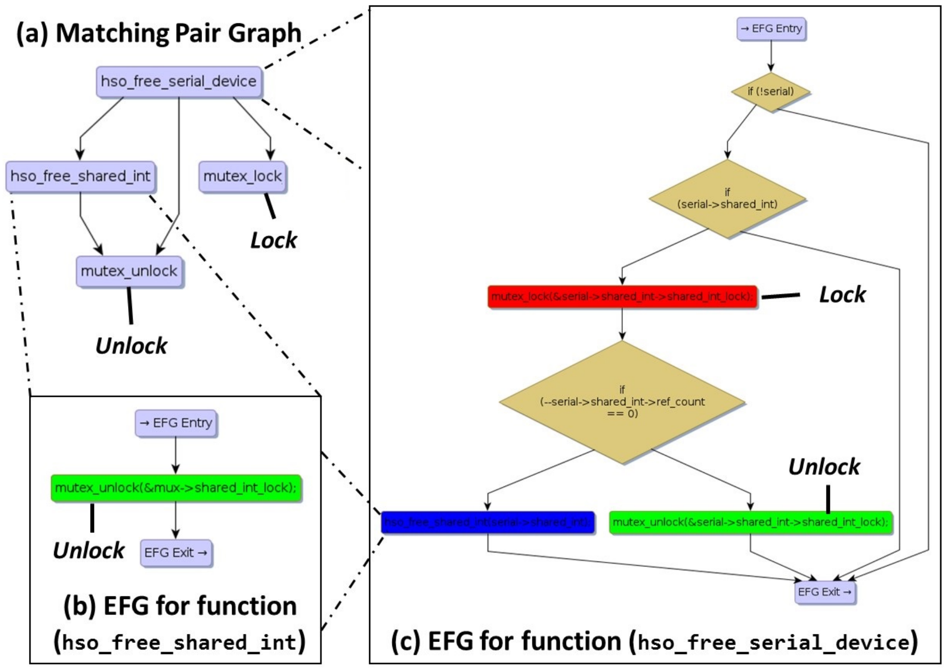

(a) Matching Pair Graph
hso_free_serial_device
hso_free_shared_int
mutex_lock
Lock
mutex_unlock
Unlock
→ EFG Entry
mutex_unlock(&mux->shared_int_lock);
Unlock
EFG Exit →
(b) EFG for function (hso_free_shared_int)
→ EFG Entry
if (!serial)
if (serial->shared_int)
mutex_lock(&serial->shared_int->shared_int_lock);
Lock
if (--serial->shared_int->ref_count == 0)
Unlock
hso_free_shared_int(serial->shared_int);
mutex_unlock(&serial->shared_int->shared_int_lock);
EFG Exit →
(c) EFG for function (hso_free_serial_device)

Figure 5.5 An example of correct pairing

5.3.2 Lock-Unlock Pairing Bug

Figure 5.6(a) shows the MPG for the lock in the function `toshsc_thread_irq`. Figure 5.6(b) shows the EFG for `toshsc_thread_irq`.

The EFG for `toshsc_thread_irq` shows a path on which the lock is not followed by an unlock. As seen from the EFG, the path is feasible if the boolean expression ($C_1\overline{C_2}$) is true. To complete the verification, one must verify that the boolean expression is satisfiable and conclude that the automatically reported violation is indeed a violation. This bug was reported to the Linux organization and it was fixed in later version.

5.3.3 Pairing Includes Loop

The EFG in this example is an unusual case. Figure 5.7(a) shows the MPG for the `Lock` in the function `destroy_async` and the EFG in Figure 5.7(b) shows that the `Lock` is matched correctly on two paths. However, there is a dangling `Unlock` upon the entry to the loop. This raises the question of whether there is another `Lock` before the loop, which would be required for a correct pairing. Since a separate EFG is created for each instance of `Lock`, the other `Lock`

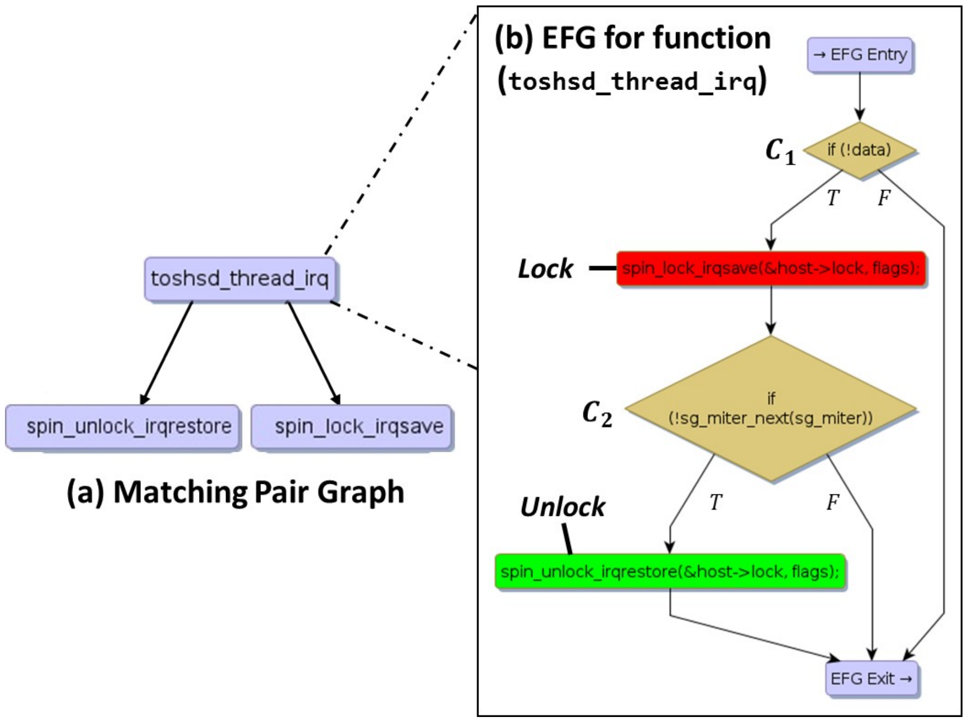


Figure 5.6 A bug discovery using EFGs

is not seen in this EFG. However, there is another `Lock` before the loop in this function. Thus, it is not an error but this unusual situation does require a careful review of the EFG.

5.3.4 An EFG Quirk

This example brings out an EFG quirk due to a peculiar programming pattern. Figures 5.8(a), (b) and (c) respectively show the MPG, EFG, and CFG for the function `drxk_gate_crtl`. The EFG shows that the `Lock` is *not* matched by an `Unlock`. However, the EFG is not conclusive in this case.

The MPG hints that the EFG may not be enough. According to the MPG, `drxk_gate_crtl` calls `Lock` and `Unlock`. However, the `Unlock` is missing in the EFG. The CFG shows that the `Lock` and `Unlock` are on two mutually exclusive paths and that is the reason the `Unlock` does not show up in the EFG for this `Lock`. The mutually exclusive paths are governed by a branch node marked as C. If $C =$ `true`, the `Lock` executes, otherwise the `Unlock` executes.

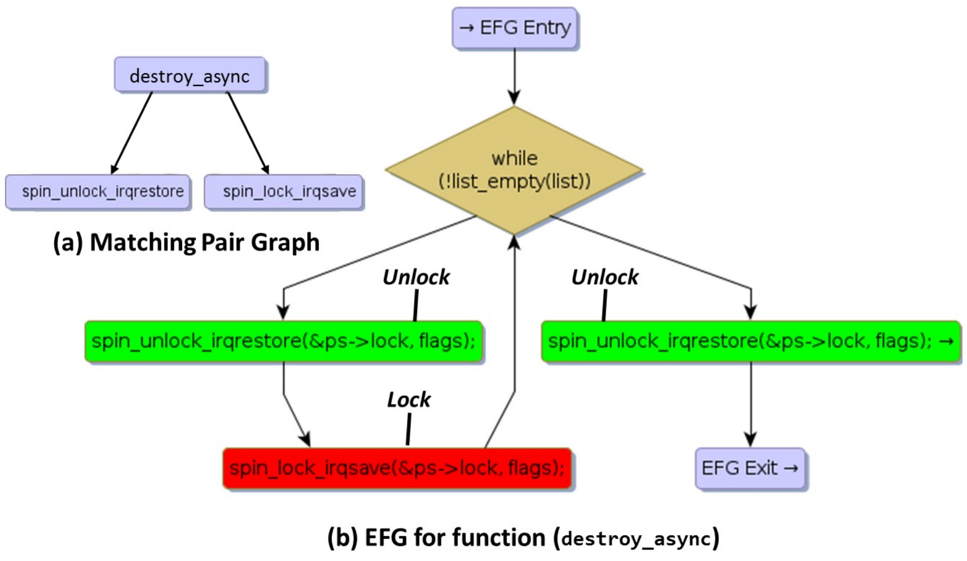

destroy_async
spin_unlock_irqrestore
spin_lock_irqsave
(a) Matching Pair Graph
→ EFG Entry
while (!list_empty(list))
Unlock
Unlock
spin_unlock_irqrestore(&ps->lock, flags);
spin_unlock_irqrestore(&ps->lock, flags); →
Lock
spin_lock_irqsave(&ps->lock, flags);
EFG Exit →
(b) EFG for function (destroy_async)

Figure 5.7 EFG points to a missing lock preceding a loop

The `Lock` and `Unlock` on disjoint paths could pair with each other if `drxk_gate_crtl` is called twice, first with $C =$ `true` and then with $C =$ `false`. This amounts to using `drxk_gate_crtl` first as a lock and then as an unlock. A quick query shows that `drxk_gate_crtl` is not called directly anywhere. Thus, it is either dead code or `drxk_gate_crtl` is called using a function pointer.

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

5.3.5 Comprehension-Driven Verification

Formal verification has been the holy grail of software engineering research [44]. 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 [58]. However, there are two fundamental limitations: (A) a completely automated and accurate analysis encounters NP hard problems [32, 79, 87], and (B) formal verification methods work mostly as automated black boxes with little support for cross-checking [1, 2, 5, 39, 41, 83, 89, 98].

We used the Linux Driver Verification tool (LDV) [5] to verify the pairing of `Lock` and `Unlock`. The above examples are drawn from the 22,843 (34.3)% `Lock` instances that LDV could not verify.

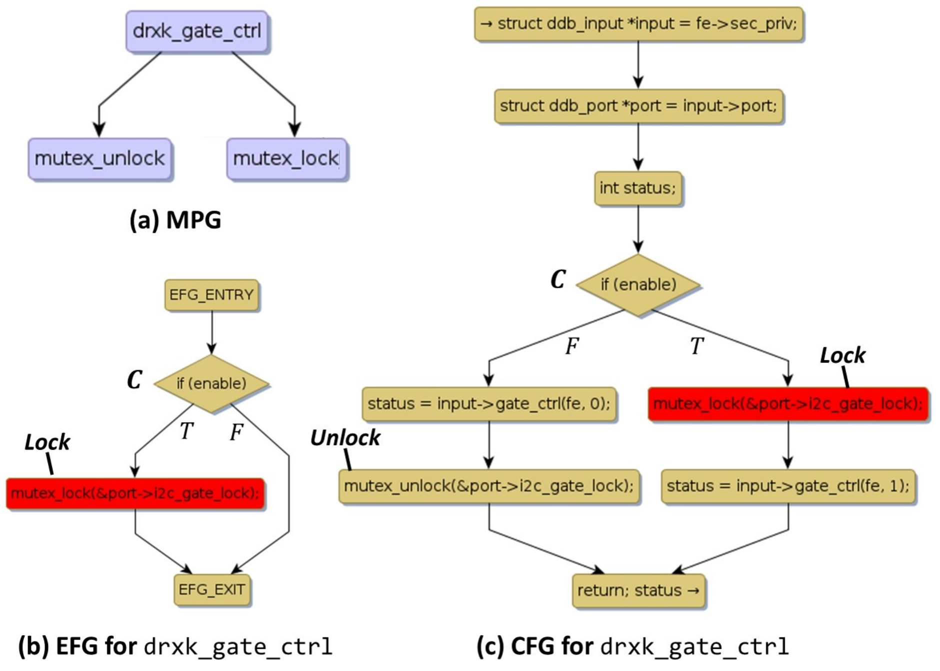

drxk_gate_ctrl
mutex_unlock
mutex_lock
(a) MPG
EFG_ENTRY
C
if (enable)
T
F
Lock
mutex_lock(&port->i2c_gate_lock);
EFG_EXIT
(b) EFG for drxk_gate_ctrl
→ struct ddb_input *input = fe->sec_priv;
struct ddb_port *port = input->port;
int status;
C
if (enable)
F
T
Lock
status = input->gate_ctrl(fe, 0);
mutex_lock(&port->i2c_gate_lock);
Unlock
mutex_unlock(&port->i2c_gate_lock);
status = input->gate_ctrl(fe, 1);
return; status →
(c) CFG for drxk_gate_ctrl

Figure 5.8 An EFG Quirk

Comprehension-driven verification is important for at least two reasons. First, it is the only alternative for cases where fully automated analysis is not possible. Second, it is the only way to cross-check the results of a fully automated analysis which may or may not be correct. Cross-checking is especially important for critical systems where failures of software can be catastrophic.

The EFG is broadly useful for a comprehension-driven solution for the Matching Pair Verification (MPV) problem, which involves verifying the correct pairing of two events on all possible execution paths. Specific examples of such events can be: *allocation* and *deallocation* of memory, *locking* and *unlocking* of mutex, or *sensitive source* and *malicious sink*.

5.4 Application: Find Vulnerability

Detecting sophisticated AC or SC vulnerabilities is like searching for a needle in haystack without knowing what the needle looks like. These sophisticated vulnerabilities are one-of-a-kind. A fully automated approach to detect these vulnerabilities is not possible. Detecting AC and SC vulnerabilities often require domain-specific knowledge [25, 42]. The detection requires exploring software to identify vulnerable code, conceiving plausible attack hypotheses, and gathering evidence by analyzing software to prove or refute each hypothesis. EFG-based program comprehension is important to support such detection.

This section motivates and illustrates the need for the EFG to detect SC vulnerabilities in software. The EFG helps the analyst to focus on the space and time changing events and their governing conditions. If executing such events causes observable space/time differences then it creates the possibility of a SC vulnerability. The governing conditions need to be user-input controlled for an attacker to force the execution of paths with observable space/time differences. Thus, to detect SC vulnerabilities, the analyst must understand the program to answer specific questions: (a) What are the space/time changing events present in the app? (b) What are the governing conditions controlling the execution of these events? (c) Can the governing conditions be controlled by user inputs? We propose a three-phase approach for this application.

This application involves several complex program comprehension problems. We show that the EFG is useful when the analyst has to gather evidence to prove or refute an SC attack scenario.

Phase I:*Automated Exploration.* The objective is to precompute information that serves as the basis for the analyst to begin the investigation. The precomputed information includes the locations of space/time changing loops and the user-input controlled conditions.

Phase II:*Hypothesis Formulation.* After reviewing the precomputed information, the analyst hypothesizes possibilities for SC attacks. By the end of Phase II, the analyst has hypotheses that need to be either validated or refuted.

Phase III:*Validating the Hypotheses.* The objective is: (1) to enable the analyst to gather evidence to refine, refute, or validate each hypothesis formulated in Phase II, and (2) to help the analyst compose the overall *modus operandi* of the attack.

Example. Consider a simple password checking app that compares the passwords stored in a server against strings submitted as passwords. The app `Accepts` if the submitted string matches with a stored password. The app `Rejects` if the match fails.

At the end of Phase I, the analyst observes `Thread.sleep()` as a time-changing event. Based on the taint analysis result, the analyst also knows that there are conditions that are controlled by inputs. In the taint analysis result shown in Figure 5.9, the secret (the passwords stored on the server) is colored red and input (the string submitted as password) is colored blue. When the taints originating from secret and the input come together the color changes to yellow in the taint flow graph.

The taints from the secret and the input come together when the input is compared against the secret passwords. The result of the comparison controls conditions C_1 and C_2 as shown in the taint flow graph (Figure 5.9).

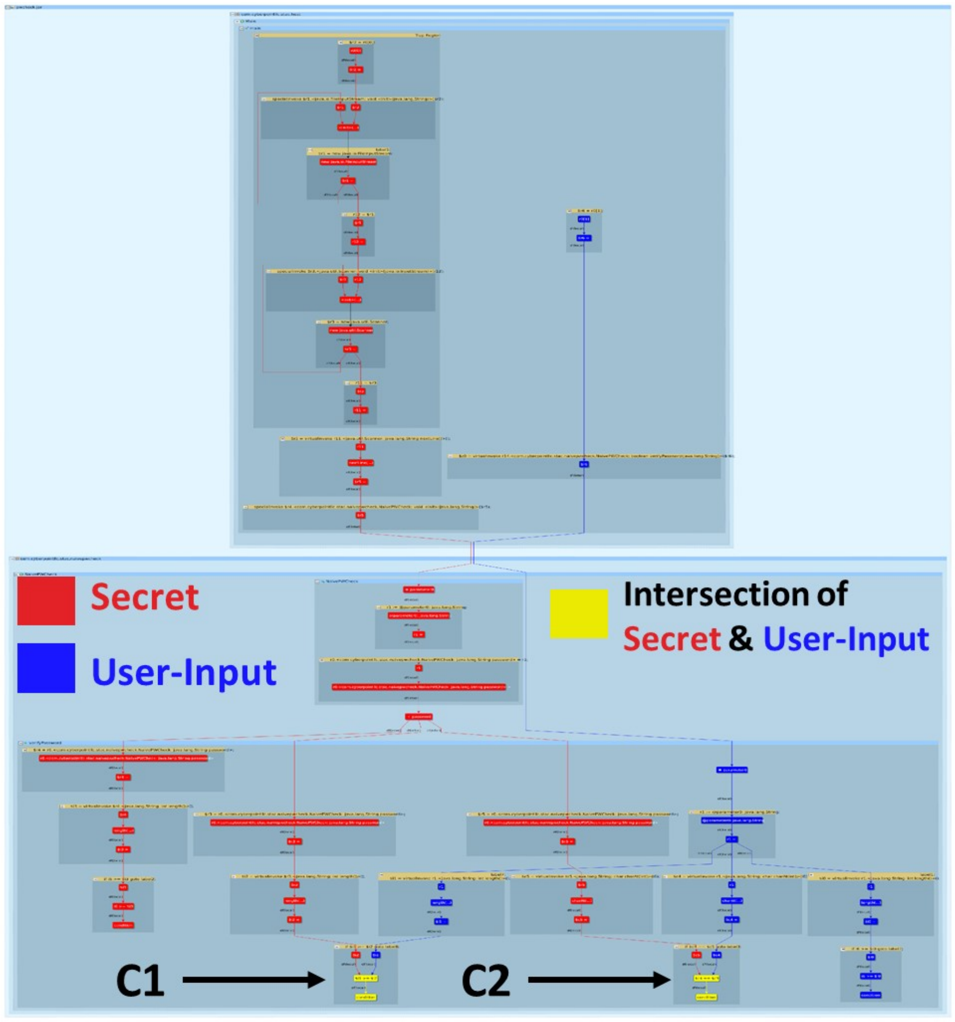


Figure 5.9 Taint flows from the secret and the user-controlled input

In Phase II, the analyst hypothesizes that, depending on the comparison result of the secret passwords with the submitted password, two paths are created either by the condition C_1 or C_2 such that time changing event `Thread.sleep(25)` happens on only one of those two paths. Moreover, one character is compared at a time. Thus, the observed time difference can reveal to the attacker that the character matches. By submitting different strings for password and observing the time differences, the attacker can learn a secret password.

To validate the hypothesis, the analyst gathers evidence in Phase III to answer the following questions:

1. Do the conditions C_1 or C_2 create differential paths with and without the `Thread.sleep(25)` event?

2. Do the conditions depend on character-wise comparison of secret and input?

The EFG is useful for answering these questions. To answer the first question, the analyst can create the EFG for the event `Thread.sleep(25)`. The resulting EFG in Figure 5.10 shows that the condition C_1 actually governs a path with the event `Thread.sleep(25)`. Thus, the execution of `Thread.sleep(25)` is dependent on the comparison of the input and the secret passwords.

To answer the second question, the analyst can create the EFG with events `Thread.sleep(25)`, and the data flow events leading to the condition C_1. If we include all such events, the EFG can become huge and not be an effective program comprehension artifact. Our tool support a gradual expansion of the EFG by adding a few data flow events at a time. The backward data flow can be expanded one step at a time by adding the neighbors one data flow edge away at each step. We expand the EFG with the new data flow events corresponding to newly added neighbors. This EFG after expanding two steps is shown in Figure 5.11. The new events added after expanding two steps are boxed. In this case, the new events were enough to reveal the character-wise comparison and thus no need to expand the EFG further.

5.5 EFG Tool Support

We describe the capabilities we have created to construct EFGs. The capabilities are developed using the Atlas platform [37].

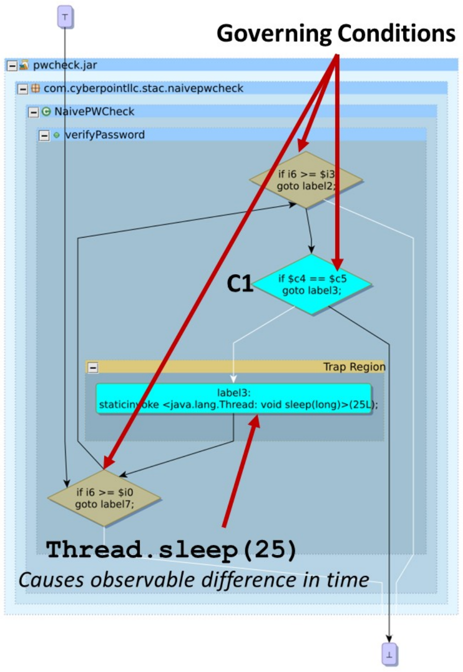

Governing Conditions
pwcheck.jar
com.cyberpointllc.stac.naivepwcheck
NaivePWCheck
verifyPassword
if i6 >= $i3 goto label2;
C1
if $c4 == $c5 goto label3;
Trap Region
label3: staticinvoke <java.lang.Thread: void sleep(long)>(25L);
if i6 >= $i0 goto label7;
Thread.sleep(25)
Causes observable difference in time

Figure 5.10 EFG with respect to `Thread.sleep(25)`

5.5.1 Programmed EFG Construction

The EFG for each `Lock` is created by transforming the CFG for the method containing the `Lock`. The program for constructing EFGs works as follows:

1. **Mark the Events in the CFG**: The first event is the particular `Lock` for which the EFG is being constructed. A taint analysis is used to identify the data flow nodes where either the pointer to the lock object flows to another entity (e.g., assigned to another variable, passed as a parameter to a callee, or returned to the caller). The data flow nodes are marked as events. The `Unlock` operations that use a tainted pointer to the lock object are also marked as events.

2. **CFG to EFG Transformation**: The CFG to EFG transformation algorithm is applied to construct the EFG using the marked events. The algorithm is described in section 5.6.

Note that an `Unlock` operation will not be included in the EFG if it does not use a tainted pointer to the lock object that we started with. For example, the `Unlock` from the example in Section 5.3.4 is not included in the EFG because it is on a disjoint path.

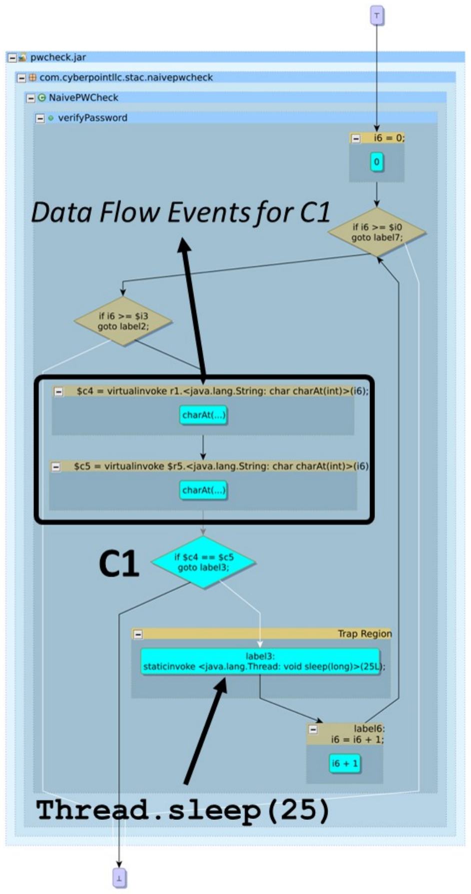


Figure 5.11 EFG with respect to `Thread.sleep(25)` and data flow events

5.5.2 Interactive EFG Construction

We support several interactive ways to construct EFGs. These include:

1. **Create EFG using Smart View**: The interaction involves: (a) select the EFG option from the Smart View menu, (b) create the EFG on-the-fly by clicking on method nodes in a graph. For example, after creating a graph such as the LCG, the analyst views the EFGs for nodes from that graph.

2. **Create EFG by clicking source code**: The EFG can be created on-the-fly by clicking on method names in a source code window.

3. **Create EFG by selecting CFG nodes**: We can select CFG nodes and use them as relevant events to create the EFG.

5.5.3 Gradual EFG Expansion

We have developed a capability for expanding the EFG gradually. We alluded to it while describing the detection of SC vulnerability. The expansion scenario is to gradually add the data flow events and use them to expand the EFG. For detecting SC vulnerability, we want to expand the EFG that includes the condition node C_1. We want to add the events for computations whose results flow into C_1. To do so, we select C_1 and invoke the data flow Smart View of Atlas. The Smart View window shown in Figure 5.12 has a scroll bar for forward or backward data flow and it allows one to specify the number of steps. We get the partial dataflow as per our specification. We select the data flow nodes from the Smart View and add them to the set of event nodes and recreate the EFG.

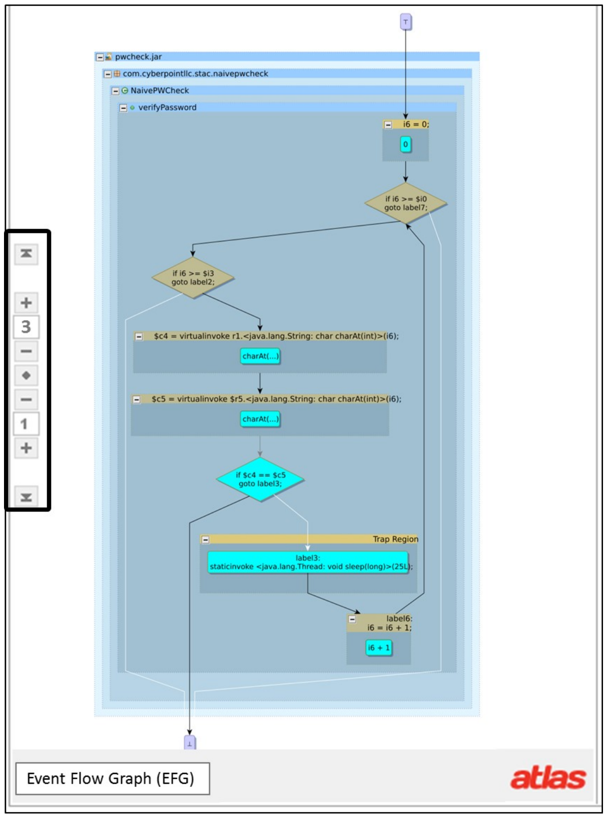

3
1
pwcheck.jar
com.cyberpointllc.stac.naivepwcheck
NaivePWCheck
verifyPassword
i6 = 0;
0
if i6 >= $i0 goto label7;
if i6 >= $i3 goto label2;
$c4 = virtualinvoke r1.<java.lang.String: char charAt(int)>(i6);
charAt(...)
$c5 = virtualinvoke $r5.<java.lang.String: char charAt(int)>(i6);
charAt(...)
if $c4 == $c5 goto label3;
Trap Region
label3: staticinvoke <java.lang.Thread: void sleep(long)>(25L);
label6: i6 = i6 + 1;
i6 + 1
Event Flow Graph (EFG)
atlas

Figure 5.12 Gradual EFG Expansion Interface

5.6 Event Flow Graph

We present a novel algorithm to compute EFG. While the number of paths in a CFG can be exponential, the algorithm is linear with respect to number of nodes and edges in the CFG. It is based on Tarjan's algorithm to compute strongly-connected components of a directed graph [86]. The algorithm computes the EFG by performing a set of graph transformation on a CFG.

5.6.1 Step 1: Marking Event Nodes

The first step is to mark the event nodes in the given CFG. For example, for `Lock` and `Unlock` pairing the marked events start with the `Lock`, include the data-flow events in which the locked object p is either aliased or escapes to another function as a parameter, a return value, or a global variable, and the subsequent `Unlock` events.

In our CFG representation, the CFG has unique entry and exit nodes and each CFG node corresponds to a program statement.

5.6.2 Step 2: T-Irreducible Graph

Transform G_{CFG}, a CFG with the marked event nodes, to the *T-irreducible* graph $G_{\text{T-irr}}$ by applying the following basic transformations $T = \{T_1, T_2, T_3\}$ until the resultant graph cannot be further reduced by applying basic transformations.

T_1: Elimination of Non-branching and Non-event Nodes

Let n be a *non-event* node with a single successor m. The T_1 transformation is the consumption of node n by m. Induced edges are introduced so that the predecessors of node n become predecessors of node m. (Figure 5.13(a))

The T_1 transformation eliminates every node from CFG that is neither a branch node nor an event node.

T_2: Elimination of Self-Loop Edges

Let n be a *non-event* node that has a self-loop edge (n, n). The T_2 transformation removes that edge. (Figure 5.13(b))

The intuition behind T_2 transformation is: when a loop block does not contain event nodes, execution of the loop is immaterial. Therefore, T_2 removes the self-loop edges.

T_3: Elimination of Irrelevant Branch Nodes

Let n be a *non-event* node that has two or more outgoing edges, all pointing to the same successor m of n. Then the T_3 transformation is the consumption of node n by m and the predecessors of node n become predecessors of node m. (Figure 5.13(c))

The intuition behind the T_3 transformation is: Imagine the case where a branch node n has only non-event nodes on its branches, and all those branches ultimately merge at node m. If the non-event nodes on those branches are eliminated by the T_1 transformation, all branches will point to node m. Since the branching at n is *irrelevant* at this point, the branch node n is eliminated.

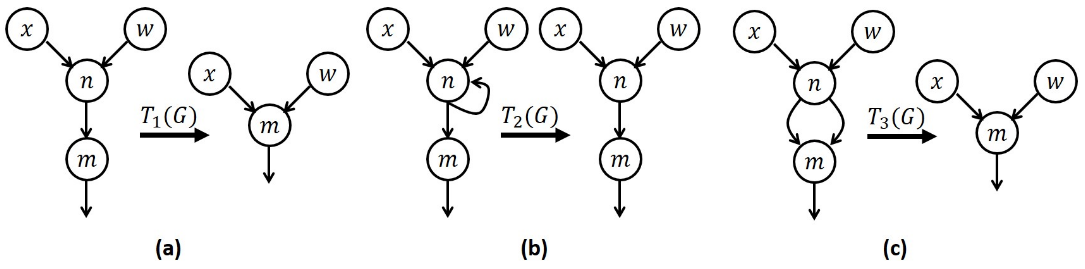


Figure 5.13 T-irreducible graph transformations: (a)T_1, (b)T_2, (c)T_3

Definition 1 *G_{CG} is the condensation graph of a directed graph G if each strongly-connected component (SCC) of G contracts to a single node in G_{CG} and the edges of G_{CG} are induced by edges in G.*

5.6.3 Step 3: Non-Event Condensation Graph

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

5.6.4 Step 4: Event Condensation Graph

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

5.6.5 Step 5: Condensed EFG

Transform G_{ECG} into a T*-irreducible* graph G_{cEFG} by applying the set of basic transformation $T = \{T_1, T_2, T_3\}$ as in Step (2). The resultant graph G_{cEFG} after this step is the *condensed EFG*.

5.6.6 Step 6: Final EFG

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

Figures 5.14(a-f) illustrate our EFG construction algorithm. The event nodes are highlighted.

5.6.7 Algorithm Complexity

The algorithmic complexity of constructing the EFG is $O(|V| + |E|)$ where $|V|$ and $|E|$ are the respective numbers of nodes and edges in the CFG. For detecting the SCCs in Step (3), we use an algorithm by Tarjan [86] to compute strongly-connected components of a directed graph. The run-time of this algorithm is also $O(|V| + |E|)$, yielding a linear run-time complexity of $O(|V| + |E|)$ for our CFG pruning (CFG to EFG transformation). The algorithm does not depend on the number of paths.

5.6.8 Observations

Initially, we believed that Step 2, which produces the T-irreducible graph, would produce the EFG. The produced graph is minimal in the sense that it cannot be further reduced by the

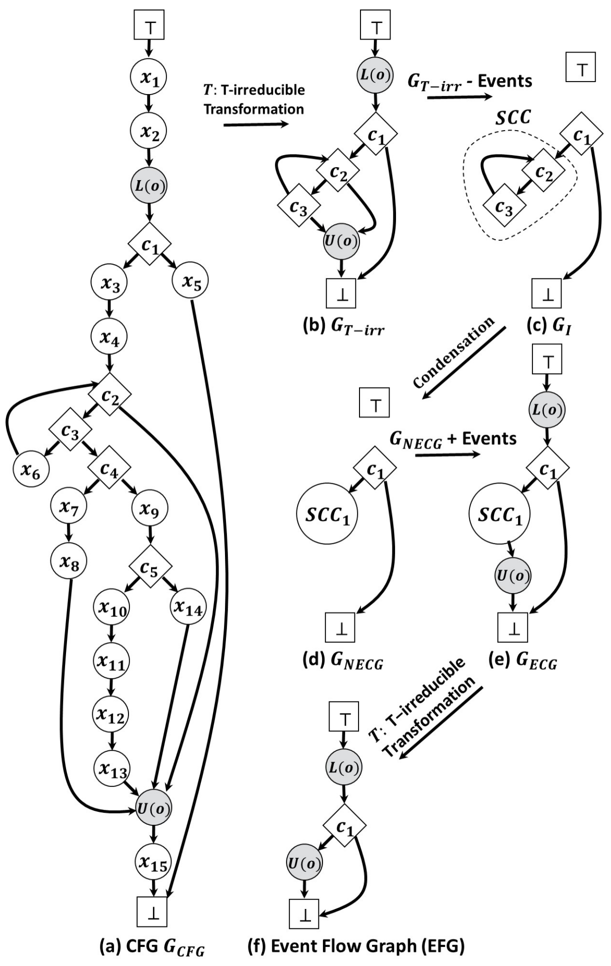


Figure 5.14 CFG to EFG Transformation Illustration

three basic transformations. Then, we encountered complex examples from the Linux kernel with huge T-irreducible graphs. These graphs include many irrelevant branch nodes that cannot be removed by the three basic transformations. We realized that the difficulty arises due to strongly connected components in the CFG. This realization led to the subsequent steps to tackle strongly connected components.

Later, we are able to formalize the notion of *irrelevant branch nodes* and prove that Steps 3 through 6 indeed remove all irrelevant branch nodes and thus one can claim a stronger minimality for EFG than the minimality of being T-irreducible. Besides the theoretical elegance of the mathematical framework, the big T-irreducible EFGs we had encountered earlier got significantly reduced. The mathematical framework and the minimality proof are presented in the next section.

5.6.9 EFG Minimality

Definition 2 *A **Control Flow Graph** (CFG) of a program is defined as* $G = (V, E, \top, \bot)$, *where* G *is a directed graph with a set of nodes* V *representing the program statements and a set of edges* E *representing the control flow between statements.* $\top$ *and* $\bot$ *denote the respective unique entry and exit nodes of the graph.*

Definition 3 *Successors of a node* u *in a directed graph* G*, denoted by* $\mathrm{suc}(u)$*, consist of the set of nodes* $v \neq u$ *such that* $\exists$ *an edge* (u, v)*.*

Definition 4 *Successors of a subgraph* S *in a directed graph* G*, denoted by* $\mathrm{suc}(S)$*, consist of the set of nodes* $v \notin S$ *such that* $v = \mathrm{suc}(u)$ *for* $u \in S$*.*

Definition 5 *For a branch node* c*, a **branch edge** is an out-coming edge of* c*.*

Definition 6 *A branch node* c *is an **irrelevant branch node** if the following conditions are satisfied: (1)* c *is a non-event node, (2) there exists a subgraph* S *containing* c *and all branch edges of* c*, (3)* S *has no event nodes, and (4)* S *has a unique successor, i.e.,* $|\mathrm{suc}(S)| = 1$*.*

Figure 5.15 shows an example of irrelevant branch node c.

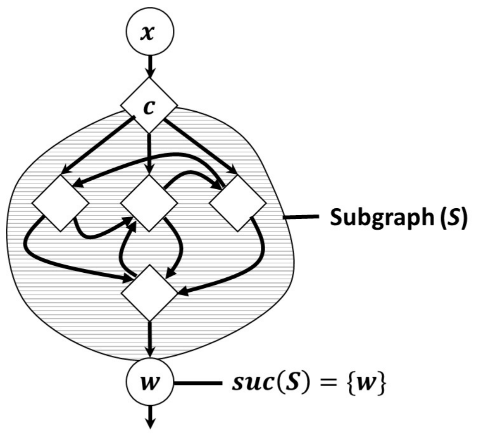


Figure 5.15 An example of irrelevant branch node c

Definition 7 *The **Event-Flow Graph** (EFG)* G_{EFG} *of a CFG* G *is the node-induced subgraph of* G *consisting of the event nodes, the relevant branch nodes, and the entry (*$\top$*) and exit (*$\bot$*) nodes.*

The above definition of EFG captures a notion of minimality by specifying that the EFG eliminates *all* irrelevant branch nodes - a notion precisely defined by the above mathematical framework. Is the EFG produced by the six steps detailed in Section 5.6 the same as the EFG defined by the mathematical framework? The following proves that the answer is yes. Thus, the EFG produced by the transformation in Section 5.6 satisfies the minimality constraint described above.

Definition 8 *The boundary of a subgraph S in a directed graph G, denoted by* $\mathrm{boundary}(S)$*, is the set of nodes $u \in S$ such that* $\mathrm{suc}(u) \in \mathrm{suc}(S)$.

Theorem 1 *Let G be a T-*irreducible *and acyclic graph. Then for any subgraph S containing non-event nodes of G:* $|\mathrm{suc}(S)| \geq 2$.

Theorem 1 Proof. If a non-event node $u \in G$ has only one successor then it is eliminated by transformation T_1. Thus, since G is *T-irreducible*, $|suc(u)| \geq 2$ for all non-event nodes $u \in G$. Also, by assumption, G is an acyclic graph. Using these two facts, we will show that every subgraph S has a node with at least two successors outside S and thus $|suc(S)| \geq 2$.

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

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

Corollary 1 Proof. Note that the condensed EFG G_{cEFG} is the graph resulting from Step (5) of the EFG construction algorithm. By construction, the condensed graph G_{cEFG} is a *T-irreducible* graph. Also, by construction G_{cEFG} is an acyclic graph. By applying the above theorem to G_{cEFG} we get the proof of the corollary.

Corollary 2 *The graph produced by Step (6) does not contain any irrelevant branch nodes.*

Corollary 2 Proof. By construction, the graph $G_{\text{T-irr}}$ resulting after Step (2), consists of only event nodes, relevant branch nodes, and the irrelevant branch nodes retained by Step (2). We will now argue that all the irrelevant branch nodes will be eliminated when G_{cEFG} is constructed in Step (5). According to the definition of irrelevant branch nodes (Definition 6), a node c is irrelevant if there is a subgraph S that contains c, all its branch edges, S has no event nodes, and $|suc(S)| = 1$. It follows from this definition and from the corollary 1 that G_{cEFG} does not contain any irrelevant branch nodes. Thus, the final graph produced by Step (6) also does not contain any irrelevant branch nodes, because it consists of the nodes in G_{cEFG} and all the event nodes.

5.7 An Assessment of EFGs

EFGs are meant to reduce the cognitive burden on the human analyst. Their utility depends on the reduction they afford relative to the original CFG. We summarize the reductions observed from an empirical study.

The events of interest for creating an EFG are `Lock`, `Unlock`, and the data flow events in which the pointer p to the lock object is either aliased or escapes to another function as a parameter, a return value, or a global variable. We created the CFG and the EFG for each function that contains events of interest.

5.7.1 An Empirical Study

We developed a pairing analysis tool using the Atlas platform [37]. The tool was used to create the CFG, the EFG, and the MPG for each `Lock` instance from 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 three Linux versions together include 37 MLOC and 66,609 `Lock` instances. Table 5.1 shows the `mutex` and `spin` lock and unlock calls and how they are paired.

We used the Linux Driver Verification tool (LDV) [5] to verify the pairing of `Lock` and `Unlock`. It could verify 43,766 (65.7)% `Lock` instances and it took 173 hours to do so. The instances in

Table 5.1 `mutex/spin Locks/Unlocks`

Calls	`mutex` synchronization	`spin` synchronization

Lock	`mutex_lock`
	`mutex_trylock`
	`mutex_lock_interruptible`
	`mutex_lock_killable`
	`atomic_dec_and_mutex_lock`
		`spin_lock`
		`spin_trylock`
		`spin_lock_irqsave`
		`spin_lock_irq`
		`spin_lock_bh`

Unlock	`mutex_unlock`	`spin_unlock`
		`spin_unlock_irqsave`
		`spin_unlock_irq`
		`spin_unlock_bh`

Section 5.3 are among the instances that LDV could not verify. LDV is the current top-rated tool in the software verification competition (SVCOMP) [18–20] in the Linux device drivers verification category. The LDV developers generously provide us with the results on the same versions of the Linux kernel and the same build configurations we have used.

5.7.1.1 Empirical Study Website

The CFG, EFG, and the MPG graphs for each of the 66,609 `Lock` instances are available on a website [6]. Each instance includes source correspondence. The data brings out different types of complexities for pairing analysis. The data can be used for understanding the real-world challenges for automated analysis. For example, it would be valuable to use the data to analyze the 22,843 instances that LDV cannot verify and develop a test suite which would capture the Linux verification challenges.

5.7.2 Quantitative Assessment of EFGs

We summarize the reduction in nodes and edges in going from CFGs to EFGs. The reduction is due to the removal of non-event nodes and irrelevant branch nodes. The reported results are for the Linux kernel (v3.17-rc1, v3.18-rc1, v3.19-rc1).

Table 5.2 shows the distribution of nodes, edges, and branch nodes for both the CFGs and EFGs for all the relevant functions. Compared to 30,914 CFGs, only 115 EFGs have more than (30) nodes, which is a reduction of 99%. Compared to 35,145 CFGs, only 879 EFGs have more than (30) edges, which is a reduction of 97%. Compared to 17,120 CFGs, only 1,810 EFGs have more than (10) branch nodes, which is a reduction of 89%. EFGs simplify path feasibility checks by reducing the number of branch nodes. Compared to 8,644 CFGs, 30,999 EFGs have no branch nodes, which is a 259% increase of cases where EFG eliminates the need for path feasibility checks.

Table 5.2 Linux kernel CFG and EFG stats

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

The reductions from CFG to EFG is particularly important for a CFG with a large number of branch nodes. Table 5.3 lists the reductions for the ten functions with the largest number of branch nodes. For example, for function `dst_ca_ioctl` the reductions from CFG to EFG are: from 349 to 2 nodes, from 518 edges to only one edge, and from 163 to no branch nodes.

Table 5.3 A comparison of CFG vs. EFG

Function Name	Nodes		Edges		Branch Nodes	
	CFG	EFG	CFG	EFG	CFG	EFG
`client_common_fill_super`	1,101	15	1,179	28	249	13
`kiblnd_create_conn`	731	18	925	34	197	15
`CopyBufferToControlPacket`	392	20	559	39	180	18
`kiblnd_cm_callback`	662	38	831	56	170	15
`kiblnd_passive_connect`	622	22	784	44	164	20
`dst_ca_ioctl`	349	2	518	1	163	0
`qib_make_ud_req`	621	10	821	15	156	5
`cfs_cpt_table_al`	522	7	672	13	153	6
`private_ioctl`	569	16	732	24	148	8
`vCommandTimer`	490	47	623	75	143	28

5.8 Related Work

Event flow graphs are inspired by the work done by Neginhal *et al.* [65] where they introduced the notion of *event view*. They require the user to determine irrelevant nodes/edges to be removed. They do not have an algorithmic notion to make CFG compact. We have a mathematical definition of EFG and a linear-time algorithm to compute the EFG.

The idiom of events has been studied in the context of distributed systems and performance analysis [51, 78]. While we share the common goals of simplicity, accuracy, and scalability of program analysis, the context, the concept, and the applicability of EFG is quite different from the above cited research.

CFG pruning techniques have been proposed in [29, 77] to overcome the computational complexity of exploring all paths. Other pruning techniques have been introduced by Choi *et al.* [30] and Ramalingam [76]. Both proposed an equivalence relation to optimize data flow graphs. The EFG advances these techniques, introduces a new event-based equivalence relation, and guarantees the optimality of reduction from CFG to EFG.

The Binary Decision Diagram (BDD) [13] has been used in different contexts of program analysis as a way to reduce the explosion of state space [16,60,84,93]. The Binary Decision Tree (BDT) to BDD reduction has been also used for path-sensitive analysis [98, 102]. Unlike BDT to BDD reduction, the EFG transformation achieves further reduction and does not require the input CFG to be acyclic. The EFG transformation deals with cyclic graphs by incorporating a *linear-time* algorithm by Tarjan [86] to compute strongly-connected components of a directed graph. EFGs can be used in place of BDT to BDD reduction.

5.9 Conclusion

This chapter is about the EFG, an application-specific program comprehension model. The chapter provides a rigorous formulation of a compact derivative of the CFG as a fundamental concept to understand and reason about execution behaviors relevant for a particular application. The goal is to reduce the cognitive burden by simplifying the CFG while retaining application-specific execution behaviors.

The chapter presents `Lock` and `Unlock` pairing and side-channel vulnerability detection as two diverse applications of the EFG as a powerful program comprehension tool. The pairing analysis application can be readily used in other contexts that involve pairing of two events, for example, the memory leak verification problem.

The tool support for EFGs is developed using the query language and visualization capabilities of the Atlas platform [37]. The tool support is interactive and it integrates inter- and intra-procedural program comprehension. For example, it provides LCG to view the loops across methods and then by clicking on a method one can view the loop-based EFG for that method.

The utility of the EFG depends on how compact it is in comparison to the CFG. The chapter presents quantitative data from an empirical study on three recent versions of the Linux kernel. The data shows significant reductions from CFGs to EFGs. The CFGs and EFGs, for each of the $66,609$ `Lock` instances for three recent versions (3.17-rc1, 3.18-rc1, and 3.19-rc1) of the Linux kernel, are posted on a website [6]. It includes $22,843$(34.3%) `Lock` instances that could not be verified by LDV [5], the currently top-ranked Linux verification tool. This data makes it easy to cross-check results produced by an automated pairing tool. It also brings out the spectrum of difficulties for automated verification.

CHAPTER 6. $\mathcal{L}$-SAP: EVIDENCE-ENABLED LINUX VERIFICATION FOR LOCK/UNLOCK PAIRING ANALYSIS

6.1 Introduction

Synchronization problems can be catastrophic - a business-transaction server can crash resulting in a big financial loss, or a safety-critical control system can halt causing loss of lives. With multi-threading and event-driven processing, it is challenging to ensure resilience of software systems to synchronization problems. These problems can elude dynamic analyses and regression testing because their occurrence often depends on intricate sequences of low-probability events [41]. Performing multiple runs of a program to examine all possible behaviors is prohibitively expensive and time-consuming. Thus, automated static analyses are crucial to complement testing and dynamic analyses.

An accurate, scalable, and completely automated static analysis has been the holy grail of research. Over the years, many static analysis approaches have been proposed to discover synchronization problems in C programs [29, 39, 41, 68, 73, 81, 89]. These state-of-the-art approaches are based on older versions of the Linux kernel ($<$ 4 MLOC) or on medium-sized programs that are orders of magnitude smaller. These approaches have led to new advances in data and control flow analyses, and new heuristics to apply techniques such as Binary Decision Diagrams (BDDs) to analyze large software [58]. These advances have pushed further the boundaries of scalability and accuracy. However, there is a fundamental limitation: a general-purpose accurate lock/unlock pairing analysis is not intrinsically scalable as it involves NP hard problems [32, 79, 87].

Our approach is to specifically address commonly occurring roadblocks for scalability and accuracy to arrive at a solution that works well in practice. The Linux kernel code base has unique combinations of specific characteristics that attract researchers and practitioners to challenge their tools [22]. This motivated us to use the Linux kernel code base as a good target system to test our approach. Algorithmic innovations presented in this paper are inspired by the following guiding research question: what scalability and accuracy roadblocks are typical in practical applications, which can be addressed by customized program analysis that achieves accuracy and scalability without being too restrictive?

Consider the lock/unlock pairing analysis: a lock event L is paired with an unlock event U iff U can release the lock acquired by L. The existence of unpaired locks on feasible execution paths results in synchronization problems. This pairing of a lock with its corresponding unlocks on all possible feasible execution paths requires the following: (a) a *data flow analysis* to *map* each lock to corresponding unlocks that reference the same *lock object*, (b) a *control flow analysis* to *pair* each lock with corresponding unlocks. This is achieved by *identifying* all possible intra- and inter-procedural execution paths that have a lock event ensued by an unlock event, and (c) a *feasibility analysis* to check the feasibility of an execution path on which a lock is not ensued by an unlock. A general-purpose, completely automated, and accurate analysis is intractable for each of the above three requirements.

We present a novel *scalable* and *accurate* static lock/unlock pairing analysis that is explicitly designed to handle the analysis roadblocks we observed in the Linux kernel. We design a *type-based* analysis and leverage the MPG (Section 6.2.2) to satisfy the analysis requirement (a) to map each lock with a set of corresponding unlocks to perform object-sensitive analysis. To efficiently meet the analysis requirement (b), we use event flow graphs (Section 6.2.3) to minimize the set of paths that must be examined for path-sensitive accurate analysis. We also design *compact function summaries* for context-sensitive scalable inter-procedural analysis. For requirement (c), among this minimized set of paths, we separate the paths on which a lock is not ensued by an unlock. By construction, this is a necessary and sufficient set of paths that should be checked for feasibility. Thus, the need for feasibility analysis is also minimized by applying it only in the cases where it is needed.

We developed $\mathcal{L}$-SAP, a tool that uses our novel lock/unlock pairing analysis. It analyzes each lock and it produces three categories of results: ($\mathcal{C}1$) an automatically verified instance with correct lock/unlock pairing, ($\mathcal{C}2$) an automatically verified instance with a unpaired lock i.e. a feasible path with a lock not ensued by unlock, and ($\mathcal{C}3$) an allocation instance where the automated verification is inconclusive. $\mathcal{L}$-SAP produces for each lock instance the following evidence: the *matching pair graph* (MPG) (Section 6.2.2) to assist with the inter-procedural reasoning by identifying relevant functions and their interactions, and the event flow graph (EFG) (Section 6.2.3) to assist with intra-procedural reasoning for each relevant function. We have applied $\mathcal{L}$-SAP to recent three versions (3.17-rc1, 3.18-rc1, and 3.19-rc1) of the Linux kernel totalling $>$ 37 MLOC. $\mathcal{L}$-SAP is fast and scalable, and it is able to accurately pair $66,151$ (99.3% of the total) locks in 3 hours with no false negatives. Our analysis discovered 7 synchronization bugs that were reported to the Linux community and accepted by them. Compared with the Linux Driver Verification (LDV) tool [5], a top-rated verification framework in the competition on software verification (SV-COMP) [18–20] in the category of Linux device drivers verification, $\mathcal{L}$-SAP reduces by 49$\times$ the number of statically unpaired locks and by 59$\times$ the analysis time.

To the best of our knowledge, we are the first to perform lock/unlock pairing analysis of the recent versions of the Linux kernel. Our evaluation results show that $\mathcal{L}$-SAP provides a scalable, practical, and accurate lock/unlock pairing analysis for the Linux kernel. $\mathcal{L}$-SAP is publicly available [4] so that other researchers can reproduce our results. $\mathcal{L}$-SAP is developed using Atlas [37], which is a platform to develop program comprehension, analysis, and validation tools.

To summarize, this chapter reports the following research contributions:

1. A novel scalable and accurate lock/unlock pairing analysis algorithm and its implementation in the tool $\mathcal{L}$-SAP (Section 6.2.4).

2. An exhaustive pairing analysis that provides for every lock a verdict (i.e., paired/unpaired) accompanied with evidence for manual validation.

3. Efficient object- and context-sensitive inter-procedural analysis by incorporating: type-based analysis to map a lock to appropriate unlocks (Section 6.2.1), and compact function summaries (Section 6.2.4.1).

4. A comprehensive evaluation of $\mathcal{L}$-SAP on three recent versions of the Linux kernel done automatically in 3 hours. The experimental results show that $\mathcal{L}$-SAP correctly pairs 99.3% of the locks and identifies 7 synchronization bugs (Section 6.4).

The remainder of the chapter is organized as follows. We first describe our static lock/unlock pairing analysis in Section 6.2. Next, Section 6.3 describes the tool support for the pairing analysis as well as the produced evidence. Section 6.4 presents the experimental results on three recent versions of the Linux kernel. Finally, we conclude in Section 6.5.

6.2 $\mathcal{L}$-SAP Approach

The lock/unlock pairing analysis used in $\mathcal{L}$-SAP is explicitly designed to pair lock/unlock function calls for `mutex` and `spin` synchronization mechanisms, both widely used in the Linux kernel. Table 6.1 shows the specific lock/unlock function calls for `mutex` and `spin` synchronization mechanisms in the Linux kernel.

Table 6.1 Lock/Unlock function calls for `mutex`/`spin` synchronization mechanisms in Linux kernel

Calls	`mutex` synchronization	`spin` synchronization

Lock Calls	`mutex_lock`
	`mutex_trylock`
	`mutex_lock_interruptible`
	`mutex_lock_killable`
	`atomic_dec_and_mutex_lock`
		`spin_lock`
		`spin_trylock`
		`spin_lock_irqsave`
		`spin_lock_irq`
		`spin_lock_bh`

Unlock Calls	`mutex_unlock`	`spin_unlock`
		`spin_unlock_irqsave`
		`spin_unlock_irq`
		`spin_unlock_bh`

Figure 6.1 shows an overview of our lock/unlock pairing analysis. The pairing analysis has five steps. In the first step, $\mathcal{L}$-SAP *maps* each lock to the set of corresponding unlocks. The mapping is performed via *type-based* analysis, which introduces the notion of *signature*. A lock $L(o)$ is mapped to unlock $U(m)$ iff the objects o and m have the same signature. In the second step, for each signature o, $\mathcal{L}$-SAP creates the *matching pair graph* (MPG_o) as defined in [47] (Section 4.1). The nodes in this graph provide the minimum set of functions for inter-

procedural pairing analysis of locks and unlocks with signature o. The directed edges in MPG_o represent call relationships. In the third step, for each function in MPG_o, $\mathcal{L}$-SAP prunes the CFG to produce a compact CFG named the *event flow graph* (EFG) (Chapter 5. The EFG enables efficient path-sensitive lock/unlock pairing by defining an equivalence relation on the CFG paths such that it is sufficient to examine only one path from each equivalence class. These first three steps set the stage for an efficient pairing algorithm. In the fourth step, $\mathcal{L}$-SAP iterates over the set of signatures to apply the pairing algorithm to each MPG_o and pair the locks and unlocks with signature o. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. In the fifth step, $\mathcal{L}$-SAP calculates Boolean expressions for the conditions governing each potential-error path on which a lock with signature o is either: (i) not paired with an unlock with signature o, or (ii) paired with a lock of signature o (a potential deadlock). Then, using BDDs [92], $\mathcal{L}$-SAP examines whether the potential-error paths are feasible. These five steps are described below in detail.

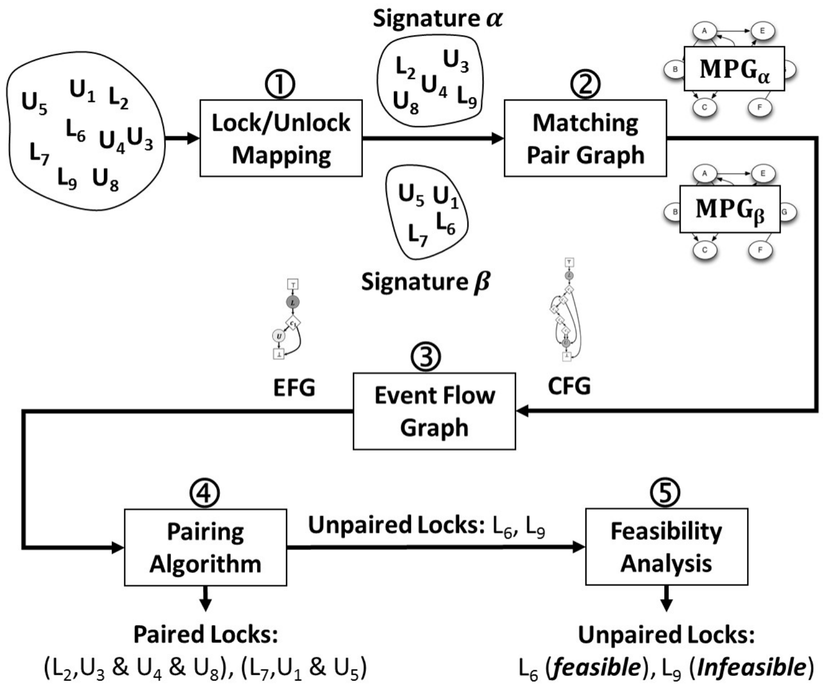


Figure 6.1 An Overview of $\mathcal{L}$-SAP Pairing Analysis

6.2.1 Step 1: Lock/Unlock Mapping

The lock/unlock mapping is performed through type-based analysis via the notion of a *signature* such that: a lock $L(o)$ is mapped to unlock $U(m)$ iff the object o and m have the same signature. The signature-based analysis works as follows:

Consider the pointer $\mathcal{P}$ given by the expression: $(a_n \cdots a_3(\texttt{.}||\texttt{->})a_2(\texttt{.}||\texttt{->})a_1)$. In this expression, $\mathcal{P}$ is being accessed through a chain of member-selection C operators (`.` and/or `->`). We define the *hierarchal type* for $\mathcal{P}$ as the tuple $(T_{a_n}, \cdots, T_{a_3}, T_{a_2}, T_{a_1})$, where T_{a_i} denotes the type associated with the member a_i. For example, the hierarchal type for pointer (`x->y->z`) is given by the tuple (X, Y, Z) where $T_x = X, T_y = Y$, and $T_z = Z$. For a directly referenced pointer, the hierarchal type is the same as its type. For example, the hierarchal type for pointer (`k`) is T_k.

We use the term *object signature* (S_o) to denote: (1) the object (variable) *name* in case o is a global variable that is directly referenced, and (2) the *hierarchal type* for the object (pointer) o otherwise. Based on these definitions, $\mathcal{L}$-SAP maps each lock to a set of corresponding unlocks as follows:

1. Mine all callsites to lock and unlock based on Table 6.1.

2. A lock $L(o)$ is mapped to unlock $U(m)$ iff $S_o = S_m$.

Let us illustrate this through an example: let us say that function A has the lock L(`x->y->z`) and function B calls unlock U(`l->m->n`). Then, $\mathcal{L}$-SAP will map the lock L(`x->y->z`) to the unlock U(`l->m->n`) iff both signatures ($S_{\texttt{x->y->z}}$ and $S_{\texttt{l->m->n}}$) are the same. In other words, their hierarchal types are the same where $(T_x, T_y, T_z) = (T_l, T_m, T_n)$. In case of a global lock object, the signature is the global variable's name.

6.2.2 Step 2: Matching Pair Graph

For each signature o, $\mathcal{L}$-SAP creates a matching pair graph (MPG_o) as defined in [47] (Section 4.1). The nodes in the graph provide the minimum set of functions for inter-procedural pairing analysis of locks and unlocks with signature o. The directed edges in MPG_o represent call relationships. MPG_o captures the four inter-procedural cases identified in Section 2.3. For the fourth case resulting from asynchronous processing, the MPG_o would have two disconnected nodes, i.e., the corresponding functions are not connected by a call sequence because they are invoked asynchronously.

We take a signature o and the associated locks/unlocks as inputs, then compute the matching pair graph MPG_o by running a set of Atlas queries against the system's call graph. This query-based approach is faster compared to recursive traversing of the system's call graph.

Currently, we do not resolve function pointers. This is a source of inaccuracy. Consider the example of lock L_A in function A and unlock U_B in function B where L_A is mapped to U_B as both have the same signature o. Now, assume function C calls A and B via function pointers. Because, we do not resolve function pointers, we will miss function C in MPG_o; it will contain only the functions A and B. The good thing is, the matching pair graph provides the human analyst with hints about the pairing possibility between L_A and U_B. Results in Section 6.4 show only a small percentage of unpaired locks due to presence of function pointers.

6.2.3 Step 3: Event Flow Graph

In this step, we prune the CFG to form equivalence classes of CFG paths. This is done by constructing for each CFG the corresponding EFG using the linear-time algorithm as described in Chapter 5. The initial step in our CFG pruning algorithm is marking the event nodes in the given CFG that are relevant to lock/unlock pairing analysis. In our CFG representation, the CFG has unique entry and exit nodes and a CFG node corresponds to a single program statement. Given the matching pair graph MPG_o for signature o and the CFG G_{CFG} for function $f \in MPG_o$, the events of interest for lock/unlock pairing are as follows: (1) the CFG nodes that correspond to lock/unlock function calls that are associated with signature o, and (2) the CFG nodes that correspond to call-sites for functions in MPG_o. Next, $\mathcal{L}$-SAP takes as input: the G_{CFG} and its marked event nodes and produces the corresponding EFG (G_{EFG}) using the EFG transformations defined in Section 5.6.

6.2.4 Step 4: Pairing Algorithm

$\mathcal{L}$-SAP iterates over the set of signatures to apply the pairing algorithm to each MPG_o to pair the locks and unlocks with signature o. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. Note that, if a function appears in two matching pair graphs with corresponding signatures o_1 and o_2, then the function would have two contexts as well as two summaries. $\mathcal{L}$-SAP visits the matching pair graph in a bottom-up manner while: (1) computing compact function summaries for each visited function, plugging in the summaries of the callees at call-sites while analyzing the callers, and (2) keeping track of the lock/unlock pairs, unpaired locks, and deadlocks.

6.2.4.1 Compact Function Summaries

For each function, the function summary is computed by traversing its EFG in a depth-first manner while keeping track of all entry/exit locks/unlocks on all EFG paths. Let us illustrate our approach to computing compact function summaries. Figures 6.2(a) and (b) show the EFG for functions f and g. In this example, f calls g at statement: `Call g;`. The function summary for g should retain the information relevant for analyzing f. The pairing analysis is concerned with what follows a given lock: (i) a lock followed by unlock implies pairing, (ii) a lock followed by another lock implies a deadlock, and (iii) a lock not followed by lock or unlock implies an unpaired lock.

The function summary consists of two sub-summaries: *entry* and *exit* summaries.

The entry summary for g summarizes: (i) the possible unlock(s) in g that can be paired with $L_P(o)$, (ii) the lock(s) in g that cause deadlocks with $L_P(o)$, and (iii) the possibility of not pairing $L_P(o)$ with lock/unlock in g. Case (iii) is possible if there exists a path in g that does not have lock/unlock events. The entry summary for g - denoted by `entry_summary` in Figure 6.2(d)- includes: the entry locks ($L_1(o)$ and $L_3(o)$), the entry unlock ($U_2(o)$), and the `THROUGH` state denoting the existence of paths in g that do not have any locks/unlocks.

The exit summary for g summarizes: (a) the possible lock(s) in g that can be paired with the lock ($L_E(o)$)/unlock ($U_E(o)$) in f, and (b) the possibility that a lock before calling g can

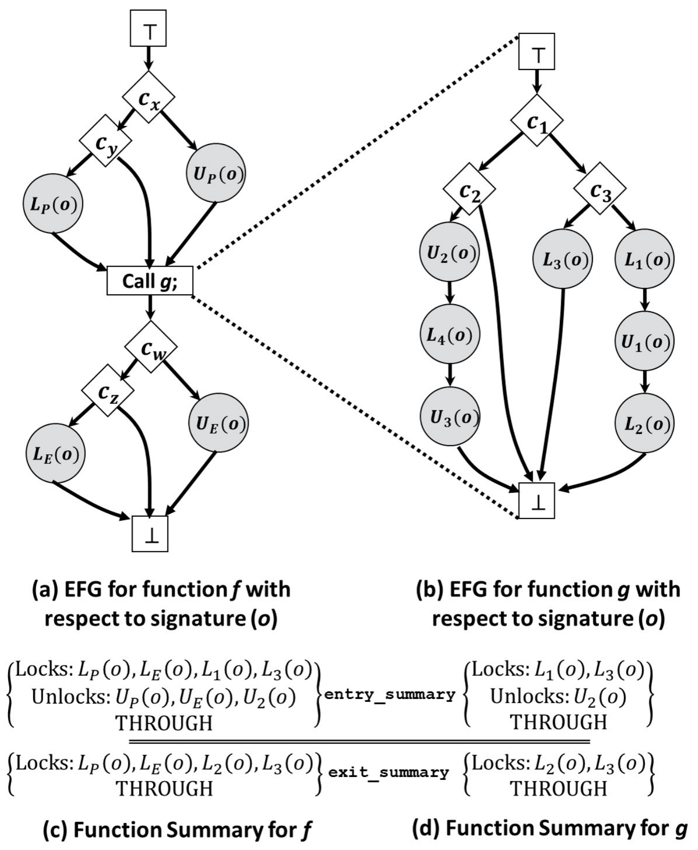

⊤
c_x
c_y
$L_P(o)$
$U_P(o)$
Call g;
c_w
c_z
$L_E(o)$
$U_E(o)$
⊥
⊤
c_1
c_2
c_3
$U_2(o)$
$L_4(o)$
$U_3(o)$
$L_3(o)$
$L_1(o)$
$U_1(o)$
$L_2(o)$
⊥
(a) EFG for function f with respect to signature (o)
(b) EFG for function g with respect to signature (o)
Locks: $L_P(o), L_E(o), L_1(o), L_3(o)$ Unlocks: $U_P(o), U_E(o), U_2(o)$ THROUGH
entry_summary
Locks: $L_1(o), L_3(o)$ Unlocks: $U_2(o)$ THROUGH
Locks: $L_P(o), L_E(o), L_2(o), L_3(o)$ THROUGH
exit_summary
Locks: $L_2(o), L_3(o)$ THROUGH
(c) Function Summary for f
(d) Function Summary for g

Figure 6.2 Compact function summaries for caller (f) and callee (g)

be paired with a lock/unlock after calling g. Case (b) is possible if there is a `THROUGH` state in g. The exit summary for g - denoted by `exit_summary` in Figure 6.2(d) - includes: the exit locks ($L_2(o)$ and $L_3(o)$) and the `THROUGH` state.

Since the pairing algorithm is traversing the MPG_o in a bottom-up manner, the summary for g is available to compute the summary for f. Figure 6.2(c) shows the entry and exit summaries for f. Note that: (1) the entry summary for g is part of the entry summary of f because the entry locks/unlocks in g can be the entry locks/unlocks for f too, and (2) the exit summary of g is part of the exit summary of f as the exit locks in g can be the exit locks in f.

6.2.4.2 Pairing Algorithm

Listing 6.1 describes the pairing algorithm. It iterates over the set of signatures and takes as input: the matching pair graph MPG_o, the EFG for each function within MPG_o. Then, it outputs: the set of lock/unlock pairs, unpaired locks, and deadlocks (if any).

```
 1 main($MPG_o$)
 2   functions ← reverse_topological_sort($MPG_o$);
 3   for(each function in functions)
 4     efg ← get_EFG(function);
 5     entry_node ← get_entry_node(efg);
 6     exit_node ← get_exit_node(efg);
 7     node_summary ← {pre_sum: {}, post_sum: {}};
 8     traverse_efg(entry_node, node_summary);
 9     summary.entry_summary ← pre_sum for entry_node;
10     summary.exit_summary ← post_sum for exit_node;
11     summaries.put(function, summary);
12     if (function ∈ roots($MPG_o$)) AND (summary.exit_summary contains a lock)
13       report the lock(s) in summary.exit_summary as unpaired lock(s);
14 end
15
16 traverse_efg(node, ns)
17   if node contains a lock function call
18     if ns.post_sum contains a lock
19       report a potential deadlock between the current lock the lock(s)in ns.post_sum.
20     update the pre_sum and post_sum of ns with the current lock.
21   else if the node is a call-site for function within $MPG_o$
22     sum ← summaries.get(called function by node);
23     if (ns.post_sum contains a lock) AND (sum.entry_summary contains a lock)
24       report a potential deadlock between the lock(s) in ns.post_sum and the lock(s) in
             sum.entry_summary;
25     ns.pre_sum ← sum.entry_summary;
26     ns.post_sum ← sum.exit_summary;
27   else if node is unlock
28     update the pre_sum and post_sum of ns with this unlock;
29
30   if node is visited before AND ns holds the same summary when the node previously
       visited
31       return ns.pre_sum;
32
33   pre_sum ← {};
34   for(each s in successors(node))
35     pre_sum += traverse_efg(s, ns);
36   if ns.post_summary contains a lock AND pre_summary contains an unlock
37     report lock/unlock pairing between the lock(s) in ns.post_summary and the unlock(s)
         in pre_sum;
38   update ns.pre_sum with pre_sum;
39   return ns.pre_sum;
40 end
```

Listing 6.1 Pairing Algorithm

The pairing algorithm (Listing 6.1) starts by visiting the matching pair graph MPG_o in a bottom-up manner (lines 2-13). For each function in MPG_o, the algorithm retrieves the EFG (line 4), gets the entry/exit nodes (lines 5-6), and passes the entry node and its empty node summary to the function `traverse_efg` to start the EFG traversal in a depth-first manner (line 8). Upon the return of `traverse_efg` (lines 9-12), the function summary for `function` is computed as follows: the `entry_summary` is the same as the entry summary (`pre_sum`) for the EFG entry node (line 9), and the `exit_summary` is equivalent to the (`post_sum`) of the EFG exit node (line 10). This function summary is then stored in a global structure (`summaries`) for later use (line 11). Lines (12-13) check whether the current function is one of the roots in MPG_o and the `exit_summary` of its summary contains a lock. If so, it reports the lock(s) in `exit_summary` as unpaired locks.

Lines (17-20) of function `traverse_efg` check whether the currently visited node (`node`) contains a lock function call, if that is the case: the algorithm checks if there is a potential deadlock by checking if the `post_sum` of the previously visited node contains a lock. If so, it reports a potential deadlock between the lock in `ns.post_sum` and the current lock at `node`. Finally, it updates the current node summary with the current lock. In lines (21-26), the algorithm checks if the current node (`node`) is a call-site for a function within MPG_o, then if the `entry_summary` of the called function contains a lock and the `post_sum` of previously visited node contains a lock (line 23): the algorithm reports a potential deadlock between the lock(s) in `ns.post_sum` and the lock(s) in `sum.entry_summary` (line 24). At lines 25-26, the current node summary is updated with the summary of the called function. In case of the current visited node is an unlock function call (lines 27-28), then the algorithm updates the current node summary with the current unlock.

In our pairing algorithm, `traverse_efg` can visit an EFG node multiple times if new information that affects the locking/unlocking analysis is present. At lines (30-31), the algorithm checks whether the current node is visited before and it stops traversing through this node, if the current node summary is the same as the one when previously visited. In other words, if no new information is presented at this node, then no need to re-visit the node. Otherwise, the traversal continues to line 33. Lines (33-35) iterate through the successors of the current node and passes each successor to `traverse_efg` to recursively visit subsequent nodes. Upon the return of `traverse_efg`, the `pre_sum` is updated with the entry summary for each of its successors. Once iterating through the successors is completed (lines 36-37), the algorithm checks whether the `post_sum` of the current node contains a lock and the `pre_sum` of the successors contains an unlock, if that is the case: the algorithm reports locks/unlocks pairing between the lock(s) in `ns.post_sum` and the unlock(s) in `pre_sum` of successors. Then at line 38, the algorithm updates the `pre_sum` of the current node's with the `pre_sum` of successors. Finally, the updated `pre_sum` for the current node is returned (line 39).

6.2.5 Step 5: Feasibility Check for Potential-Error Paths

A path on which a lock is not paired with an unlock may or may not be an error depending on whether the path is feasible or not. This also applies to a path that has a lock paired with another lock. Thus, checking feasibility of such paths is required to avoid false positives. Unlike existing approaches to static lock/unlock pairing, $\mathcal{L}$-SAP does not encode any information about path feasibility throughout the lock/unlock pairing algorithm. Instead, it only checks the feasibility of *potential-error paths*. These are the paths that have a lock that is : (1) not paired with any unlocks, or (2) paired with a lock with the same signature (potential deadlock). $\mathcal{L}$-SAP applies feasibility analysis to EFG paths instead of CFG paths. This is because the EFG contains fewer paths/conditions than its corresponding CFG. Currently, $\mathcal{L}$-SAP can perform intra-procedural feasibility analysis as follows:

Let the lock L_p in function A be unpaired lock, and p be the path that has L_p. Then, if p is an intra-procedural path, $\mathcal{L}$-SAP will check the feasibility of the EFG path p. If p is an inter-procedural path (i.e., across functions), then $\mathcal{L}$-SAP will only check the feasibility of the sub-path of p contained within A. We call the intra-procedural EFG path that will be checked for feasibility, a *potential-error path*.

To check the feasibility of a potential-error path, $\mathcal{L}$-SAP traverses this path and calculates a *path condition* which is the conjunction (AND) ($\wedge$) of the branch conditions along that path. These are the conditions that must be true for the path to be executed. Then, it translates the path condition to Boolean expressions by assigning a Boolean variable to each condition. To reveal the branch condition correlations in the Boolean expressions, $\mathcal{L}$-SAP uses textual equality. For example, let (`error` $\wedge$ `!error`) be a path condition for a problematic path. Then, by assigning each condition a boolean variable, the corresponding boolean expression will be: $c_1 \wedge c_2$ where ($c_1 =$ `error`) and ($c_2 =$ `!error`). Based on the textual equality between `error` in c_1 and c_2, $\mathcal{L}$-SAP can infer the correlation: $c_2 = !c_1$. Thus, the resulting boolean expression should be: $c_1 \wedge !c_1$. This final expression's satisfiability will be tested to determine the problematic path's feasibility.

Finally, $\mathcal{L}$-SAP uses BDDs [92] to check the satisfiability of the resultant boolean expressions for each potential-error path to determine its feasibility. In future, we hope to advance our feasibility analysis to cover inter-procedural potential-error paths and to infer more accurate branch correlations via techniques such as: global value numbering [34] or constant propagation [90].

6.3 Evidence-Enabled Verification Using $\mathcal{L}$-SAP

The $\mathcal{L}$-SAP tool is implemented using Atlas [37] from EnSoft [3]. Atlas is a platform to develop program comprehension, analysis, and validation tools. Both Atlas platform and $\mathcal{L}$-SAP are available for download as Eclipse plugins at [4]. Atlas first compiles the given source code and creates a graph database of relationships between program artifacts. Then, one can interactively comprehend and/or analyze the source code using Atlas's interpreter or by writing Java programs to analyze the source code using Atlas APIs.

$\mathcal{L}$-SAP leverages the Atlas's query capabilities for lock/unlock mapping (Section 6.2.1) and generating the matching pair graph (Section 6.2.2). It also uses Atlas APIs to implement: the algorithm to transform a CFG into an EFG (Section 5.6), the lock/unlock pairing algorithm (Listing 6.1), and the feasibility analysis (Section 6.2.5). Figure 6.3 shows a screenshot of the different components of $\mathcal{L}$-SAP and Atlas.

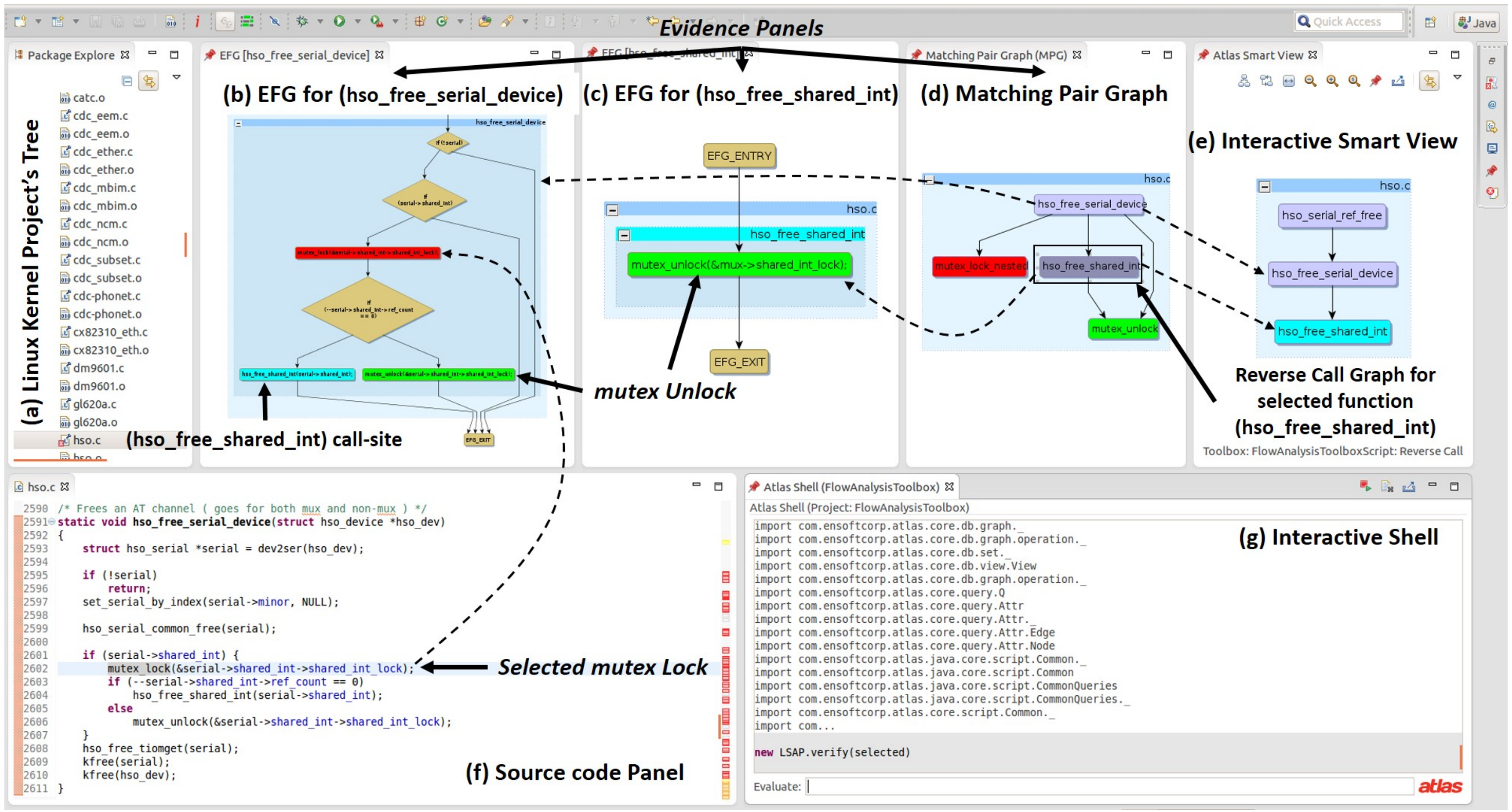
Pre-Analysis Setup. First, the user downloads and imports the Linux kernel version of interest into Eclipse. Then, Atlas compiles the imported kernel project based on the user-provided configurations and creates a graph database of relationships between program artifacts. This process takes around 2 hours[1] for latest kernel version. Then, the user runs $\mathcal{L}$-SAP on the imported Linux kernel project via the interactive shell (Figure 6.3(g)) by invoking the script (`LSAP.verify()`).

$\mathcal{L}$-SAP supports three operational modes: (a) automated verification, (b) interactive verification, and (c) team verification. $\mathcal{L}$-SAP can be rerun in different operational modes without having to recreate the graph database.

6.3.1 Automated Verification

The $\mathcal{L}$-SAP tool breaks the lock/unlock pairing verification problem into verification instances where each instance corresponds to a lock call. $\mathcal{L}$-SAP automatically verifies as many instances as possible with a strong inherent guarantee of correctness. The $\mathcal{L}$-SAP verification results fall into three categories: ($\mathcal{C}1$) *safe instances*: the automatically verified instances with

[1]Based on 2.70 GHz Intel Xeon CPU machine with 128 GB memory, running Ubuntu Linux 14.



```
/* Frees an AT channel ( goes for both mux and non-mux ) */
static void hso_free_serial_device(struct hso_device *hso_dev)
{
	struct hso_serial *serial = dev2ser(hso_dev);

	if (!serial)
		return;
	set_serial_by_index(serial->minor, NULL);

	hso_serial_common_free(serial);

	if (serial->shared_int) {
		mutex_lock(&serial->shared_int->shared_int_lock);
		if (--serial->shared_int->ref_count == 0)
			hso_free_shared_int(serial->shared_int);
		else
			mutex_unlock(&serial->shared_int->shared_int_lock);
	}
	hso_free_tiomget(serial);
	kfree(serial);
	kfree(hso_dev);
}
```

```
Atlas Shell (Project: FlowAnalysisToolbox)
import com.ensoftcorp.atlas.core.db.graph._
import com.ensoftcorp.atlas.core.db.graph.operation._
import com.ensoftcorp.atlas.core.db.set._
import com.ensoftcorp.atlas.core.db.view.View
import com.ensoftcorp.atlas.core.db.graph.operation._
import com.ensoftcorp.atlas.core.query.Q
import com.ensoftcorp.atlas.core.query.Attr
import com.ensoftcorp.atlas.core.query.Attr._
import com.ensoftcorp.atlas.core.query.Attr.Edge
import com.ensoftcorp.atlas.core.query.Attr.Node
import com.ensoftcorp.atlas.java.core.script.Common._
import com.ensoftcorp.atlas.java.core.script.Common
import com.ensoftcorp.atlas.java.core.script.CommonQueries
import com.ensoftcorp.atlas.java.core.script.CommonQueries._
import com.ensoftcorp.atlas.core.script.Common._
import com...

new LSAP.verify(selected)

Evaluate:
```

Figure 6.3 The $\mathcal{L}$-SAP tool and the automated verification of the selected lock in function `hso_free_serial_device`

no violation, ($\mathcal{C}2$) *unsafe instances*: the automatically verified instances with one or more violations shown by missing unlock(s) on feasible path(s), and ($\mathcal{C}3$) *inconclusive instances*: the remaining instances where the verification is inconclusive.

6.3.2 Interactive Verification

The interactive verification mode is primarily to complete the verification of inconclusive instances. However, it can be handy to cross-check safe and unsafe instances. The interactive modes are:

1. Verifying Selected Driver. The user selects, from the project's tree panel (Figure 6.3(a)), a subset of source files corresponding to one or multiple drivers and invokes $\mathcal{L}$-SAP to verify the locks in those drivers.

2. Verifying Selected Lock. The user selects a lock from the source code or from a previously constructed visual model (by clicking on the node corresponding to the lock).

6.3.2.1 Illustration of Interactive Verification

The user selects the marked lock in the source code panel (Figure 6.3(f)) and invokes $\mathcal{L}$-SAP (`LSAP.verify(selected)`) through the interactive shell (Figure 6.3(g)). $\mathcal{L}$-SAP verifies the instance and produces the visual models as the supporting evidence in other panels as shown in Figures 6.3(b)-(d).

6.3.3 Visual Models for Evidence

$\mathcal{L}$-SAP enables interactive reasoning. The user can interact visually and programmatically to augment or refine the automatically produced evidence. $\mathcal{L}$-SAP uses the MPG model to assist with the inter-procedural reasoning by identifying relevant functions and their interactions, and the EFG model to assist with intra-procedural reasoning for each relevant function.

6.3.3.1 Illustration of Visual Models as the Evidence

Figure 6.3(d) shows the MPG that is used for pairing the selected lock in function `hso_free_serial_device` in Figure 6.3(f). Figures 6.3(b) and 6.3(c) show the EFGs for `hso_free_serial_device` and hso_free_shared_int, respectively. The red, green, and cyan highlighted nodes correspond to the lock, its paired unlocks and the call-site for function hso_free_shared_int which belongs to the MPG. It is easy to observe from the EFG for 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 hso_free_shared_int (inter-procedural). The EFG for hso_free_shared_int shows a matching unlock on all paths. This evidence simplifies the inter-procedural cross-check to conclude that the automatic verification is correct.

The visual models are especially valuable to understand complex cases. For example, the CFG for dst_ca_ioctl in (v3.19-rc1) has 349 nodes and 163 branch nodes. The corresponding EFG has only 2 nodes and no branch nodes. Thus, the EFG is more effective as evidence compared to the CFG. If the user has any doubts or wishes to get more details, the user can query the CFG and compare it with the EFG.

6.3.3.2 Illustration of Visual Interaction

There is a 2-way source correspondence between the displayed visual models and source code. One can click on a function node in a visual model (e.g., MPG) to open up the EFG for the selected function. The EFG nodes correspond to statements in the source code. One can click on an EFG node to observe the corresponding source code. Additionally, there are *Interactive Smart Views* (Figure 6.3(e)). For example, when the user clicks on function hso_free_shared_int in Figure 6.3(d), the reverse call graph smart view (if selected) will instantly produce the reverse call graph for the selected function as shown in Figure 6.3(e).

6.3.3.3 Illustration of Programmable Interaction

The user can programmatically query and mine the graph database of the imported project to gather additional evidence via *Interactive Shell* (Figure 6.3(g)). Figure 6.4 shows the visual models for the lock in function drxk_gate_crtl reported as inconclusive instance by $\mathcal{L}$-SAP. Figures 6.4(a), (b) and (c) show the MPG, EFG, and CFG. The MPG shows that function drxk_gate_crtl calls lock and unlock, however, the EFG shows that the lock is *not* matched by an unlock. The corresponding CFG shows the lock and unlock are not matched because they are on disjoint paths: if $C =$ true, the lock occurs, otherwise, the unlock occurs.

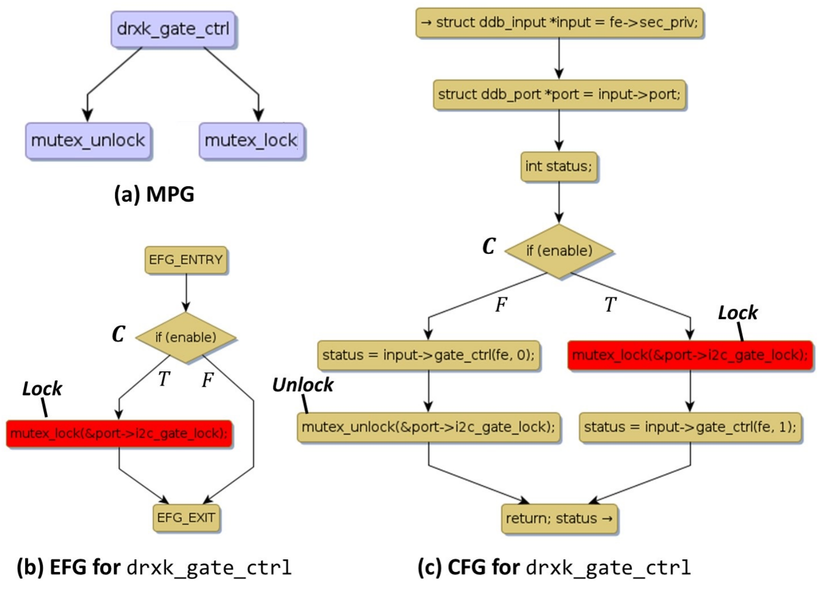

drxk_gate_ctrl
mutex_unlock
mutex_lock
(a) MPG
EFG_ENTRY
C
if (enable)
T
F
Lock
mutex_lock(&port->i2c_gate_lock);
EFG_EXIT
(b) EFG for drxk_gate_ctrl
→ struct ddb_input *input = fe->sec_priv;
struct ddb_port *port = input->port;
int status;
C
if (enable)
F
T
Lock
status = input->gate_ctrl(fe, 0);
mutex_lock(&port->i2c_gate_lock);
Unlock
mutex_unlock(&port->i2c_gate_lock);
status = input->gate_ctrl(fe, 1);
return; status →
(c) CFG for drxk_gate_ctrl

Figure 6.4 Visual models for `drxk_gate_crtl` hint to presence of calls via function pointers

The user hypothesizes that lock and unlock can match if `drxk_gate_crtl` is called twice, first with C = `true` and then with C = `false`. Through a quick query, the user observes that `drxk_gate_crtl` is not called directly anywhere. The user hypothesizes that `drxk_gate_crtl` is called using a function pointer. This interactive reasoning for discovering the functions called via function pointers can be conducted programmatically using the interactive shell (Figure 6.3(g)) to ask the following questions: What are the functions that: (Q1) set function pointers to `drxk_gate_crtl`?, (Q2) communicate the function pointer?, and (Q3) invoke calls via the function pointer? This interactive verification led to the discovery of an actual bug.

6.3.4 Team Verification

$\mathcal{L}$-SAP enables collaborative cross-checking. It creates a website (Figure 6.5) with the verification evidence for each instance. The website enables classroom projects to be conducted easily. It also enables Linux developers to share specific verification instances with their colleagues. A website created by $\mathcal{L}$-SAP can be found at [6]. The website has all the verification instances and their corresponding evidence for the kernel versions: v3.17-rc1, 3.18-rc1, and 3.19-rc1.

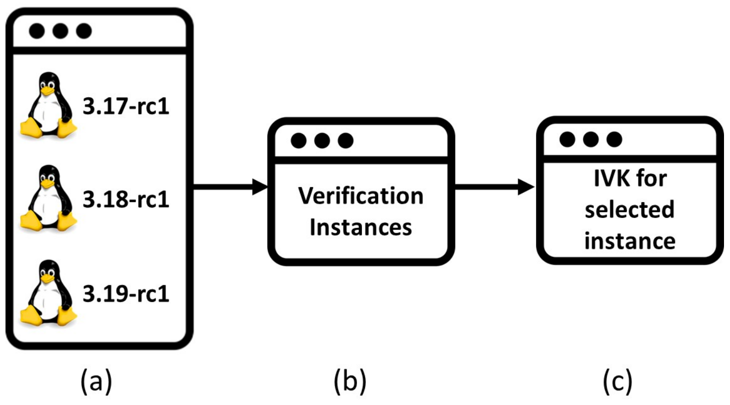


Figure 6.5 Website hierarchy

6.4 Empirical Evaluation & Results

In this section, we discuss lock/unlock pairing analysis results obtained by evaluating $\mathcal{L}$-SAP on three recent versions of the Linux kernel. These three versions together have 37 million lines of complex multi-threaded C code. To evaluate $\mathcal{L}$-SAP, we have considered three evaluation criteria: (1) competitiveness against the currently top-rated Linux kernel device driver verification tool (LDV) [5], (2) analysis speed, and (3) pairing accuracy. Finally, we present examples of verification instances from the three result categories ($\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$) to give a qualitative sense of how the automatically generated evidence simplifies the verification task. All the verification instances and their accompanied evidence (i.e., MPG, EFG, and CFG) are publicly available on the website [6]. Our experiments were done on a 2.70 GHz Intel Xeon CPU machine with 128 GB memory, running Ubuntu Linux 14.

6.4.1 $\mathcal{L}$-SAP: The Lock/Unlock Pairing Analysis Tool

6.4.1.1 Empirical Setup

We used $\mathcal{L}$-SAP to analyze 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. In Table 6.2, columns `LOC`, `Functions`, `Build`, `Nodes`, `Edges`, and `Time` show for each kernel version the numbers for lines of code, functions, build time, nodes and edges in the graph database pre-computed by Atlas and the time for this pre-computation.

Table 6.2 Linux Kernel Artifacts

Kernel	LOC	Functions	Build	Index (Graph Database)		
				Nodes	Edges	Time
3.17-rc1	12.3 M	571,012	15m 12s	43.1 M	133.4 M	2h 14m
3.18-rc1	12.3 M	571,498	15m 48s	43.2 M	133.6 M	2h 5m
3.19-rc1	12.4 M	577,650	16m 29s	43.4 M	134.2 M	2h 15m

$\mathcal{L}$-SAP is applied to pair the locks/unlocks for the `mutex` and `spin` synchronization mechanisms in the Linux kernel (Table 6.1).

6.4.1.2 Experimental Results

We compare $\mathcal{L}$-SAP against the Linux Driver Verification tool (LDV) [5] which is the current top-rated tool in the software verification competition (SV-COMP) [18–20] in the Linux device drivers verification category. The LDV's developers were generous to provide us with the LDV's results on the same versions of the Linux kernel and the same build configurations we have used.

Table 6.3 shows the comparison of $\mathcal{L}$-SAP and LDV. Column `Type` identifies the synchronization mechanism. Columns `Sigs`, `Locks` and `Unlocks` show the numbers of signatures and lock/unlock calls. For instance, in kernel 3.18-rc1, for the `spin` synchronization, the numbers for signatures, lock calls, and unlock calls are respectively 2180, 14265, and 16917. Note that a lock may be paired with multiple unlocks on different execution paths. Columns `C1`, `C2` and `C3` show the numbers of lock instances that fall in each of the three results categories $\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$, respectively.

$\mathcal{L}$-SAP is fast and scalable; it generates in 3 hours $66,609$ instances and accurately verifies $66,151$ (99.3%) instances ($\mathcal{C}1$ category). To date no errors have been discovered through manual cross-checks of automatically verified instances. The remaining 458 instances ($\mathcal{C}2$ and $\mathcal{C}3$ categories) are verified manually using the automatically generated evidence. We have discovered 8 synchronization bugs that we reported to the Linux community and subsequently confirmed. Of these, 7 bugs were automatically found and one bug was manually discovered while analyzing an instance (Section 6.4.2.3) in category $\mathcal{C}3$.

Table 6.3 Comparison of $\mathcal{L}$-SAP and LDV

Kernel	Type	Sigs	Locks	Unlocks	$\mathcal{L}$-SAP				LDV			
					$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Analysis Time	$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Analysis Time
3.17-rc1	`spin`	2,165	14,180	16,817	14,097 (99.4%)	1	82	53m 20s	8,962 (63.2%)	0	5,218	26h 16m
	`mutex`	1,687	7,887	9,497	7,813 (99.1%)	1	73	14m 55s	5,494 (69.7%)	0	2,393	26h 31m
3.18-rc1	`spin`	2,180	14,265	16,917	14,188 (99.5%)	3	74	53m 57s	9,152 (64.2%)	0	5,113	30h 22m
	`mutex`	1,664	7,893	9,550	7,801 (98.8%)	0	92	12m 59s	5,427 (68.8%)	0	2,466	29h 40m
3.19-rc1	`spin`	2,206	14,393	17,026	14,314 (99.5%)	2	77	53m 25s	9,204 (63.9%)	0	5,189	31h 55m
	`mutex`	1,700	7,991	9,653	7,938 (99.3%)	0	53	15m 29s	5,527 (69.2%)	0	2,464	29h 12m
All Kernels		**11,602**	**66,609**	**79,460**	**66,151 (99.3%)**	**7**	**451**	**3h 24m**	**43,766 (65.7%)**	**0**	**22,843**	**172h 56m**

In the case of LDV: there are 43, 766 (65.7%) verified instances, however, there is no guarantee of correctness. LDV does not report any instances for category $\mathcal{C}2$, i.e. it misses the 7 bugs found by $\mathcal{L}$-SAP. Of the 66, 609 verification instances, LDV gives inconclusive answers for 22, 843 instances. LDV produces voluminous evidence which is also not easy to understand.

Column `Analysis Time` denotes the total time needed for each analysis. $\mathcal{L}$-SAP takes 53 minutes for `spin` locks, and (12 to 15) minutes for `mutex` locks. Overall, $\mathcal{L}$-SAP takes three hours for completing the analysis of three versions of the Linux kernel while LDV takes 172 hours.

6.4.2 Case Studies & Qualitative Assessment of Visual Models

In this section, we present examples of verification instances from the three result categories ($\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$) to give a qualitative sense of how the automatically generated evidence simplifies the verification task. The visual models are meant to produce evidence to reduce the burden on the human analyst. Optimizing the visual models is crucial for reducing the human effort. All the verification instances and their accompanied evidence (i.e., MPG, EFG, and CFG) are publicly available on the website [6].

6.4.2.1 An Example from $\mathcal{C}1$ Category

This section presents an example to show the manual effort it would take an analyst to cross-check an instance that $\mathcal{L}$-SAP verifies automatically and reports no violation. This example belongs to category $\mathcal{C}1$ which has 99.3% instances. Figure 6.6 shows the visual models for a lock that $\mathcal{L}$-SAP has reported to be correctly matched. Figure 6.6(a) shows the MPG for the lock in the function `hso_free_serial_device`. Figures 6.6(b) and 6.6(c) show the EFGs for the MPG functions `hso_free_shared_int` and `hso_free_serial_device`, respectively.

In this example, it is easy to observe from the EFG of function `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 EFG of the called function `hso_free_shared_int` (Figure 6.6(b)) shows a matching

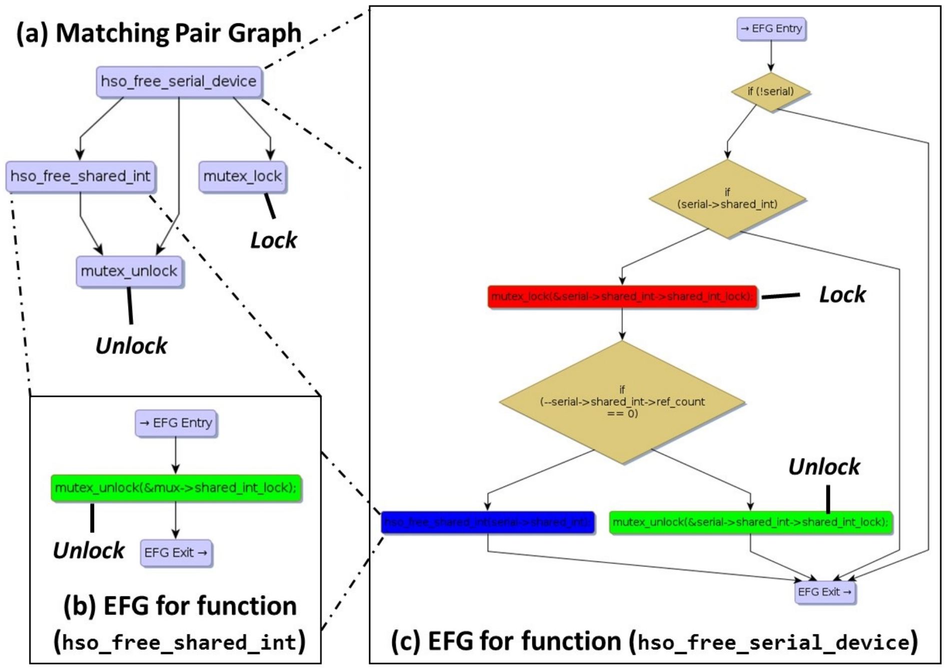

(a) Matching Pair Graph
hso_free_serial_device
hso_free_shared_int
mutex_lock
Lock
mutex_unlock
Unlock
→ EFG Entry
mutex_unlock(&mux->shared_int_lock);
Unlock
EFG Exit →
(b) EFG for function (hso_free_shared_int)
→ EFG Entry
if (!serial)
if (serial->shared_int)
mutex_lock(&serial->shared_int->shared_int_lock);
Lock
if (--serial->shared_int->ref_count == 0)
Unlock
hso_free_shared_int(serial->shared_int);
mutex_unlock(&serial->shared_int->shared_int_lock);
EFG Exit →
(c) EFG for function (hso_free_serial_device)

Figure 6.6 Visual models for an automatically verified instance

unlock on all paths within the called function. This evidence simplifies the inter-procedural cross-check to conclude that the automatic verification is correct.

Bugs Discovery. Below, we present two bug examples. The first example belongs to category $\mathcal{C}2$ (7 instances) where $\mathcal{L}$-SAP correctly reports a bug. The second example shows an inconclusive case from category $\mathcal{C}3$ (451 instances) where the evidence produced combined with interactive reasoning shows a bug. A listing of the 8 bugs reported in this paper can be found at [9].

6.4.2.2 A Bug Example from $\mathcal{C}2$ Category

Figure 6.7 shows the visual models for a discovered bug. This bug was discovered automatically by $\mathcal{L}$-SAP and then cross-checked manually. Figure 6.7(a) shows the MPG for the lock in the function `toshsc_thread_irq`. Figure 6.7(b) shows the EFG for `toshsc_thread_irq`.

The EFG for `toshsc_thread_irq` shows a path on which the lock in not followed by an unlock. As seen from the EFG, the path is feasible if the boolean expression $(C_1\overline{C_2})$ is true. To complete

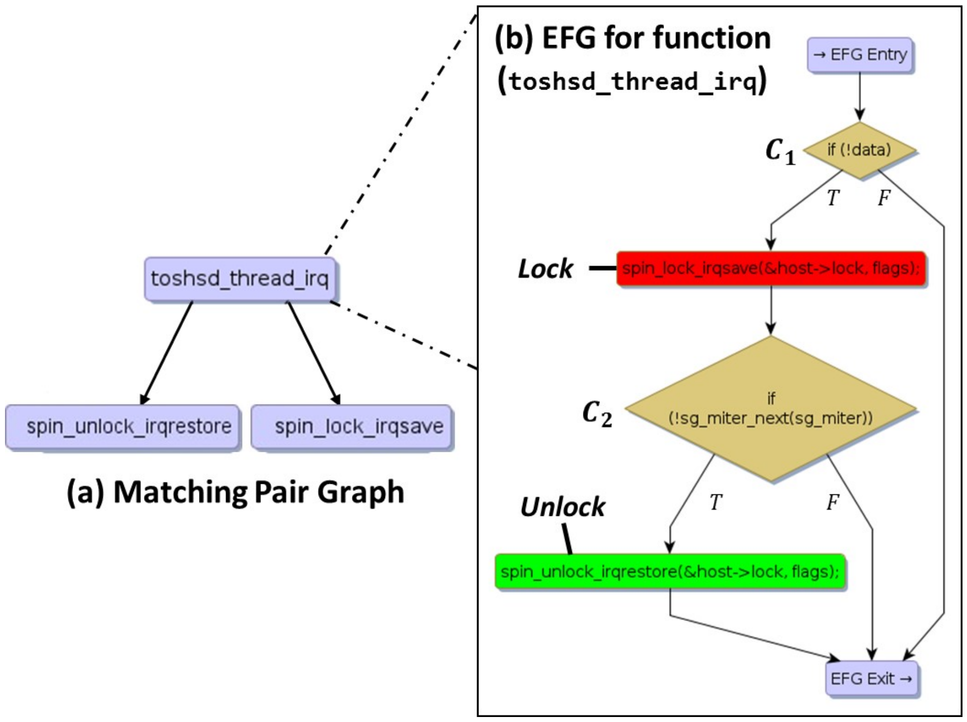


Figure 6.7 A bug discovery using visual models

the verification, one must verify that the boolean expression is satisfiable and concludes that the automatically reported violation is indeed a violation. This bug was reported to the Linux organization and it is fixed.

6.4.2.3 A Bug Example from $\mathcal{C}3$ Category

This example brings out interactive reasoning where the querying capability in Atlas is crucial. The MPG points to the possibility of functions relevant to the verification, but may be missing because they may have been called using function pointers. The querying capability is needed to find these functions.

Figure 6.8 shows the visual models for the lock in function `drxk_gate_crtl` reported as unpaired by $\mathcal{L}$-SAP. Figures 6.8(a), (b) and (c) show the MPG, EFG, and CFG. The MPG shows that function `drxk_gate_crtl` calls lock and unlock, however, the EFG shows that the lock is *not* matched by an unlock. The corresponding CFG shows why they are not matched. The lock and unlock are on disjoint paths: if $C =$ `true`, the lock occurs, otherwise, the unlock occurs.

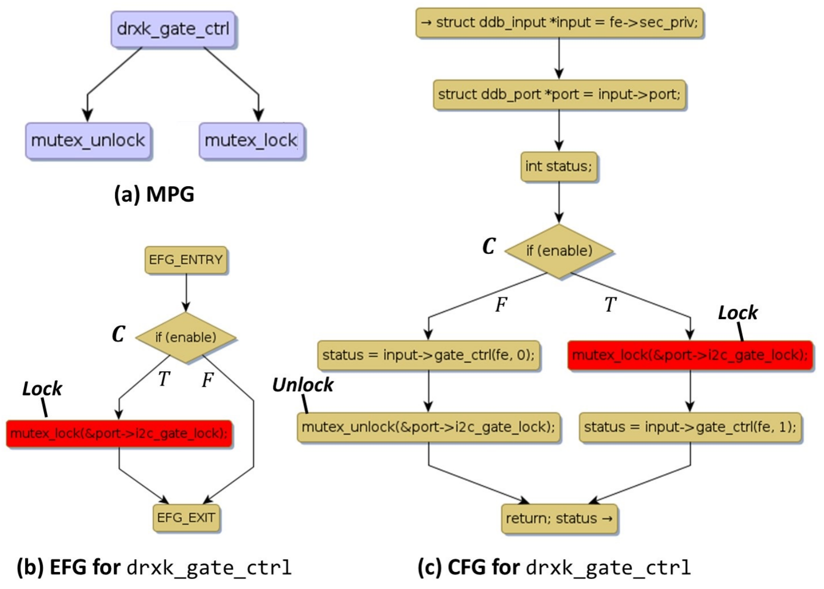

drxk_gate_ctrl
mutex_unlock
mutex_lock
(a) MPG
EFG_ENTRY
C
if (enable)
T
F
Lock
mutex_lock(&port->i2c_gate_lock);
EFG_EXIT
(b) EFG for drxk_gate_ctrl
→ struct ddb_input *input = fe->sec_priv;
struct ddb_port *port = input->port;
int status;
if (enable)
Unlock
status = input->gate_ctrl(fe, 0);
mutex_unlock(&port->i2c_gate_lock);
status = input->gate_ctrl(fe, 1);
return; status →
(c) CFG for drxk_gate_ctrl

Figure 6.8 Visual models for `drxk_gate_crtl` pointing to presence of calls via function pointers

The lock and unlock on disjoint paths could match if `drxk_gate_crtl` is called twice, first with C = `true` and then with C = `false`. This amounts to using `drxk_gate_crtl` first as a lock and then as an unlock. A quick query shows that `drxk_gate_crtl` is not called directly anywhere. Thus, it is either dead code or `drxk_gate_crtl` is called using a function pointer. It is apparent that the evidence in this example is not sufficient, however, it gives the analyst valuable clues to start with.

As shown in Figure 6.8, function `tuner_attach_tda18271` calls `drxk_gate_crtl` via function pointer. `demo_attach_drxk` sets the function pointer to `drxk_gate_crtl`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `tuner_attach_tda18271`. Figure 6.9 shows the refined MPG after it is augmented with these functions newly discovered by human intervention.

This interactive reasoning for discovering the functions called via function pointers can be conducted visually using Atlas. The queries amount to asking the following questions:

1. What are functions that set function pointers to function `drxk_gate_crtl`?

2. What are the functions that communicate the function pointer?

3. What are the functions that invoke calls via the function pointer?

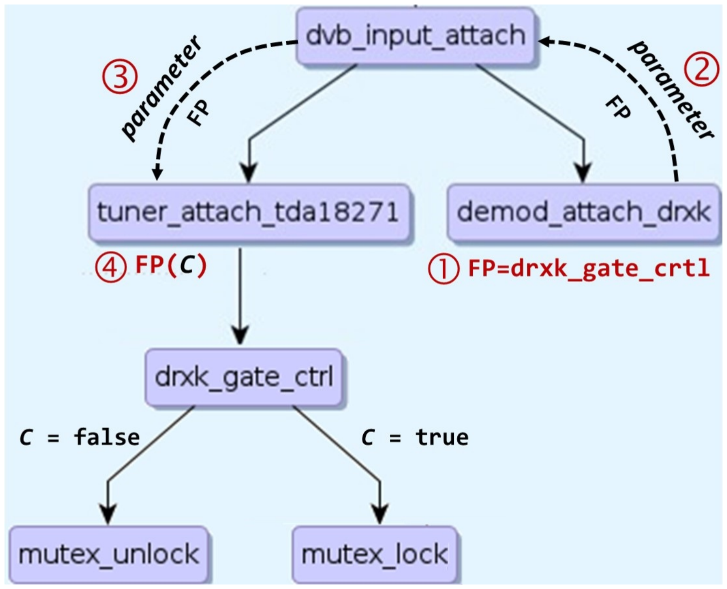


Figure 6.9 The augmented MPG for `drxk_gate_crtl` after resolving calls via function pointers

We found a bug while working on this instance. Recall that `drxk_gate_crtl` must be called twice; first it acts like a lock and then as an unlock. There is a path on which there is a return before the second call and thus a bug because the second call for unlocking does not happen on that return path.

6.4.3 Current Limitations of $\mathcal{L}$-SAP

The results in category $\mathcal{C}3$ correspond to instances where $\mathcal{L}$-SAP gives inconclusive answers for 451 ($< 0.7\%$) instances. This percentage of reported unpaired locks and deadlocks is attributed to limitations in $\mathcal{L}$-SAP's automatic verification. Table 6.4 shows the breakdown for these 451 verification instances across the following limitation factors:

- Not being able to recognize infeasibility of paths in some cases due to: (a) the use of textual equality is not advanced enough to find complex intra-procedural correlations between branch conditions, and (b) the lack of inter-procedural feasibility analysis.
- Inability to process function pointers: $\mathcal{L}$-SAP cannot track the inter-procedural cases in which a function is called via a function pointer.

- The use of signature-based analysis (Section 6.2.1): $\mathcal{L}$-SAP deems two different objects identical if their signatures match.

Table 6.4 Breakdown of instances in $\mathcal{C}3$ category across different limitation facrtos

Kernel	Type	$\mathcal{C}3$	Intra-procedural Feasibility	Inter-procedural Feasibility	Function Pointers	Signature Problems
3.17-rc1	`spin`	82	14	16	22	30
	`mutex`	73	10	31	28	4
3.18-rc1	`spin`	74	13	20	23	18
	`mutex`	92	13	35	39	5
3.19-rc1	`spin`	77	12	16	28	21
	`mutex`	53	6	22	21	4
All Kernels		**451**	**68**	**140**	**161**	**82**

The following example from category $\mathcal{C}3$ shows the case where $\mathcal{L}$-SAP reports a violation that turns out to be a verification flaw due to analysis limitation. In this example, $\mathcal{L}$-SAP reports a deadlock because a lock is followed by another lock with the same signature (Section 6.2.1). $\mathcal{L}$-SAP deems the two objects to be the same because they have identical signatures. In reality, the two objects are different but $\mathcal{L}$-SAP lacks a refined notion of signature to distinguish them.

Figure 6.10 shows the EFG for function `ucma_lock_files`. The EFG shows a lock immediately followed by another lock. It represents either an inadvertent coding error or a mistaken identity for two locks that are different but have the same signature. Here, the source correspondence is important to resolve the matter. By clicking on each lock call, one can find that the two locks operate on different objects. There are 82 instances with this issue.

6.5 Conclusions

$\mathcal{L}$-SAP is a static tool that uses a novel scalable and accurate lock/unlock pairing analysis. It uses algorithmic innovations based on a study of observed difficulties for lock/unlock pairing in the Linux kernel. We evaluated $\mathcal{L}$-SAP on three recent versions of the Linux kernel. The

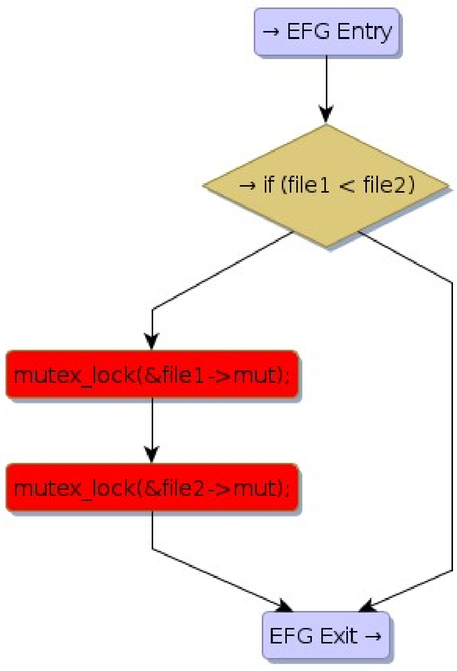


Figure 6.10 The EFG for function `ucma_lock_files` shows incorrect automatic verification

evaluation results show major accuracy and scalability improvements over the currently top-rated Linux kernel device driver verification tool (LDV) [5]. The analysis using $\mathcal{L}$-SAP has led to the discovery of eight synchronization bugs. For each pairing of a lock with corresponding unlocks with the same signature, $\mathcal{L}$-SAP produces evidence which includes the matching pair graph (MPG) of the minimum set of functions for inter-procedural analysis and the call chains between them, the event flow graphs and a compact summary for each of the functions in MPG. This evidence makes it easy for the human analyst to cross-check the results produced by $\mathcal{L}$-SAP, or to complete the analysis manually for the cases where $\mathcal{L}$-SAP reports potential-error paths but cannot provide conclusive results.

CHAPTER 7. $\mathcal{M}$-SAP: EVIDENCE-ENABLED LINUX VERIFICATION FOR ALLOCATION/DEALLOCATION PAIRING ANALYSIS

7.1 Introduction

A memory leak is a situation where an allocated memory by a program is never deallocated subsequently. This common error can have catastrophic effects on long running systems. With the increasingly larger and more complex software systems, it has become increasingly challenging to ensure resilience of software systems to memory leaks, especially in the era of multi-threading and event-processing. These memory leaks can elude dynamic analyses and regression testing because their occurrence often depends on intricate sequences of low-probability events [41].

Consider the allocation/deallocation pairing analysis: an allocation A is paired with a deallocation D iff D can deallocate the memory block allocated by A. We use the term allocation/deallocation to refer to an allocation/deallocation operation (i.e., function call). The existence of unpaired allocations on feasible execution paths results in memory leaks. This pairing of an allocation with its corresponding deallocations on all feasible execution paths requires the following: (a) a *data flow analysis* to *map* each allocation to corresponding deallocations that reference the same memory block, (b) a *control flow analysis* to *pair* each allocation with corresponding deallocations. This is achieved by *identifying* all possible intra- and inter-procedural execution paths that have an allocation ensued by a corresponding deallocation, and (c) a *feasibility analysis* to check the feasibility of an execution path on which an allocation is not ensued by a deallocation. A general-purpose, completely automated, and accurate analysis is intractable for each of the above three requirements.

We present a *scalable* and *accurate* static allocation/deallocation pairing analysis that is explicitly designed to handle the analysis roadblocks we observed in the Linux kernel. We design a pointer analysis that mines points-to information directly from source code, not from an intermediate representation of the code (Static Single Assignment (SSA) [56]), to satisfy the analysis requirement (a) to map each allocation with a set of corresponding deallocations. To efficiently meet the analysis requirement (b), we use event flow graphs (Chapter 5) to minimize the set of paths that must be examined for path-sensitive accurate analysis. We also design *compact function summaries* for scalable context-sensitive inter-procedural analysis. For requirement (c), among this minimized set of paths, we separate the paths on which an allocation is not ensued by a deallocation. By construction, this is a necessary and sufficient set of paths that should be checked for feasibility. Thus, the need for feasibility analysis is also minimized by applying it only in the cases where it is needed. A general-purpose, completely automated, and accurate analysis is intractable for each of the above three requirements.

There is a vast body of work pertaining to static techniques for memory leak detection [1, 5, 52, 55, 83, 97, 99] in C programs. These state-of-the-art approaches have led to new advances in data and control flow analyses pushing further the boundaries of scalability and accuracy. Nevertheless, these approaches have the following drawbacks: (1) the empirical evaluation of their tools is based on older versions of the Linux kernel ($<$ 4 MLOC) or on medium-sized programs that are orders of magnitude smaller, (2) the analysis is *not* exhaustive in a sense that it *only* produces results for allocation instances where memory leaks are detected and ignoring other allocations without a verdict (i.e., safe or memory leak), (3) a skin-deep description about the approach implementation details which makes it challenging for researchers and practitioners to replicate and incorporate the proposed approach, or reproduce the reported results, and (4) for each memory leak detected, the analysis produces very little evidence, voluminous evidence, or evidence that refers to the intermediate representation of the verification internals but not to the source code. Such evidence is hard to decipher; it does not simplify human validation.

Our proposed approach is to specifically mitigate these challenges by: (a) addressing commonly occurring roadblocks for scalability and accuracy to arrive at a solution that works well in practice, (b) being exhaustive by providing a verdict for each analyzed allocation instance accompanied with verification-critical evidence for manual validation, and (c) providing an infrastructure that enables researches and practitioners to easily implement an exact replica of our proposed approach and re-produce our results. The Linux kernel code base has unique combinations of specific characteristics that attract researchers and practitioners to challenge their tools [22]. This motivated us to use the Linux kernel code base as a good target system to test our approach.

We developed $\mathcal{M}$-SAP, a tool that uses our novel allocation/deallocation pairing analysis. It analyzes each allocation and it produces three categories of results: ($\mathcal{C}1$) an automatically verified instance with correct allcation/deallocation pairing, ($\mathcal{C}2$) an automatically verified instance with a memory leak i.e. a feasible path with an allocation not ensued by deallocation, and ($\mathcal{C}3$) an allocation instance where the automated verification is inconclusive. $\mathcal{M}$-SAP produces for each allocation instance the following evidence: the *memory taint graph* (MTG) (Section 7.2.1.13) to assist with the inter-procedural reasoning by identifying relevant functions and their interactions, the Event Flow Graph (EFG) (Section7.2.2) to assist with intra-procedural reasoning for each relevant function, and the Points-to graph (PtG) (Section 4.3) to assist with data flow reasoning by identifying the points-to relations between the pointers of interest at different source code locations. We have applied $\mathcal{M}$-SAP to a recent version (3.17-rc1) of the Linux kernel totalling > 12 MLOC. $\mathcal{M}$-SAP is fast and scalable, and it is able to accurately pair 92.3% $(1,060)$ of the total analyzed allocations in one hour with no false negatives. Our analysis discovered 50 memory leaks that are reported[1] to the Linux community. Compared with the Linux Driver Verification (LDV) tool [5], a state-of-the-art and top-rated verification framework in the competition on software verification (SV-COMP) [18–20] in the category of Linux device drivers verification, $\mathcal{M}$-SAP reduces by $9\times$ the number of statically unpaired allocations and by $30\times$ the analysis time.

To the best of our knowledge, we are the first to perform allocation/deallocation pairing analysis on a recent version of the Linux kernel. Our evaluation results show that $\mathcal{M}$-SAP pro-

[1]Some of the reported memory leaks have been accepted and fixed. Others are still valid in the latest branch of the Linux kernel.

vides a scalable, practical and accurate allocation/deallocation pairing analysis for the Linux kernel. $\mathcal{M}$-SAP is developed using Atlas [37], which is a platform to develop program comprehension, analysis, and validation tools.

This chapter makes the following key contributions:

1. A novel scalable and accurate allocation/deallocation pairing analysis algorithm and its implementation in the tool $\mathcal{M}$-SAP (Section 7.2.3).
2. An exhaustive pairing analysis that provides for every allocation a verdict accompanied with evidence for manual validation.
3. An abstract syntax tree (AST) parsing algorithm that mines the points-to information directly from C source code to build the corresponding PtGs (Section 7.2.1).
4. An evaluation of $\mathcal{M}$-SAP on a recent version of the Linux kernel done automatically in one hour. The experimental results show that $\mathcal{M}$-SAP correctly pairs 92.3% of the total analyzed allocations and identifies 50 memory leaks (Section 7.4).

The remainder of the chapter is organized as follows. We first describe our static allocation/deallocation pairing analysis in Section 7.2. Next, Section 7.3 describes the result categories and the produced evidence by the $\mathcal{M}$-SAP tool. Section 7.4 presents the experimental results on a recent version of the Linux kernel. Finally, we conclude in Section 7.5.

7.2 $\mathcal{M}$-SAP Approach

Figure 7.1 shows an overview of our allocation/deallocation pairing analysis. The pairing analysis has four steps.

In the first step, $\mathcal{M}$-SAP *maps* each allocation to the set of corresponding deallocations. An allocation A is mapped with a deallocation D iff D can deallocate the memory block allocated by A. The mapping is performed via a pointer analysis that tracks the allocated memory block throughout the program by maintaining PtGs of the allocated memory block (AMB) at each visited code statement. Due to inter-procedural analysis, this step results on a *memory taint graph* (MTG_A) for each allocation instance A. The nodes in this graph provide the set

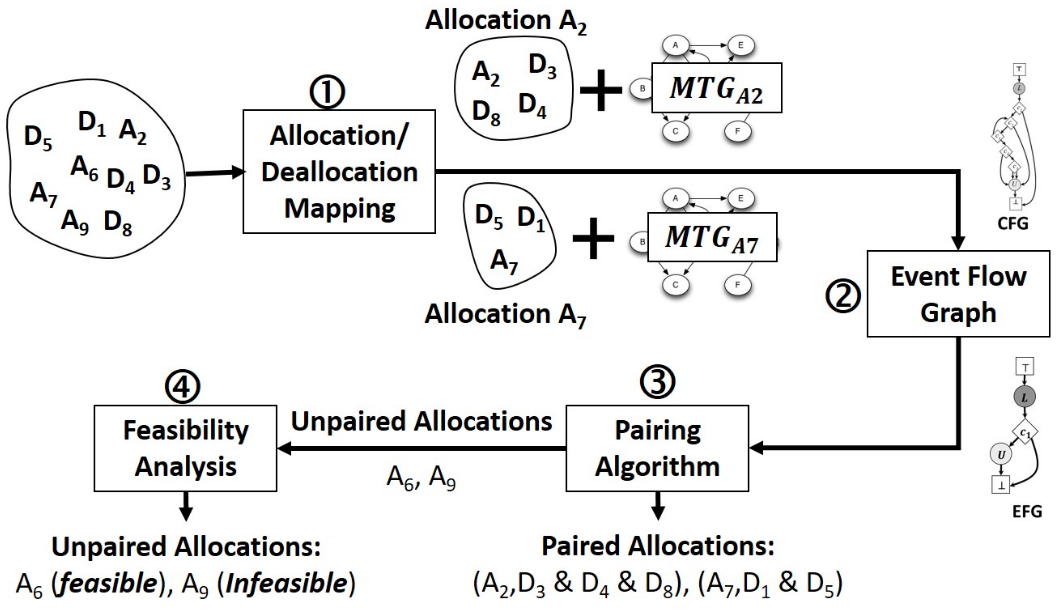


Figure 7.1 An Overview of $\mathcal{M}$-SAP Pairing Analysis

of functions for inter-procedural pairing analysis of the allocation A and the corresponding mapped deallocations. The directed edges in MTG_A represent call relationships. In the second step, for each function in MTG_A, $\mathcal{M}$-SAP prunes the CFG to produce a compact CFG named the Event Flow Graph (EFG). The EFG enables efficient path-sensitive allocation/deallocation pairing by defining an equivalence relation on the CFG paths such that it is sufficient to examine only one path from each equivalence class. These first two steps set the stage for an efficient pairing algorithm.

In the third step, $\mathcal{M}$-SAP iterates over the set of allocation instances to apply the pairing algorithm to each MTG_A and pair the allocation instance A with its corresponding deallocations. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. In the fourth step, $\mathcal{M}$-SAP calculates Boolean expressions for the conditions governing each potential-error path on which an allocation is not paired with a deallocation. Then, using BDDs [92], $\mathcal{M}$-SAP examines whether the potential-error paths are feasible. Next, we describe the four steps in detail.

7.2.1 Step 1: Allocation/Deallocation Mapping

An allocation A is mapped to a deallocation $D(m)$ iff $D(m)$ can deallocate the allocated memory block (AMB$_A$) by A. This can only happen iff m points-to AMB$_A$. To know whether m points-to AMB$_A$, we present our pointer analysis that traverses the CFG paths from A reaching $D(m)$ while building and maintaining points-to graphs (Section 4.3) for AMB$_A$. In our representation, a CFG node corresponds to a single program statement. Our pointer analysis is: field-sensitive (by distinguishing different fields in a `struct`), flow-sensitive (by tracking flow of statements), and object-sensitive (by distinguishing different allocation sites as different memory blocks (AMBs)).

State-of-the-art approaches to static memory leak detection establish their pointer analyses based on the points-to information inferred from an intermediate representation of the source code such as static single assignment (SSA) [56]. Unlike these approaches, our pointer analysis leverages the points-to information *directly* from source code by recursively traversing the abstract syntax tree (AST) node for each analyzed code statement. This results on source code correspondence for the produced evidence to simplify manual cross-checking and validation.

In the canonical form, a statement in a C program is one of the following: (1) an assignment of the form, $p = \&v$ (address), $p = q$ (copy), $p = *q$ (load) or $*p = q$ (store), (2) a call statement, $p = \mathcal{F}_k(....,q,....)$, at call-site k, where $\mathcal{F}_k$ is understood to be a function pointer (or function in the special case), and (3) a return statement, `return` p. Here, p and q are local or global variables and v is a local or global variable, or a heap object. Our pointer analysis considers a subset of the canonical form as follows: (1) an assignment of the form, $p = \&v$ (address) or $p = q$ (copy), (2) a call statement, $p = \mathcal{F}_k(....,q,....)$, at call-site k, where $\mathcal{F}_k$ is understood to be a function **not** a function pointer, and (3) a return statement, `return` p.

Listing 7.1 describes the allocation/deallocation mapping algorithm. The mapping algorithm takes as input: an allocation node (`alloc_node`), which corresponds to the occurrence of memory block allocation, and the PtG for the function containing the `alloc_node` and outputs the corresponding deallocations (if any). The mapping algorithm maintains for each function a PtG and keeps track of all returned pointers from all return statements of the current visited function in the set `returnedPtrs`. In addition, the algorithm maintains for each expression E the set E.ptrs that denotes the pointers that can pointed-to by other pointers if E happens to be a right expression of a binary expression.

```
Input: An alloc_node corresponding to an allocation callsite and the current PtG for the
    function containing the alloc_node
Output: the set of deallocation callsites corresponding to alloc_node (if any)
visit_node(CFG-Node, PtG)
    visited.add(CFG-Node);
    astNode ← getASTNode(CFG-Node);
    expressions ← getExpressions(astNode);
    for(each expression ($E$) in expressions)
        process_expression($E$);
    endfor
    for(each successor in successors(CFG-Node))
        if(successor $\notin$ visited) OR (new information added to PtG)
            visit_node(successor, PtG);
        endif
    endfor
end

process_expression($E$)
    if($E \rightarrow$ label|case|break|continue|default statement) /* CASE 1 */
        return;
    endif
    if($E \rightarrow$ Id) /* CASE 2 */
        $p$ = getIDP(Id);
        if($p$ == null)
            $p$ = new IDP(Id);
            new points-to edge: $p \rightarrow$ new MLoc();
        endif
        $E$.ptrs = $\{p\}$;
    endif
    if($E \rightarrow E_1(\rightarrow|\bullet)$Id) /* CASE 3 */
        ptr_name = concat($E_1$.ptrs.name, Id.name);
        $p$ = getFRP(ptr_name);
        if($p$ == null)
            $p$ = new FRP(ptr_name);
            new field-edge($E_1$.ptrs $\dashrightarrow p$);
            new points-to edge: $p \rightarrow$ new MLoc();
        endif
        $E$.ptrs = $\{p\}$;
    endif
    if($E \rightarrow \circledast E_1 \circledcirc$) /* CASE 4 */
        if($\circledast$ is &)
            $E$.reference = true;
        endif
        $E$.ptrs = $E_1$.ptrs;
    endif
    if($E \rightarrow E_1[E_2]$) /* CASE 5 */
        $E$.ptrs = $E_1$.ptrs
    endif
    if($E \rightarrow (E_1)E_2$) /* CASE 6 */
        $E$.ptrs = $E_2$.ptrs
    endif
    if($E \rightarrow E_L \circledast E_R$) /* CASE 7 */
        if($\circledast$ is =)
            for(each ptr in $E_L$.ptrs)
                remove all out points-to edges;
                if($E_R$.reference == true)
                    new points-to edge: ptr $\rightarrow q$ : $\forall q \in E_R$.ptrs
                else
                    new points-to edge: ptr $\rightarrow q$ : $\forall q \in$ pt($E_R$.ptrs)
                endif
                for(each $q$ contained by ptr)
                    remove all out points-to edges;
                    $M$ = find the FRP contained by $E_R$.ptrs and corresponds to $q$
                    if($E_R$.reference == true)
                        new points-to edge: $q \rightarrow i$ : $\forall i \in M$
                    else
                        new points-to edge: $q \rightarrow j$ : $\forall j \in$ pt($M$)
                    endif
                endfor
            endfor
        endif
        $E$.ptrs = $E_L$.ptrs
```

```
72        endif
73        if ($E \rightarrow$ allocation$_K(E_1)$)  /* CASE 8 */
74            $p$ = new MLoc($K$);
75            $E$.ptrs = $\{p\}$;
76        if ($E \rightarrow$ deallocation$_M(E_1)$)  /* CASE 9 */
77            if (AMB $\in$ pt($E_1$.ptrs))
78                map allocation node (alloc_node) to deallocation node (CFG-Node);
79            endif
80            remove out points-to edges from $E_1$.ptrs;
81            new points-to edge: $q \rightarrow$ new MLoc(); : $\forall q \in E_1$.ptrs
82            $E$.ptrs = $\{\}$;
83        endif
84        if ($E \rightarrow E_F(\{E_i\})$)  /* CASE 10 */
85            PtG$_{E_F}$ = clone(PtG);
86            alter_names($\{E_i\}$.ptrs, $\{p_i\}$);
87            (PtG, $E$.ptrs) = visit_node($E_F$, PtG$_{E_F}$);
88            undo_alter_names($\{E_i\}$.ptrs, $\{p_i\}$);
89        endif
90        if ($E \rightarrow$ return $E_1$;)  /* CASE 11 */
91            $E$.ptrs = $E_1$.ptrs, returnedPtrs.addAll($E$.ptrs);
92        endif
93    end
```

Listing 7.1 Allocation/Deallocation Mapping Algorithm

Our mapping algorithm (Listing 7.1) starts from the given allocation node (`alloc_node`) and traverses its containing CFG in a depth-first manner as in function `visit_node` (lines 3-15). In line 5, the algorithm retrieves the AST subtree corresponding to the code statement denoted by the CFG node `CFG-node`. Then in line 6, it retrieves all the expressions by recursively visiting the AST node. In lines (7-9), the algorithm iterates over the expressions in their *parsing precedence order* and passes each expression E to function `process_expression`. After processing all expressions, the global PtG may be altered by new points-to information. Finally, the algorithm recursively visits each successor of the current node if the successor is not visited before or the current global PtG has new information than before processing the expressions of the current node (lines 10-14).

Function `process_expression` is the core of our mapping algorithm where the updates to the global PtG occur based on the inferred information from the passed expression E. Next, we discuss how our mapping algorithm handles each of the 11 cases:

7.2.1.1 CASE 1: Special Statements

Our pointer analysis ignores statements that do not include any points-to information such as: `break`, `default`, `goto`, `continue`, and `case` statements.

7.2.1.2 CASE 2: $E \rightarrow$ Id

For an `Id` expression, the algorithm queries the PtG for an existing IDP pointer node that corresponds to the pointer represented by `Id`, if it does not exist, the algorithm creates a new IDP pointer node p, creates a new memory location node (`new MLoc()`) and creates a new points-to edge from p to the new memory location. Finally, the set of pointers associated with E consists of p.

7.2.1.3 CASE 3: $E \rightarrow E_1(\rightarrow |\bullet)$Id

Definition 9 ***Pointer Container.*** *For a given pointer p, the containing pointer C_p points-to a structure that can directly reference p through a pointer/strcuture de-referencing operation ($\rightarrow$ or $\bullet$). We use the term* ***pointer container*** *to refer to C_p of pointer p.*

For a field reference expression, the algorithm builds the pointer's name by concatenating the names of: (1) the pointer node in E_1.ptrs which corresponds to the container pointer for pointer Id, and (2) the IDP Id. Then, the algorithm queries the PtG for an existing FRP pointer corresponding to the pointer name `ptr_name`, if it does not exist, the algorithm creates a new FRP pointer node p, adds a new field edge ($\dashrightarrow$) from the container pointer in E_1.ptrs to p, and creates a new points-to edge from p to a new memory location node. Finally, the set of pointers associated with E consists of p.

7.2.1.4 CASE 4: $E \rightarrow \circledast E_1 \circledcirc$

This case corresponds to a unary expression where $\circledast$ and $\circledcirc$ denote the prefix and suffix operators. If the prefix operator is a reference operator ($\circledast = \&$), the algorithm sets the flag `reference` to `true`, so that the algorithm knows that the pointers mined from the expression E_1 are referenced. Therefore, any points-to edges should point to the pointer nodes corresponding to E_1.ptrs rather than the pointers that E_1.ptrs point to. For other prefix and suffix operators, the algorithm ignores such pointer arithmetic operations due to the complexity of tracking memory locations. Finally, the set of pointers associated with E are the same as E_1.ptrs.

7.2.1.5 CASE 5: $E \to E_1[E_2]$

Meeting an array expression, our algorithm only parses the name part (E_1) of the expression and drops the subscript expression (E_2). Thus, the E.ptrs is the same as E_1.ptrs.

7.2.1.6 CASE 6: $E \to (E_1)E_2$

Meeting a casting expression where E_1 is the casting type and E_2 is the casted expression: the algorithm only parses E_2 and drops E_1. Thus, the E.ptrs is the same as E_2.ptrs.

7.2.1.7 CASE 7: $E \to E_L \circledast E_R$

For a binary expression with an operator $\circledast$ and a left E_L and right E_R expressions, our algorithm works as follows:

- $\circledast$ is an assignment operator (=): In this case, the algorithm iterates over each pointer `ptr` in the left expression's pointers (E_L.ptrs) and performs the following:
  - Remove all the out points-to edges from `ptr`.
  - If the pointers in E_R are by-reference (the flag `reference` is `true`), then the algorithm creates a new points-to edge from `ptr` to every pointer q in E_R.ptrs. Otherwise (i.e., `reference` is `false`), the algorithm creates a new points-to edge from `ptr` to every pointer in (`pt(`E_R`.ptrs)`) which corresponds to the memory locations or pointers pointed-to by the pointers in E_R.ptrs.
  - Then, for each pointer q contained by `ptr`, the algorithm operates as follows:
    - Remove all the out points-to edges from q.
    - Find the FRP nodes (M) contained by E_R.ptrs and correspond to q. Then, create a points-to edge from q to every pointer $i \in M$ if the `reference` flag is set to `true`, otherwise, create a new points-to edge from q to every pointer or memory location pointed-to by the pointers in M.
  - Finally, the E.ptrs is the same as the E_L.ptrs.

- For all other operators, the algorithm unsoundly ignores such operators due to the complexity of tracking memory locations. Therefore, the E.ptrs is the same as the E_L.ptrs.

7.2.1.8 CASE 8: $E \rightarrow$ **allocation**$_K(E_1)$

For an allocation callsite at address K, the algorithm creates a new memory location p associated with the address K and E.ptrs contains p.

7.2.1.9 CASE 9: $E \rightarrow$ **deallocation**$_M(E_1)$

For a deallocation callsite at address M, the algorithm checks whether the pointers in E_1.ptrs can point-to the allocated memory block (AMB) node by `alloc_node`, then it reports a mapping between the allocation node `alloc_node` and the current deallocation node. Finally, it removes all out points-to edges from E_1.ptrs and creates a new points-to edge from each pointer in E_1.ptrs to a new memory location.

7.2.1.10 CASE 10: $E \rightarrow E_F(\{E_i\})$

In a function call expression: E_F denotes the expression for the function name which could resolve to either an IDP or an FRP (the case of function pointers), and $\{E_i\}$ is the set of parameters' expressions where i is the index of a parameter. The algorithm processes the function call expression as follows:

- The algorithm clones the current PtG. Then, it alters the pointer names in the cloned PtG to reflect the mapping between the passed pointers ($\{E_i\}$.ptrs) names and the formal parameter ($\{p_i\}$) names. Let us illustrate this through the example in Figure 7.2. Figure 7.2(a) shows the PtG for function L before calling function M. It shows that pointers `A` and `E->K` are passed as the first and second parameters to function M, respectively. Figure 7.2(b) shows the cloned and altered PtG: all occurrences of pointer `A` has been renamed to `x`, including the pointers where `A` is a container of other pointers. Similarly, the pointer node `E->k` is renamed to `y`.

- The algorithm passes the new cloned PtG with altered names along with the entry node for the called function E_F to function `visit_node` (Listing 7.1) to process E_F.

- Once all the CFG nodes in function E_F are visited, the algorithm clones the returned PtG and undo the name alteration occurred in first step. Finally, the E.ptrs corresponds to the returned pointers from the function call and the algorithm resumes processing other statements occurring after the E_F callsite.

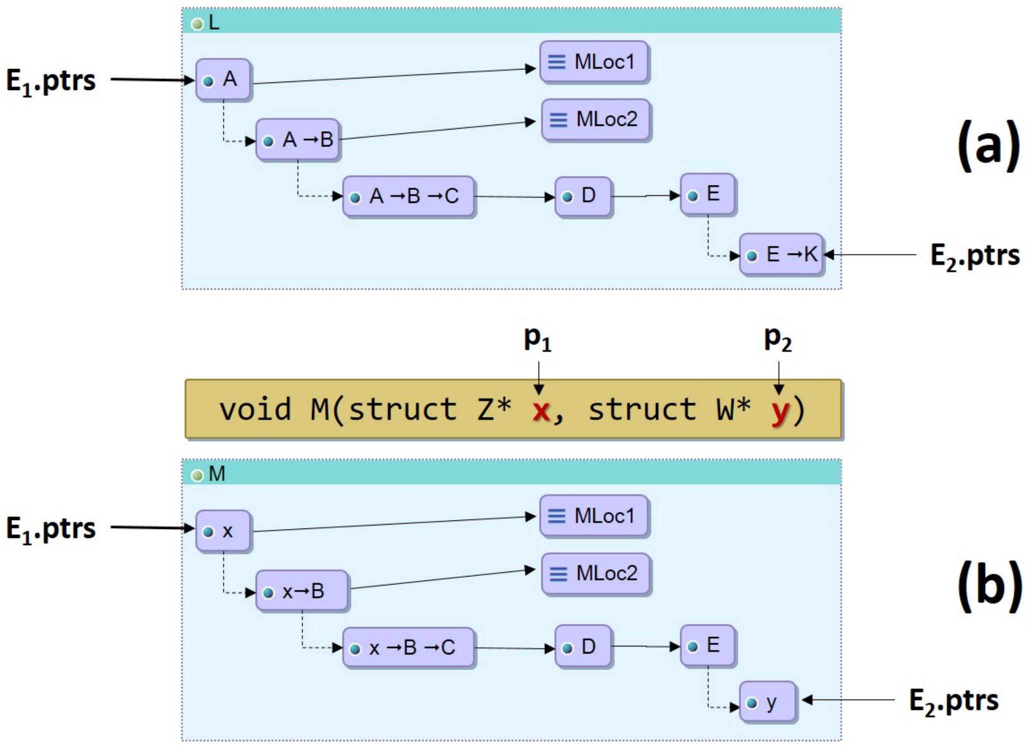


Figure 7.2 Modifying the PtG with respect to the called function formal parameter names

Currently, our mapping algorithm does not process function calls that are represented by function pointers. Moreover, it does not process library calls and other calls to commonly used functions for printing/formatting and string manipulation (e.g., `memcpy`, `strcpy`, `printf`, etc). These functions do not manipulate the passed pointers. Therefore, they are not affecting the points-to information.

7.2.1.11 CASE 11: $E \rightarrow$ return E_1;

For a C return statement, the E.ptrs is the same as the pointers mined from the expression E_1. The mapping algorithm conservatively keeps track of all pointers returned from every return statement in the set `returnedPtrs`. Once all the return statements, of the current function M, are visited, the algorithm proceeds as follows:

- The algorithm finds the callers of M. Then for each caller function L, the algorithm randomly picks a node `randNode` in the L's CFG that corresponds to a callsite for M.
- The algorithm passes the node `randNode` along with the PtG for M to function `visit_node` (Listing 7.1) to start processing through L's statements. This context switching requires cloning and name alteration for the passed PtG to match the variable names and formal parameters in function L.

7.2.1.12 Notes About Pointer Analysis

Like many other state-of-the-art approaches, our pointer analysis treats arrays, array's subscripts and double pointers as monotonic objects. In addition, the analysis is not sound in handling pointer arithmetic and self-updating arithmetic (i.e., pre/post increment/decrement) by treating, for example, an occurrence of `c = a + b` as an occurrence of `c = a`.

Leveraging and parsing source code instead of intermediate representation provides a source correspondence for the nodes and edges in the PtG to produce comprehensible evidence that facilitate manual cross-checking and validation.

7.2.1.13 Memory Taint Graph (MTG)

Throughout the allocaion/deallocation mapping algorithm (Listing 7.1), $\mathcal{M}$-SAP keeps track of all visited functions where the pointers pointing-to the allocated memory block escape directly or indirectly (i.e., through their pointer containers). We denote the induced call graph of these functions as the *memory taint graph* (MTG$_A$) (Section 4.2) for the allocated memory block by the allocation A. The taint graph is associated with only one allocation instance. This graph will be used later to pair an allocation with the corresponding set of mapped deallocations.

7.2.2 Step 2: Event Flow Graph

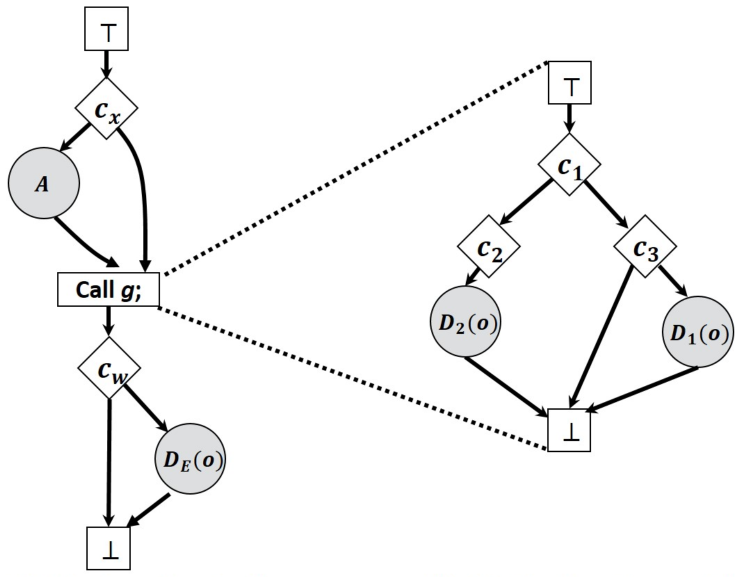
In this step, we prune the CFG to form equivalence classes of CFG paths. This is done by constructing for each CFG the corresponding EFG using the linear-time algorithm as described in Chapter 5. The initial step in our CFG pruning algorithm is marking the event nodes in the given CFG that are relevant to allocation/deallocation pairing analysis. Given the memory taint graph MTG_A for the allocation A and the CFG (CFG_f) for function $f \in \text{MTG}_A$, the events of interest for allocation/deallocation pairing are as follows: (1) the CFG nodes that correspond to the allocation A (`alloc_node`) and its corresponding deallocation nodes, and (2) the CFG nodes that correspond to callsites for functions in MTG_A. Next, $\mathcal{M}$-SAP takes as input: the CFG (CFG_f) and its marked event nodes and produces the corresponding EFG (EFG_f) using the EFG transformations defined in Section 5.6

7.2.3 Step 3: Allocation/Deallocation Pairing

$\mathcal{M}$-SAP iterates over the allocation instances to apply the pairing algorithm to each MTG_A to pair the allocation A with its corresponding deallocations, that reference the same allocated memory block. For efficiency, the pairing algorithm computes context-sensitive function summaries using the EFGs computed in the previous step. Note that, if a function appears in two memory taint graphs with corresponding allocations A_1 and A_2, then the function would have two contexts as well as two summaries. $\mathcal{M}$-SAP visits the memory taint graph in a bottom-up manner while: (1) computing compact function summaries for each visited function, plugging in the summaries of the callees at call-sites while analyzing the callers, and (2) keeping track of the allocation/deallocation pairs and unpaired allocations.

7.2.3.1 Compact Function Summaries

For each function, the function summary is computed by traversing its EFG in a depth-first manner while keeping track of all entry/exit allocations/deallocations on all EFG paths. Let us illustrate our approach to computing compact function summaries. Figures 7.3(a) and (b) show the EFG for functions f and g. In this example, f calls g at statement: `Call g;`. The function summary for g should retain the information relevant for analyzing f. The pairing analysis is concerned with what follows a given allocation: (i) an allocation followed by deallocation implies pairing, and (ii) an allocation not followed by deallocation implies an unpaired allocation (potential memory leak). Recall that we only have one allocation node (callsite) per memory taint graph.



(a) EFG for function *f* with respect to allocated memory (*o*)

(b) EFG for function *g* with respect to allocated memory (*o*)

$$\left\{\begin{array}{c}\text{Allocation: } A\\ \text{Deallocations: } D_E(o), D_1(o), D_2(o)\\ \text{THROUGH}\end{array}\right\} \quad \texttt{entry\_summary} \quad \left\{\begin{array}{c}\text{Deallocations: } D_1(o), D_2(o)\\ \text{THROUGH}\end{array}\right\}$$

$$\left\{\begin{array}{c}\text{Allocations: } A\\ \text{THROUGH}\end{array}\right\} \quad \texttt{exit\_summary} \quad \{\text{THROUGH}\}$$

(c) Function Summary for *f*

(d) Function Summary for *g*

Figure 7.3 Compact function summaries for caller (f) and callee (g)

The function summary consists of two sub-summaries: *entry* and *exit* summaries.

The entry summary for g summarizes: (i) the possible deallocations in g that can be paired with A, (ii) the entry allocation in g, and (iii) the possibility of not pairing A with deallocation in g. Case (iii) is possible if there exists a path in g that does not have allocations/deallocations.

The entry summary for g - denoted by `entry_summary` in Figure 7.3(d)- includes: the entry deallocations ($D_1(o)$ and $D_2(o)$), and the `THROUGH` state denoting the existence of paths in g that do not have any allocations/deallocations.

The exit summary for g summarizes: (a) the possible allocation in g that can be paired with the deallocation ($D_E(o)$) in f, and (b) the possibility that an allocation before calling g can be paired with a deallocation after calling g. Case (b) is possible if there is a `THROUGH` state in g. The exit summary for g - denoted by `exit_summary` in Figure 7.3(d) - includes: the `THROUGH` state.

Since the pairing algorithm is traversing the MTG_A in a bottom-up manner, the summary for g is available to compute the summary for f. Figure 7.3(c) shows the entry and exit summaries for f. Note that: (1) the entry summary for g is part of the entry summary of f because the entry allocation and deallocations in g can be the entry allocation and deallocations for f too, and (2) the exit summary of g is part of the exit summary of f as the exit allocation in g can be the exit allocation in f.

7.2.3.2 Pairing Algorithm

Listing 7.2 describes the pairing algorithm. It iterates over all allocation instances and takes as input: an allocation A, the memory taint graph MTG_A, the EFG for each function within MTG_A. Then, it outputs: the set of allocation/deallocation pairs and unpaired allocations (if any).

The pairing algorithm (Listing 7.2) starts by visiting the memory taint graph MTG_A in a bottom-up manner (lines 2-13). For each function in MTG_A, the algorithm retrieves the EFG (line 4), gets the entry/exit nodes (lines 5-6), and passes the entry node and its empty node summary to the function `traverse_efg` to start the EFG traversal in a depth-first manner (line 8). Upon the return of `traverse_efg` (lines 9-12), the function summary for `function` is computed as follows: the `entry_summary` is the same as the entry summary (`pre_sum`) for the EFG entry node (line 9), and the `exit_summary` is equivalent to the (`post_sum`) of the EFG exit node (line 10). This function summary is then stored in a global structure (`summaries`) for later use (line 11). Lines (12-13) check whether the current function is one of the roots in MTG_A and the `exit_summary` of its summary contains the allocation A. If so, it reports the allocation A in `exit_summary` as unpaired allocation (i.e., potential memory leak).

```
 1 main(MTG_A)
 2   functions ← reverse_topological_sort(MTG_A);
 3   for(each function in functions)
 4     efg ← get_EFG(function);
 5     entry_node ← get_entry_node(efg);
 6     exit_node ← get_exit_node(efg);
 7     node_summary ← {pre_sum: {}, post_sum: {}};
 8     traverse_efg(entry_node, node_summary);
 9     summary.entry_summary ← pre_sum for entry_node;
10     summary.exit_summary ← post_sum for exit_node;
11     summaries.put(function, summary);
12     if (function ∈ roots(MTG_A)) AND (summary.exit_summary contains allocation)
13       report the allocation A in summary.exit_summary as unpaired allocation;
14 end
15
16 traverse_efg(node, ns)
17   if the node is a call-site for function within MTG_A
18     sum ← summaries.get(called function by node);
19     ns.pre_sum ← sum.entry_summary;
20     ns.post_sum ← sum.exit_summary;
21   else if node is deallocation
22     update the pre_sum and post_sum of ns with this deallocation;
23
24   if node is visited before AND ns holds the same summary when the node previously
       visited
25       return ns.pre_sum;
26
27   pre_sum ← {};
28   for(each s in successors(node))
29     pre_sum += traverse_efg(s, ns);
30   if ns.post_summary contains allocation AND pre_summary contains deallocation
31     report allocation/deallocation pairing between the allocation in ns.post_summary and
           the deallocation(s) in pre_sum;
32   update ns.pre_sum with pre_sum;
33   return ns.pre_sum;
34 end
```

Listing 7.2 Pairing Algorithm

Lines (17-20) of function `traverse_efg` check if the current node (`node`) is a call-site for a function within MTG$_A$, then, the current node summary is updated with the summary of the called function. In case of the current visited node is a deallocation function call (lines 21-22), then the algorithm updates the current node summary with the current deallocation.

In our pairing algorithm, `traverse_efg` can visit an EFG node multiple times if new information that affects the allocation/deallocation pairing analysis is present. At lines (24-25), the algorithm checks whether the current node is visited before and it stops traversing through this node, if the current node summary is the same as the one when previously visited. In other words, if no new information is presented at this node, then no need to re-visit the node. Otherwise, the traversal continues to line 27. Lines (28-29) iterate through the successors of the current node and passes each successor to `traverse_efg` to recursively visit subsequent nodes. Upon the return of `traverse_efg`, the `pre_sum` is updated with the entry summary for each of its successors. Once iterating through the successors is completed (lines 30-31), the algorithm checks whether the `post_sum` of the current node contains an allocation and the `pre_sum` of the successors contains a deallocation, if that is the case: the algorithm reports allocation/deallocation pairing between the allocation A in `ns.post_sum` and the deallocations in `pre_sum` of successors. Then at line 32, the algorithm updates the `pre_sum` of the current node's with the `pre_sum` of successors. Finally, the updated `pre_sum` for the current node is returned (line 33).

7.2.4 Step 4: Feasibility Check for Potential-Error Paths

A path on which an allocation is not paired with a deallocation may or may not be an error depending on whether the path is feasible or not. Thus, checking feasibility of such paths is required to avoid false positives. Unlike existing approaches to static allocation/deallocation pairing, $\mathcal{M}$-SAP does not encode any information about path feasibility throughout the allocation/deallocation pairing algorithm. Instead, it only checks the feasibility of *potential-error paths*. These are the paths that have an allocation that is not paired with any deallocations. $\mathcal{M}$-SAP applies feasibility analysis to EFG paths instead of CFG paths. This is because the EFG contains fewer paths/conditions than its corresponding CFG. Currently, $\mathcal{M}$-SAP can perform intra-procedural feasibility analysis as follows:

Let the allocation A_p in function L be unpaired allocation, and p be the path that has A_p. Then, if p is an intra-procedural path, $\mathcal{M}$-SAP will check the feasibility of the EFG path p. If p is an inter-procedural path (i.e., across functions), then $\mathcal{M}$-SAP will only check the feasibility of the sub-path of p contained within L. We call the intra-procedural EFG path that will be checked for feasibility, a *potential-error path*.

To check the feasibility of a potential-error path, $\mathcal{M}$-SAP traverses this path and calculates a *path condition* which is the conjunction (AND) ($\wedge$) of the branch conditions along that path. These are the conditions that must be true for the path to be executed. Then, it translates the path condition to Boolean expressions by assigning a Boolean variable to each condition. To reveal the branch condition correlations in the Boolean expressions, $\mathcal{M}$-SAP uses textual equality. For example, let (`error` $\wedge$ `!error`) be a path condition for a problematic path. Then, by assigning each condition a boolean variable, the corresponding boolean expression will be: $c_1 \wedge c_2$

where (c_1 = `error`) and (c_2 = `!error`). Based on the textual equality between `error` in c_1 and c_2, $\mathcal{M}$-SAP can infer the correlation: $c_2 = !c_1$. Thus, the resulting boolean expression should be: $c_1 \wedge !c_1$. This final expression's satisfiability will be tested to determine the problematic path's feasibility.

In case of a condition along the potential-error path that tests any of the pointers pointing-to the allocated memory block for nullness, $\mathcal{M}$-SAP assigns `true` or `false` value to the Boolean variable associated with that condition in the Boolean expression of that path. Consider the example of the potential-error path q which goes through the `true` branch of the condition C(`ptr == NULL`) after allocating a memory block at `ptr = kmalloc(...)`. In this case, $\mathcal{M}$-SAP assigns that Boolean variable associated with C a `false` value in the Boolean expression of q so that q becomes infeasible when tested for feasibility.

Finally, $\mathcal{M}$-SAP uses BDDs [92] to check the satisfiability of the resultant boolean expressions for each potential-error path to determine its feasibility.

7.3 Enabling Evidence for Human-Machine Collaboration

In this section, we present the evidence produced by $\mathcal{M}$-SAP and the result categories that will facilitate the human reasoning to comprehend, cross-check, and validate the produced results.

7.3.1 Creating Verification Instances

The $\mathcal{M}$-SAP tool breaks the verification problem into verification instances. Each verification instance corresponds to an allocation call site. $\mathcal{M}$-SAP automatically verifies as many instances as possible with a strong inherent guarantee of correctness. The $\mathcal{M}$-SAP verification results fall into three categories:

- $\mathcal{C}1$: the automatically verified instances with no violation.
- $\mathcal{C}2$: the automatically verified instances with one or more violations shown by missing deallocation(s) on feasible path(s).
- $\mathcal{C}3$: the remaining instances where the verification is inconclusive.

7.3.2 Instance Verification Kit (IVK)

Verification instances are bundled into an IVK. The IVKs are designed as a vehicle to integrate automation and human intelligence to solve the verification problem. The IVKs are produced automatically and they provide verification-critical, compact evidence for the human analyst to reason with. Each IVK includes: (a) the source location of the allocation call, (b) the source locations of the paired deallocation calls and the points-to graph for each deallocated pointer, and (d) the visual models MTG, PtG, CFG, and EFG.

7.4 Empirical Evaluation & Results

In this section, we discuss allocation/deallocation pairing analysis results obtained by evaluating $\mathcal{M}$-SAP on a recent version of the Linux kernel of more than 12 million lines of complex multi-threaded C code. To evaluate $\mathcal{M}$-SAP, we have considered three evaluation criteria: (1) competitiveness against the current state-of-the-art and top-rated Linux kernel device driver verification tool (LDV) [5], (2) analysis speed, and (3) pairing accuracy. Finally, we present examples of verification instances from the three result categories ($\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$) to give a qualitative sense of how the automatically generated evidence simplifies the verification task. Our experiments were done on a 2.70 GHz Intel Xeon CPU machine with 128 GB memory, running Ubuntu Linux 14.

7.4.1 Implementation and Experimental Setup

We implemented $\mathcal{M}$-SAP using Atlas from EnSoft [3]. Atlas is a platform to develop program comprehension, analysis, and validation tools. Both Atlas platform and $\mathcal{M}$-SAP are Eclipse plugins. Atlas first compiles the given source code and creates a graph database of relationships between program artifacts. Then, one can interactively comprehend and/or analyze the source code using Atlas's interpreter or by writing Java programs to analyze the source code using Atlas APIs. For more information about Atlas, refer to [37] and EnSoft's website [3].

$\mathcal{M}$-SAP leverages the Atlas APIs to implement the allocation/deallocation mapping algorithm (Listing 7.1), the algorithm to transform a CFG into an EFG (Section 5.6), the allocation/deallocation pairing algorithm (Listing 7.2), and the feasibility analysis (Section 7.2.4).

We used $\mathcal{M}$-SAP to analyze the recent versions (3.17-rc1) of the Linux kernel. We enabled all possible `x86` build configurations via `allmodconfig` flag. In Table 7.1, columns `LOC`, `Functions`, `Build`, `Nodes`, `Edges`, and `Time` show the corresponding numbers for lines of code, functions, build time, nodes and edges in the graph database pre-computed by Atlas and the time for this pre-computation.

Table 7.1 Linux Kernel (v 3.17-rc1) Artifacts

LOC	Functions	Build	Index (Graph Database)		
			Nodes	Edges	Time
12.3 M	571,012	15m 12s	43.1 M	133.4 M	2h 14m

$\mathcal{M}$-SAP is applied to pair `kmalloc` allocation function call with `kfree` deallocation function call which both widely used in the Linux kernel.

7.4.2 $\mathcal{M}$-SAP: Experimental Results

We compare $\mathcal{M}$-SAP against the Linux Driver Verification tool (LDV) [5] which is the current top-rated tool in the software verification competition (SV-COMP) [18–20] in the Linux device drivers verification category. The LDV's developers were generous to provide us with the LDV's results on the same versions of the Linux kernel and the same build configurations we have used.

Table 7.2 shows the comparison of $\mathcal{M}$-SAP and LDV. Columns `kmalloc` and `kfree` show the numbers for allocation/deallocation callsites. Note that an allocation instance may be paired with multiple deallocations on different execution paths. Columns `C1`, `C2` and `C3` show the numbers of allocation instances that fall in each of the three results categories $\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$, respectively.

$\mathcal{M}$-SAP can accurately pair 92.3% of the allocations. After manually examining $\mathcal{C}1$ cases for imprecision assessment, we found that $\mathcal{M}$-SAP does not produce any false negatives and can

kmalloc	kfree	$\mathcal{M}$-SAP				LDV			
		$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Analysis Time	$\mathcal{C}1$	$\mathcal{C}2$	$\mathcal{C}3$	Analysis Time
1,149	2,203	1,060 (92.3%)	50	39	1h 21m	357 (31.1%)	0	792	30h 44m

Table 7.2 Allocation/deallocation pairing results on Linux kernel version (3.17-rc1)

precisely pair all the cases in $\mathcal{C}1$ column. The remaining 89 allocations ($\mathcal{C}2$ and $\mathcal{C}3$ categories) are verified manually. We have discovered 50 cases in $\mathcal{C}2$. These cases amount to actual memory leaks that have been reported to the Linux community[2]. The results in category $\mathcal{C}3$ correspond to instances where $\mathcal{M}$-SAP gives inconclusive answers. In Section 7.4.4, we discuss the analysis limitation factors behind these cases.

In the case of LDV: there are 357 (31.1%) automatically paired allocations, however, there is no guarantee of correctness. LDV does not report any instances in category $\mathcal{C}2$, i.e. it misses the 50 memory leaks found by $\mathcal{M}$-SAP. Of the 1,149 allocations, LDV fails to handle 792 allocations: 454 due to false positives and 338 due to scalability (LDV crashes). By contrast, $\mathcal{M}$-SAP does not have scalability issue, it correctly handles all but (7.7%) of the allocations. We also checked that $\mathcal{M}$-SAP subsumes all correct parings done by LDV, and every allocation unpaired by $\mathcal{M}$-SAP is either unpaired or not handled by LDV. Thus, $\mathcal{M}$-SAP is strictly and significantly more accurate than LDV.

Column `Analysis Time` denotes the total time needed for each analysis. $\mathcal{M}$-SAP takes 81 minutes for completing the analysis of while LDV takes 31 hours.

7.4.3 Case Studies & Qualitative Assessment of Visual Models

In this section, we present examples of verification instances from the three result categories ($\mathcal{C}1$, $\mathcal{C}2$ and $\mathcal{C}3$) to give a qualitative sense of how the automatically generated evidence simplifies the verification task. The visual models are meant to produce evidence to reduce the burden on the human analyst. Optimizing the visual models is crucial for reducing the human effort.

[2]Some of the reported memory leaks have been accepted and fixed. Others are still valid in the latest branch of the Linux kernel.

7.4.3.1 An Example from $\mathcal{C}1$ Category

This section presents an example to show the manual effort it would take an analyst to cross-check an instance that $\mathcal{M}$-SAP verifies automatically and reports no violation. This example belongs to category $\mathcal{C}1$ which has 92.3% instances. Figure 7.4 shows the visual models for an allocation that $\mathcal{M}$-SAP has reported to be correctly matched. Figure 7.4(a) shows the MTG for the allocation in function `lpfcdiag_loop_get_xri`. Figure 7.4(b) shows the PtG for pointer `dmabuf` after the allocation node and before the condition C_2. Figures 7.4(c) and 7.4(d) show the CFG and EFG for the function `lpfcdiag_loop_get_xri`, respectively.

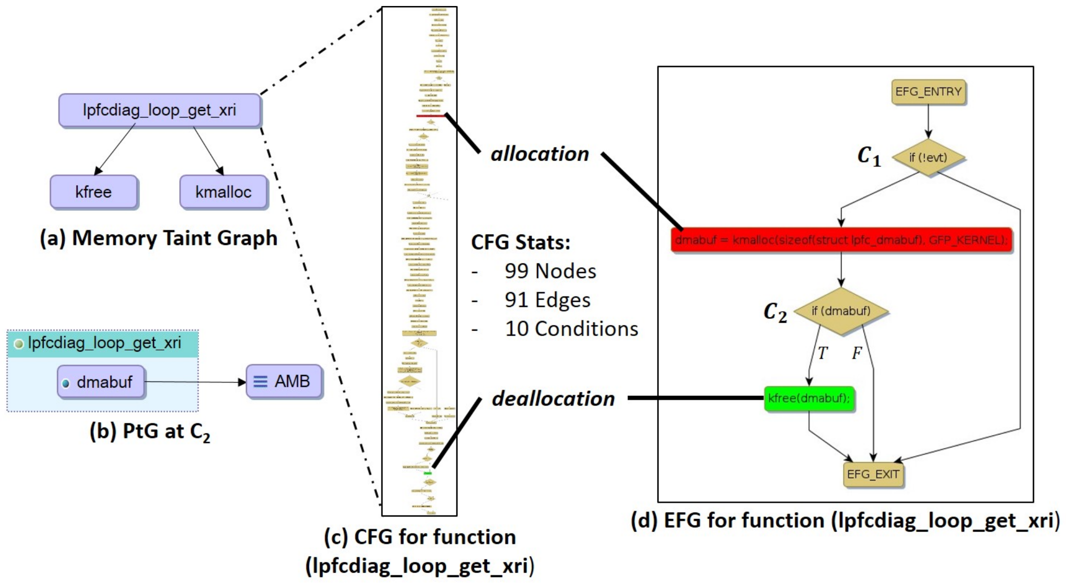


Figure 7.4 Visual models for an automatically verified instance

In this example, it is easy to observe from the EFG of function `lpfcdiag_loop_get_xri` that the allocation is followed by a branch node with two paths: (1) one path leads to a matching deallocation, and (2) the other path leads to the terminal node (potential-error path). The analyst can query the PtG for pointer `dmabuf` to cross-check whether the `dmabuf` points-to `null` (Figure 7.4(b)). In this example, $\mathcal{M}$-SAP successfully concludes that the potential-error path is infeasible because it corresponds to the `false` branch of condition C_2 which tests whether the pointer (`dmabuf`) that references the allocated memory block points-to `null`. This evidence simplifies the cross-check to conclude that the automatic verification is correct. This simplification is also apparent by the occurred reduction in nodes, edges, and conditions by going from the CFG to EFG. The CFG for function `lpfcdiag_loop_get_xri` has 99 nodes, 91 edges, and 10 conditions compared to 6 nodes, 7 edges and 2 conditions in the corresponding EFG.

Bugs Discovery. Below, we present two bug examples. The first example belongs to category $\mathcal{C}2$ where $\mathcal{M}$-SAP correctly reports a bug. The second example shows an inconclusive case from category $\mathcal{C}3$ where the evidence produced combined with interactive reasoning shows a bug.

7.4.3.2 Bug Example 1

Figure 7.5 shows the visual models for a discovered bug. This bug was discovered automatically by $\mathcal{M}$-SAP and then cross-checked manually. Figure 7.5(a) shows the MTG for the allocation in function `acm_probe`. Figure 7.5(b) shows the PtG for pointer `acm->country_codes` after the allocation node and before the condition C_1. Figures 7.5(c) and 7.5(d) show the CFG and EFG for the function `acm_probe`, respectively.

The EFG for `acm_probe` shows two paths on which the allocation in not followed by a deallocation: (1) $P_1 = C_1$, and (2) $P_2 = \overline{C_1}C_2C_3$.

As seen from the EFG, the path P_1 is infeasible because the `true` branch of C_1 is taken only if the pointer `acm->country_codes` points-to `null`, however, `acm->country_codes` points-to the allocated memory block as can be seen from the PtG (Figure 7.5(b)) after the allocation node. For path P_2, the path is feasible if the boolean expression $\overline{C_1}C_2C_3$ is true. To complete the verification, one must verify that the boolean expression is satisfiable and concludes that the automatically reported violation is indeed a violation. This bug was reported to the Linux organization and it got fixed in a later version.

7.4.3.3 Bug Example 2

This bug belongs to category $\mathcal{C}3$ where the $\mathcal{M}$-SAP tool gives inconclusive answer. Figure 7.6 shows the visual models for a discovered bug. Figure 7.6(a) shows the MTG for the

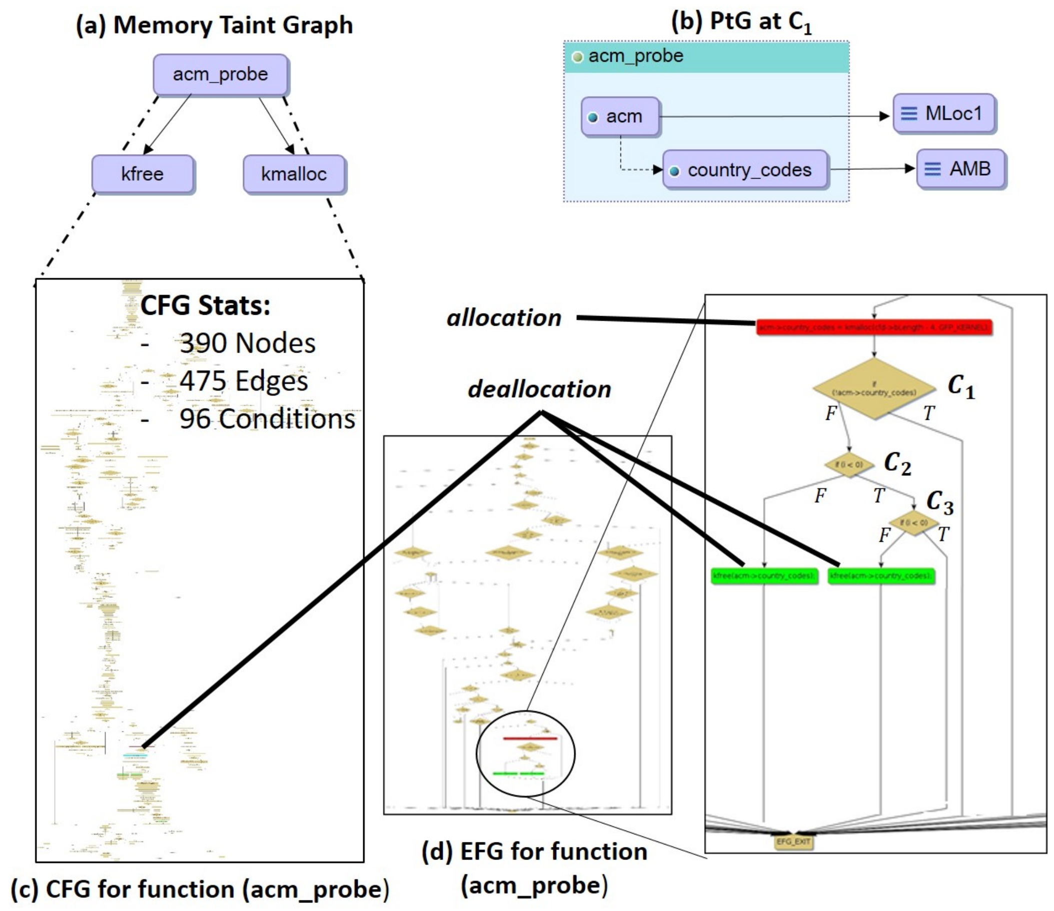


Figure 7.5 A bug discovery using visual models

allocation in function `wil_write_file_wmi`. Figures 7.6(b) and 7.6(c) show the CFG and EFG for the function `wil_write_file_wmi`, respectively.

The EFG for `wil_write_file_wmi` shows two paths on which the allocation in not followed by a deallocation: (1) $P_1 = C_1$, and (2) $P_2 = \overline{C_1}C_2$. As seen from the EFG, $\mathcal{M}$-SAP successfully determines that path P_1 is infeasible because the `true` branch of C_1 is taken only if the pointer `wmi` points-to `null`, however, `wmi` points-to the allocated memory block by the allocation node. On path P_2, the $\mathcal{M}$-SAP inconclusively reports a memory leak due to its inability to determine the feasibility of P_2. For path P_2, the path is feasible if the boolean expression $\overline{C_1}C_2$ is true. To complete the verification, one must verify that the boolean expression is satisfiable. The analyst checks the CFG for function `wil_write_file_wmi` and sees that C_2 is dependent on the value returned from the function call `simple_write_to_buffer`. Investigating

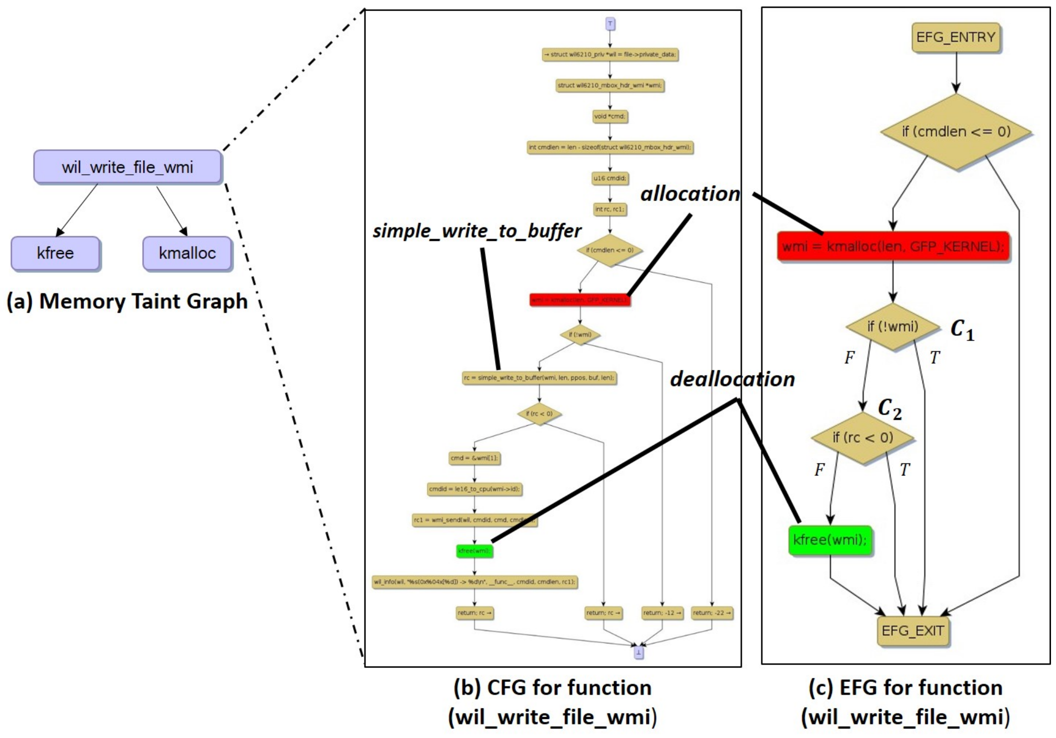


Figure 7.6 A bug discovery using visual models

through `simple_write_to_buffer`, the human analyst is able to conclude that path P_2 is feasible and this inconclusive allocation instance results on a memory leak. This bug was reported to the Linux organization and it got fixed in a later version.

7.4.4 Current Limitations of $\mathcal{M}$-SAP

The results in category $\mathcal{C}3$ correspond to instances where $\mathcal{M}$-SAP gives inconclusive answers for 39 (3%) instances. This percentage of reported unpaired allocations is attributed to limitations in $\mathcal{M}$-SAP's automatic verification due to the following limitation factors:

- **26 Instances:** Not being able to recognize infeasibility of paths in some cases due to: (a) the use of textual equality is not advanced enough to find complex correlations between branch conditions, and (b) the lack of inter-procedural feasibility analysis.

- **13 Instances:** Not being able to handle complex syntax including arrays, array's subscripts and double pointers. In addition to unsound handling of pointer arithmetic by treating, for example, an occurrence of `x + e` as an occurrence of `x`.

The following example from category $\mathcal{C}3$ shows the case where $\mathcal{M}$-SAP reports a violation that turns out to be a verification flaw due to analysis limitation. In this example, $\mathcal{M}$-SAP reports a memory leak because the allocation is not followed by a deallocation on some path. In this example, $\mathcal{M}$-SAP is not able to decide on the feasibility of the potential-error path. In reality, the potential-error path is infeasible but $\mathcal{L}$-SAP lacks the required capability to test this case for feasibility.

Figure 7.7 shows the EFG for function `rndis_query_oid`. The EFG shows two paths where the allocation is not followed by deallocation: (1) $P_1 = \overline{C_1}$ and (2) $P_2 = \overline{C_1 C_2}$. As seen from the EFG, the path P_1 is infeasible because the `true` branch of C_1 is taken only if the pointer `u.buf` points-to `null`, however, `u.buf` points-to the allocated memory block as can be seen from the PtG (Figure 7.7(b)) after the allocation node. For path P_2, the path is feasible if the boolean expression $\overline{C_1 C_2}$ is true. To complete the verification, one must verify that the boolean expression is satisfiable. In reality, this path P_2 is infeasible.

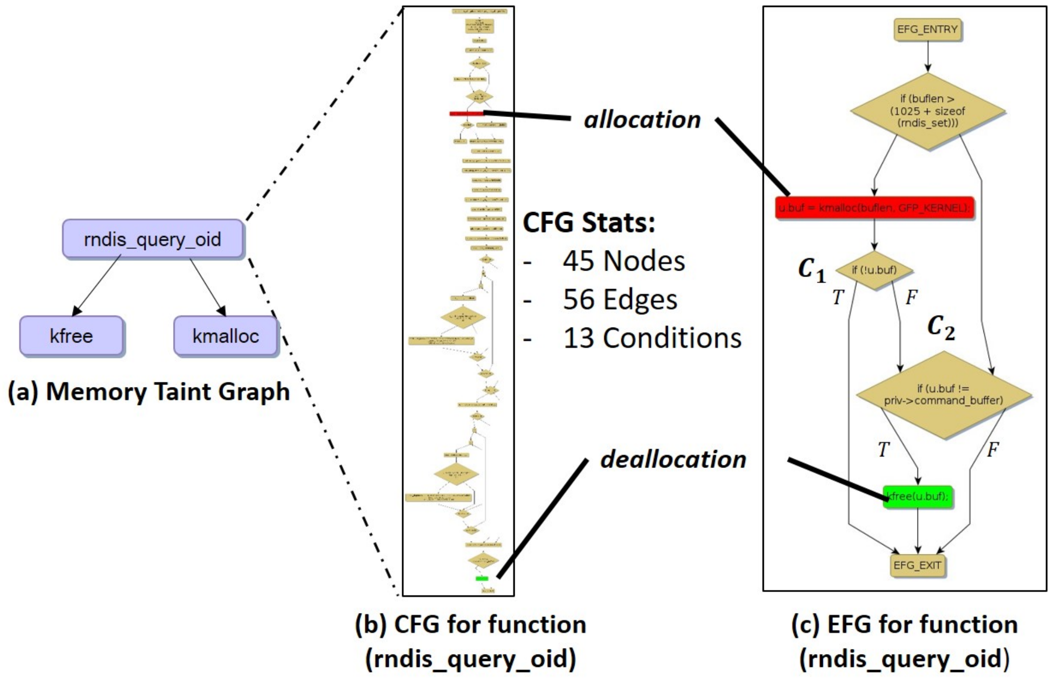

rndis_query_oid
kfree
kmalloc
(a) Memory Taint Graph
allocation
deallocation
CFG Stats:
- 45 Nodes
- 56 Edges
- 13 Conditions
(b) CFG for function (rndis_query_oid)
EFG_ENTRY
if (buflen > (1025 + sizeof (rndis_set)))
u.buf = kmalloc(buflen, GFP_KERNEL)
C_1
if (!u.buf)
T
F
C_2
if (u.buf != priv->command_buffer)
T
F
kfree(u.buf);
EFG_EXIT
(c) EFG for function (rndis_query_oid)

Figure 7.7 An Example from $\mathcal{C}3$ category

7.4.5 Off Limits Allocation Instances

In the Linux kernel (v3.17-rc1), there are $1,814$ `kmalloc` function calls (allocation instances) more than what is reported in Table 7.2. These allocation instances correspond to the cases which are out of the scope of our proposed pairing analysis. $\mathcal{M}$-SAP is not intended to handle these cases as it lacks the following capabilities:

- **725 Instances:** Inability to process function pointers. $\mathcal{M}$-SAP cannot track the interprocedural cases in which a reference to the allocated memory block is passed as a parameter to a function called via a function pointer.
- **1,366 Instances:** Inability to perform backward pointer (data flow) tracking.
- **650 Instances:** Inability to process escapes through global structures and lists.
- **392 Instances:** The current analysis is not offset-sensitive. In the Linux kernel, there are cases where a field of a structure can be accessed indirectly by adding/subtracting an offset from another field within the same structure.

7.5 Conclusions

$\mathcal{M}$-SAP is a static tool that uses a novel scalable and accurate allocation/deallocation pairing analysis. It uses algorithmic innovations based on a study of observed difficulties for allocation/deallocation pairing in the Linux kernel. $\mathcal{M}$-SAP applies a pointer analysis, that leverages the points-to information directly from source code (not an intermediate representation of source code). The analysis is field-sensitive (by distinguishing different fields in a `struct`), flow-sensitive (by tracking flow of statements), and object-sensitive (by distinguishing different allocation sites as different memory blocks). We evaluated $\mathcal{M}$-SAP on a recent version of the Linux kernel. The evaluation results show major accuracy and scalability improvements over the currently top-rated Linux kernel device driver verification tool (LDV) [5]. $\mathcal{M}$-SAP is able to accurately pair 92.3% of the total analyzed allocations with their corresponding deallocations in one hour with no false negatives. The analysis using $\mathcal{M}$-SAP has led to the discovery of 50 memory leaks. For each pairing of an allocation with corresponding deallocations, $\mathcal{M}$-SAP produces evidence which includes the memory taint graph (MTG) of the set of functions for inter-procedural analysis and the call chains between them, the event flow graphs and a compact summary for each of the functions in MTG, and the PtG to assist with data flow analysis. This evidence makes it easy for the human analyst to cross-check the results produced by $\mathcal{M}$-SAP, or to complete the analysis manually for the cases where $\mathcal{M}$-SAP reports potential-error paths but cannot provide conclusive results.

CHAPTER 8. CONCLUSIONS AND FUTURE DIRECTIONS

This thesis presents a new approach to solve hard software verification problems of large software. These problems that have remained intractable despite the many attempts to solve them completely automatically. The new approach is not about pushing the boundaries of complete automation. It is about fusing automation and human reasoning to solve hard software verification problems. Verification-critical and compact evidence to empower human reasoning as well as efficient automation is the key element of this new approach called Evidence-Enabled Verification (EEV).

This thesis presents EEV with two challenging applications: (1) EEV for Lock/Unlock Pairing and its implementation in $\mathcal{L}$-SAP tool to verify the correct pairing of mutex lock and spin lock with their corresponding unlocks on all feasible execution paths, and (2) EEV for Allocation/Deallocation Pairing and its implementation in $\mathcal{M}$-SAP tool to verify the correct pairing of memory allocation with its corresponding deallocations on all feasible execution paths. The tools incorporate algorithmic innovations to advance the state of the art for scalable and accurate verification. The tools partitions the verification problem into verification instances. Each verification instance corresponds to a lock call site in case of $\mathcal{L}$-SAP and an allocation site in case of $\mathcal{M}$-SAP. Both tools produce three categories of results: ($\mathcal{C}1$) automatically verified instances with correct pairings, ($\mathcal{C}2$) automatically verified instances with violations, and ($\mathcal{C}3$) instances where the automated verification is inconclusive. Each instance is accompanied with visual verification-critical evidence to simplify cross-checking for verification instances in $\mathcal{C}1$ and $\mathcal{C}2$ and to complete verification for verification instances in $\mathcal{C}3$.

We applied the $\mathcal{L}$-SAP tool to verify three recent versions of the Linux kernel with altogether 66,609 verification instances. $\mathcal{L}$-SAP is fast and scalable, and it is able to accurately pair 99.3% of the total locks in 3 hours. Our analysis discovered 8 synchronization bugs that were reported to the Linux community and accepted by them. Compared with the LDV tool [5], $\mathcal{L}$-SAP reduces by 49$\times$ the number of statically unpaired locks and by 59$\times$ the analysis time. We applied the $\mathcal{M}$-SAP tool to verify a recent version of the Linux kernel with $2,963$ verification instances. $\mathcal{M}$-SAP is fast and scalable, and it is able to accurately pair 92.3% of the total analyzed allocations in one hour with no false negatives. Our analysis discovered 50 memory leaks that are reported to the Linux community. Compared with the LDV tool [5], $\mathcal{M}$-SAP reduces by 9$\times$ the number of statically unpaired allocations and by 30$\times$ the analysis time.

We believe that the extreme accuracy and scalability of EEV stem from the visual models that leverage the intrinsic regularity developers build into large software to make its complexity manageable for them. Developers may not document their good design, but it is intrinsically there to extract using mathematically rigorous abstractions and use them to make software verification a practical reality for large software. This is a research direction worth pursuing. Another worth exploring direction is how to harness the evidence generated using such abstractions for targeted testing.

BIBLIOGRAPHY

[1] Clang static analyzer. http://clang-analyzer.llvm.org/.

[2] Coverity static analysis. http://www.coverity.com.

[3] Ensoft corp. http://www.ensoftcorp.com.

[4] L-sap: Scalable and accurate lock/unlock pairing analysis for the linux kernel. http://home.engineering.iastate.edu/~atamrawi/l-sap.

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

[6] Linux results. http://kcsl.ece.iastate.edu/linux-results/.

[7] The llvm compiler. http://llvm.org/.

[8] Open64 compiler. http://www.open64.net/.

[9] Reported bugs by L-SAP. http://home.engineering.iastate.edu/~atamrawi/l-sap/bugs.html.

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

[11] Timesort. https://en.wikipedia.org/wiki/Timsort.

[12] XINU. http://en.wikipedia.org/wiki/XNU.

[13] S. B. Akers. Binary decision diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.

[14] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[15] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

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

[17] T. Ball and S. K. Rajamani. The s lam project: debugging system software via static analysis. In *ACM SIGPLAN Notices*, volume 37, pages 1–3. ACM, 2002.

[18] D. Beyer. Competition on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.

[19] D. Beyer. Second competition on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 594–609. Springer, 2013.

[20] D. Beyer. Status report on software verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 373–388. Springer, 2014.

[21] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

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

[23] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Software EngineeringESEC/FSE'97*, pages 361–377. Springer, 1997.

[24] D. Bruening and Q. Zhao. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223. IEEE Computer Society, 2011.

[25] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[26] C. Canal and A. Idani. *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, volume 8938. Springer, 2015.

[27] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309. ACM, 1998.

[28] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *ACM SIGPLAN Notices*, volume 42, pages 480–491. ACM, 2007.

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

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

[31] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Notices*, volume 37, pages 258–269. ACM, 2002.

[32] A. Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[33] J. Clause and A. Orso. Leakpoint: pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 515–524. ACM, 2010.

[34] C. Click. Global code motion/global value numbering. In *ACM SIGPLAN Notices*, volume 30, pages 246–257. ACM, 1995.

[35] M. Das. Unification-based pointer analysis with directional assignments. *Acm Sigplan Notices*, 35(5):35–46, 2000.

[36] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Notices*, volume 37, pages 57–68. ACM, 2002.

[37] 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*, pages 588–591. ACM, 2014.

[38] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. 1998.

[39] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[40] A. Dinning and E. Schonberg. *An empirical comparison of monitoring algorithms for access anomaly detection*, volume 25. ACM, 1990.

[41] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[42] A. Futoransky, D. Saura, and A. Waissbein. Timing attacks for recovering private entries from database engines.

[43] A. Galloway, G. Lüttgen, J. T. Mühlberg, and R. I. Siminiceanu. Model-checking the linux virtual file system. In *Verification, Model Checking, and Abstract Interpretation*, pages 74–88. Springer, 2009.

[44] B. Gates. Bill Gates Keynote: Microsoft Tech-Ed 2008, 2008.

[45] J. Gleick and R. C. Hilborn. Chaos, making a new science. *American Journal of Physics*, 56(11):1053–1054, 1988.

[46] P. Godefroid and S. K. Lahiri. From program to logic: An introduction. In *Tools for Practical Software Verification*, pages 31–44. Springer, 2012.

[47] 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*, pages 151–160. IEEE, 2010.

[48] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *Computer Aided Verification*, pages 412–417. Springer, 2011.

[49] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *ACM SIGPLAN Notices*, volume 49, pages 326–336. ACM, 2014.

[50] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. *ACM SIGPLAN Notices*, 49(6):337–348, 2014.

[51] K. Jayaram and P. Eugster. Program analysis for event-based distributed systems. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 113–124. ACM, 2011.

[52] Y. Jung and K. Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th international symposium on Memory management*, pages 131–140. ACM, 2008.

[53] A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, and A. Strakh. Towards an open framework for c verification tools benchmarking. In *Perspectives of Systems Informatics*, pages 179–192. Springer, 2012.

[54] A. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov. Establishing linux driver verification process. In *Perspectives of Systems Informatics*, pages 165–176. Springer, 2010.

[55] W. Le and M. L. Soffa. Generating analyses for detecting faults in path segments. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 320–330. ACM, 2011.

[56] P. Lee, F. Pfenning, and A. Platzer. Static single assignment.

[57] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/java user's manual. *ESC*, 2000:002, 2000.

[58] O. Lhoták. *Program analysis using binary decision diagrams.* PhD thesis, McGill University, 2006.

[59] E. N. Lorenz. Deterministic nonperiodic flow. *Journal of the atmospheric sciences*, 20(2):130–141, 1963.

[60] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In *Static Analysis*, Lecture Notes in Computer Science. 2002.

[61] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *ACM Sigplan Notices*, volume 44, pages 134–143. ACM, 2009.

[62] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 24–33. ACM, 1991.

[63] J. T. Mühlberg and G. Lüttgen. *Blasting linux code.* Springer, 2007.

[64] A. Navabi, N. Kidd, and S. Jagannathan. Path-sensitive analysis using edge strings. 2010.

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

[66] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[67] M. N. Ngo and H. B. K. Tan. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 215–224. ACM, 2007.

[68] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur. The yogi project: Software property checking via static analysis and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 178–181. Springer, 2009.

[69] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *ACM Sigplan Notices*, volume 44, pages 397–407. ACM, 2009.

[70] W. Penninckx, J. T. Mühlberg, J. Smans, B. Jacobs, and F. Piessens. Sound formal verification of linuxs usb bp keyboard driver. In *NASA Formal Methods*, pages 210–215. Springer, 2012.

[71] D. Perkovic and P. J. Keleher. Online data-race detection via coherency guarantees. In *OSDI*, volume 96, pages 47–57, 1996.

[72] H. Post, C. Sinz, and W. Küchlin. Towards automatic software model checking of thousands of linux modulesa case study with avinux. *Software Testing, Verification and Reliability*, 19(2):155–172, 2009.

[73] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[74] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011.

[75] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 110–121. IEEE, 2003.

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

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

[78] S. P. Reiss. Event-based performance analysis. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 74–83. IEEE, 2003.

[79] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[80] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[81] N. Sterling. Warlock-a static data race analysis tool. In *USENIx Winter*, pages 97–106, 1993.

[82] P. Stratis. Formal verification in large-scaled software: Worth to ponder, 2014.

[83] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *Software Engineering, IEEE Transactions on*, 40(2):107–122, 2014.

[84] Y. Sui, S. Ye, J. Xue, and P.-C. Yew. Spas: scalable path-sensitive pointer analysis on full-sparse ssa. In *Programming Languages and Systems*, pages 155–171. Springer, 2011.

[85] A. S. Tanenbaum and H. Bos. *Modern operating systems.* Prentice Hall Press, 2014.

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

[87] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[88] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155, 2009.

[89] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214. ACM, 2007.

[90] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[91] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Static Analysis*, pages 170–183. Springer, 2007.

[92] J. Whaley. Javabdd-java binary decision diagram library, 2010.

[93] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM.

[94] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 501–504. ACM, 2007.

[95] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[96] X. Xie, J. Xue, and J. Zhang. Acculock: Accurate and efficient detection of data races. *Software: Practice and Experience*, 43(5):543–576, 2013.

[97] Y. Xie and A. Aiken. Context-and path-sensitive memory leak detection. *ACM SIGSOFT Software Engineering Notes*, 30(5):115–125, 2005.

[98] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16, 2007.

[99] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[100] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: helping programmers narrow down causes of memory leaks. *ACM SIGPLAN Notices*, 46(6):270–282, 2011.

[101] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):17, 2013.

[102] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *ICSE*, 2004.