Human-centric verification for software safety and security

by

Payas Rajendra Awadhutkar

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Software Systems)

Program of Study Committee:
Suraj Kothari, Co-major Professor
Yong Guan, Co-major Professor
Srikanta Tirthapura
Wei Le
Chinmay Hegde

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2020

DEDICATION

I would like to dedicate this work to my family, friends, mentors, and colleagues. I would not be where I am without constant support from all of you. I have been privileged to have all of you in my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I want to thank my family - my mother, father, and sister. When I was young, my mother said to me "It does not matter who you turn out to be, always strive to be the best". Those words inspired me and set me on a path of lifelong learning. They have been highly influential in shaping my thinking process.

Without the mentors, I could not have refined my thinking process. In retrospect, I feel especially fortunate in meeting my mentor Dr. Suresh Kothari. Mathematics was my favorite subject during school years, but over the years I was restricted to being a guy who likes mathematics and my mathematical intuition stagnated. I was introduced to Dr. Kothari by my professor Dr. Pradeep Waychal. It was a chance meeting but it changed my life. Without him I would have stayed "an engineer who likes mathematics" but now I can say I am "an engineer who uses mathematics". I feel fortunate to have met him and I will be forever grateful to Dr. Waychal for introducing me to him.

This journey would not have been the same without the support of my friend Sneha Bhansali. Apart from research, I have been passionate about teaching, and Sneha has always been my sounding board. I firmly believe that you cannot claim to have mastered a concept unless you can teach it. Discussions with Sneha definitely helped me become a better teacher and in turn improved my ability to do research.

I would like to acknowledge that this work would not have been possible without my amazing colleagues, both inside and outside of Iowa State University. I was privileged to have worked with them. In particular, this work would not have been possible without the discussions and technical support of Suresh Kothari, Benjamin Holland, Ahmed Tamrawi, Ganesh Ram Santhanam, Jon Mathews, Nikhil Ranade, and Jeremías Sauceda. I also acknowledge the Defense Advanced Research Projects Agency (DARPA), which helped fund this research. I must also acknowledge that:

"This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.".

Ph.D. is perhaps the longest marathon I will ever run. Just like any other Marathon runner, I could not have done it without all the support. Thank you all for believing in me!

ABSTRACT

Software forms a critical part of our lives today. Verifying software to avoid violations of safety and security properties is a necessary task. It is also imperative to have an assurance that the verification process was correct. We propose a human-centric approach to software verification. This involves enabling human-machine collaboration to detect vulnerabilities and to prove the correctness of the verification.

We discuss two classes of vulnerabilities. The first class is Algorithmic Complexity Vulnerabilities (ACV). ACVs are a class of software security vulnerabilities that cause denial-of-service attacks. The description of an ACV is not known a priori. The problem is equivalent to searching for a needle in the haystack when we don't know what the needle looks like. We present a novel approach to detect ACVs in web applications. We present a case study audit from DARPA's Space/Time Analysis for Cybersecurity (STAC) program to illustrate our approach.

The second class of vulnerabilities is Memory Leaks. Although the description of the Memory Leak (ML) problem is known, a proof of the correctness of the verification is needed to establish trust in the results. We present an approach inspired by the works of Alan Perlis to compute evidence of the verification which can be scrutinized by a human to prove the correctness of the verification. We present a novel abstraction, the Evidence Graph, that succinctly captures the verification evidence and show how to compute the evidence. We evaluate our approach against ML instances in the Linux kernel and report improvement over the state-of-the-art results. We also present two case studies to illustrate how the Evidence Graph can be used to prove the correctness of the verification.

CHAPTER 1. OVERVIEW

1.1 Research Theme

Software is ubiquitous in the modern world. Even mission-critical and safety-critical applications are increasingly dependent on software. As demonstrated by Ariane 5 disaster [1], software failures can prove extremely costly. More recently, a software glitch caused all of Boeing 737 max aircrafts to be grounded [2]. Thus, it is imperative to ensure that the software does not violate any safety or security properties.

Completely automated verification of every software safety and security property is not possible while completely manual verification of software is not feasible. Even when a completely automated verifier can be developed for a specific problem, how does one know that the verification is correct? There needs to be a *proof* of the verification that provides assurance that the result is indeed correct.

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

The theme of our research has been *Human-Machine Collaboration*. We were inspired by the writings of Frederick Brooks, who writes [3]: *"If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself."*. Our work has been about building IA systems to enable human-machine collaboration for verifying software.

1.2 Organization

This thesis is divided into two parts.

1. *Description of the vulnerability is not known a priori:* The first part discusses our work in the context of a class of software security vulnerabilities called Algorithmic Complexity Vulnerabilities (ACV). Often, the description of the ACV is not known a priori. Finding ACVs in large software is thus equivalent to searching for a needle in the haystack when you don't know what the needle looks like. Thus, first, the vulnerability needs to be hypothesized and then the software needs to be searched for the hypothesized vulnerability. We describe an approach to detect ACVs developed as part of the DARPA STAC project [4]. This discussion spans Chapters 2-4.

2. *Description of the vulnerability is known a priori:* Even when the description of the vulnerability is known, i.e. we know what to search for, verifying software to assure there is no vulnerability is a hard problem in general. Proving correctness of the verification is also a daunting task. In Chapter 5 we discuss a visionary paper by Alan Perlis [5] and argue for the need for human scrutiny of the verification proofs. We provide an example to support our motivation. In Chapter 6 we discuss the challenges involved in constructing such verification proofs. In Chapter 7 we discuss an approach to compute verification proof artifacts and a schema to present it to a human for further scrutiny. In Chapter 8 we present results from an empirical study of the Linux kernel.

CHAPTER 2. DETECTING ALGORITHMIC COMPLEXITY VULNERABILITIES

In this chapter, we discuss a class of cybersecurity vulnerabilities known as Algorithmic Complexity Vulnerabilities. DARPA identified this class of vulnerabilities as a rising threat and launched the Space/Time Analysis for Cybersecurity (STAC) program [4] to develop novel tools that can detect the presence of Algorithmic Complexity Vulnerabilities. Chapters 3 and Chapter 4 discuss the research performed on detecting Algorithmic Complexity Vulnerabilities on the DARPA STAC program.

2.1 Introduction

Algorithmic Complexity Vulnerabilities (ACV) are a class of vulnerabilities that can be exploited to mount a denial of service (DoS) attack. [6]. MITRE describes the effect as "An algorithm in a product has an inefficient worst-case computational complexity that may be detrimental to system performance and can be triggered by an attacker, typically using crafted manipulations that ensure that the worst case is being reached." [7]. In contrast with traditional DoS attacks, which involve flooding the target server with redundant inputs to block a legitimate request, ACVs allow DoS attacks with very few requests or a small input. In other words, attacker performs far less amount of work compared to their target.

Let's illustrate an ACV with an example. We will use the example of a *billion laughs attack* on an XML parser [8]. A standard XML file contains a "contents section" with instances of predefined entities. The entities are of the form `<!ENTITY name "value">` where name is the variable name for the entity and value is its definition. An entity can refer to another entity using an ampersand (&) i.e. the value for that entity would be of the form `&entityname`. When an XML parser encounters such entities while parsing the contents section, it will replace the value with the definition of the

referenced entity and continue parsing. If the entity expansion is left unchecked then an attacker can easily specify a huge XML document using a small number of entities by repetitively referencing the entity definitions. Listing 2.1 shows a standard exploit [8]. It uses 10 different XML entities (`lol-lol9`) where `lol` is defined as "lol". All other entities are defined as 10 of some other entity. The document contents only one instance of `lol9`. The parser will expand it to 10 `lol8s`, each of which is expanded to 10 `lol7s` and so on.

Listing 2.1 XML-bomb

```
1  <?xml version="1.0"?>
2  <!DOCTYPE lolz [
3   <!ENTITY lol "lol">
4   <!ELEMENT lolz (#PCDATA)>
5   <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
6   <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
7   <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
8   <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
9   <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
10  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
11  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
12  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
13  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
14 ]>
15 <lolz>&lol9;</lolz>
```

Listing 2.2 shows how a standard XML parser is used. On line 17, the `parse` method is invoked to parse the XML file and serialize it. The serialization can expand an entity a large number of times. For example, in the case of billion laughs, the entity `lol9` eventually expands a billion times, thus earning the moniker billion laughs. Naturally, there needs to be an expansion limit. This limit is not set by default and if the developers don't set the limit then the parser keeps expanding the XML entities unconditionally. In that case, by the time the parser is finished parsing the billion laugh XML, it has created a document of over 3 GB. In other words, if left unchecked the XML

expansion consumes an exponential amount of resources in the worst-case. Note that, the input file is disproportionately small in size, barely 100 bytes. If the application using the XML parser is not handling this worst-case then it is said to contain an ACV which can be exploited to deny memory resources to benign users.

Listing 2.2 "Standard XML Parser"

```
// Standard usage of XML parser
// If the expansion limit is not set,
// then the code is vulnerable to xml bomb.

int main (int argc, char* argv[]) {
  // Initializations

  // Critical: Expansion limit
  // If this is not set then XML bomb can be triggered
  // sm.setEntityExpansionLimit(100);

  const char* xmlFile = argv[1];
  try {
    // The file can include arbitrary number of expansions
    // Without a expansion limit, the parser will expand all
    // This can lead to creation of a gigantic file on disk
    parser.parse(xmlFile);
  } catch (...) {
  }
  return 0;
}
```

An example would be Microsoft Word, which uses an XML-like parser to load its documents. If a user attempts to open a word document containing the code in Listing 2.1, then Word will attempt to load the expanded document in the memory, and in most cases will hang [9].

In recent years, exploits using ACVs are on the rise. Crosby and Wallch [6] coined the term ACV in 2003 and theorized an attack on hash tables. In 2011, Klink and Walde [10] demonstrated the attack and noted that it plagues hash table implementations in almost all the widely used libraries. This attack was further refined and demonstrated by Bernstein *et al.* a year later [11]. It is imperative that we find ways to mitigate the risks posed by ACVs and hence develop technology to detect ACVs.

2.2 DARPA STAC Engagements

This motivated the DARPA Space/Time Analysis for Cybersecurity (STAC) program [4]. It called for a novel human-on-the-loop approach to detect ACVs as completely automated detection of ACVs is intractable [4].

DARPA STAC program was structured like a Cyber Defense Competition. There were three kinds of research teams. BLUE teams were tasked with developing tools to detect vulnerabilities in given software. The RED team was tasked with creating challenge apps in order to evaluate the tools developed in the STAC program. The WHITE team was tasked with evaluating the performance of BLUE teams using the apps developed by the RED team.

Let's first shed some light on these challenge apps. DARPA contracted security professionals (RED team) to develop apps containing vulnerabilities based on real-world software. These apps are fairly large and the vulnerable code is hardened against detection techniques by obfuscating the code. Each app comes with a description of the overt functionality and a YES/NO question about a vulnerability and analysts are tasked with answering that question. This includes the type of resource consumption (space or time), the threshold for resource consumption, and the constraints on input size. In order to be considered a valid ACV, the detected ACV by a tool must exceed the threshold while staying within the input constraints. Most of these apps are already publicly available on GitHub [12].

We participated in the program as a BLUE team. The apps were divided into 8 sets and each set on average contained 30 apps. The analysis process of each set was called as an *engagement*.

The time allotted for every engagement was gradually shortened (starting from 2 months to less than one week in the end), while the number of apps to analyze was increased. We emerged as one of the top-performing teams on the program, eventually achieving 100% accuracy. In Chapter 3 we discuss our approach to detecting ACVs and in Chapter 4 we discuss the tool we developed and present a case study of ACV detection.

2.3 Related Work

We discuss related work in three categories: static analysis for extracting high-level semantic patterns from loops, model-based formal verification for extracting loop invariants and classifying loops, and tools to analyze domain-specific loops.

Semantic patterns: Existing work on extracting semantic loop patterns [13, 14, 15] can classify loops based on a high-level specification of their semantics, e.g., whether a loop involves a search, selection, or initialization of a data structure. While these patterns are useful in general for understanding the high-level functionality, they cannot filter out loops reachable from particular user input, or classify loops based on their dependence on local and inter-procedural data, which are critical aspects to hypothesizing ACVs.

Formal Verification, Symbolic Analysis and Loop Summarization: Formal Verification approaches to derive loop invariants or estimate loop iteration counts [16, 17, 18, 19] have been used for several applications including WCET (worst-case execution time). Many techniques have been proposed to summarize the loop effect [20, 21, 22, 23] using symbolic analysis. LESE [24] aims to compute iteration count precisely using symbolic execution and use it to infer the loop effect. The technique in [22] generates pre- and post-conditions as loop summaries using dynamic symbolic execution. These techniques focus on single-path loops. Proteus [20] classifies the complexity of the loop execution for multi-path loops into four types based on conditions on the paths. All the above techniques rely on the presence of an induction variable in the loop. In contrast, our analyses characterize loops with respect to any variable that affects termination (which subsumes induction variables), and apply to loops with single or multiple paths.

Program Abstractions: Program Dependence Graphs (PDG) [25, 26] have been traditionally used to abstract data and control dependences of a program. The information in the Termination Dependence Graph (TDG) and Loop Projected Control Graph (LPCG) abstractions collectively contain all the information in the PDG that is relevant to the termination of a loop [27]. The TDG uses a kind of taint analysis for reaching definitions [28] to track the flow of information from various sources to the termination condition (sink). This differs from the traditional taint analysis used for PDG construction in that our taint analysis also tracks flows from callsites to their return values to capture potential dependencies through callsites to the termination conditions. In addition, the TDG classifies the inter-procedural dependencies based on the source type. This information can help alert analysts about the additional analysis that needs to be done to audit the loop for ACVs. The LPCG differs from the PDG in the control dependence aspect in that it elides equivalent paths from the loop header to the termination conditions.

Domain-specific analysis: The most closely related work to ours is the analysis of Android loops by CLAPP [29]. CLAPP analyzes loops to extract information about the operations that influence and control the number of iterations of a loop, as well as operations that constitute the loop's body. For the security aspect, CLAPP identifies loops with calls to a set of high-risk APIs (which can be captured using subsystems interactions in our approach), loops whose iterations depend on certain external data sources such as network data, and potentially infinite loops (which can be inferred using the number of terminating conditions from our loop catalog). However, like all the other approaches, CLAPP does not support interactive visualization of key aspects of a loop, which is crucial in helping analysts comprehend loops and hypothesize ACVs in them.

CHAPTER 3. LOOP CHARACTERIZATIONS FOR DETECTING ALGORITHMIC COMPLEXITY VULNERABILITIES

In this chapter we discuss our approach to detecting ACVs. It summarizes key concepts from our paper *Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities* by Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. It was published at the 24th Asia-Pacific Software Engineering Conference (APSEC 2017), Nanjing, China, December, 2017 [27]. Our approach to detect ACVs is a four-step approach: (a) automatically generate a loop catalog that identifies all loops and the characteristics of each loop. We introduce new concepts and patterns to define loop characteristics, (b) applying the knowledge gained from the loop catalog, the analyst can configure filters to select loops that the analyst wants to scrutinize, (c) the analyst uses the visual querying mechanism to scrutinize selected loops and the control flow paths containing the loops to gather evidence for ACVs, (d) based on the evidence, the analyst performs targeted dynamic analysis to confirm each ACV. The dynamic analysis and the tool for it are described in the paper [30].

An important part of our research has been to decipher relevant loop characteristics by studying publicly known examples of ACVs and the ACV challenges posed by DARPA. For example, an automatic analyzer can determine the bound to be 10 for the loop `for(i=0; i<10;i++) int arr = new int[Integer.MAX];`. However, not the bound but the large array allocation in each iteration is the loop characteristic relevant for ACVs. The relevant characteristics may be hidden in a method that is invoked within the loop. Thus, an inter-procedural analysis is necessary to compute loop characteristics. Moreover, the relevant characteristics may be just on one path, and not on the other paths within a loop. Thus, a path-sensitive analysis is also important to reason about ACVs. Furthermore, it is not enough to characterize a loop in isolation. It is important to characterize loops in the context of program artifacts that connect a loop to the rest of the program. For

example, since ACVs are triggered by attacker's input, it is important to characterize whether the termination of a loop can be controlled by user input.

In summary, our research contributions are:

- *Loop Abstractions*: These are the building blocks for characterizing loops, and they include: (a) a data flow abstraction for loops called *termination dependence graph*, (b) a control flow abstraction called *loop projected control graph*.
- *Loop Characterizations and Patterns*: These are derived using loop abstractions and applied to enable the analyst to create filters to select loops with high possibility of ACVs. Specifically, we have 24 loop patterns based on different termination characteristics.
- *Interactive Querying for Visual Loop Scrutiny*: These mechanisms, called *Smart Views*, enable the analyst to scrutinize selected loops and gather the evidence for ACVs. The analyst can compose powerful program analyses using Smart Views and a graphical query language.

3.1 Research Gap

We discuss the research gap that has motivated the research presented in this chapter.

The program artifacts that can lead to ACVs include loops, recursion, or resource-intensive library APIs [31]. In this research the focus is on loops. To assess the relevant loop characteristics, we studied loops with ACVs. We have curated 15 representative loop snippets from the challenge apps provided by DARPA. We have made these loops snippets available in a public repository [32].

As an experiment, we tried the state-of-the-art loop analysis tool Proteus [20] that received the 2016 Distinguished FSE Paper award. None of the 15 loops can be precisely summarized by Proteus. We then identified specific characteristics of what makes the curated loops complex. Using them as markers of complexity, we found that the loops in commonly cited benchmarks (e.g. SV-COMP [33]) lack the complexity that one encounters in detecting ACVs.

Our complexity markers can be summarized as: (a) loop termination depends on variables that are not *induction variables* [34], (b) loop termination logic involves inter-procedural dependencies,

(c) the complex connection between user input and loop termination, (d) the multitude of paths and the presence or the lack thereof of guards on these paths to prevent excessive resource consumption operations in the loop. Of the 15 ACV loops we have gathered from the DARPA challenge apps, all 15 loops have the complexity markers (a), (b), (c), and 11 loops have the marker (d).

As an alternative to precise analysis, we tried the use of smells to detect ACVs [35]. In line with the findings of [36], these smells tend to be either too specific (too many false negatives) or too generic (too many false positives).

3.2 Loop Abstractions

Loop abstractions capture and represent the essentials of loops and the connecting parts of the program that affect loop behaviors. One abstraction is to capture the loop termination behavior based on the data flow to the loop termination conditions. Another abstraction is to facilitate path-sensitive analysis of loop behaviors to identify a control flow path with "expensive computation" as a potential ACV.

Termination Dependence Graph (TDG): This is a data flow graph designed to capture: (a) the data sources that influence the loop termination, and how the termination depends on local or inter-procedural data flow, and (b) the modifications of the variables that affect the loop termination.

This abstraction serves as the foundation for developing loop behavior patterns, such that each pattern implies a specific mode in which the loop terminates. These patterns are discussed in Section 3.3.3.

The union of backward data flow slices from the termination conditions gives us all the data sources which can influence the termination conditions. However, this is not sufficient to reason about the loop termination behavior because it does not capture all the modifications of these data sources. So we also need to compute forward data flow slices starting from the data sources to capture the modifications of the variables that affect the loop termination.

The TDG is defined with respect to a loop's *termination conditions*, i.e., the branch conditions through which the loop can exit. The TDG for a loop is $S \cup F$ where (1) S includes the union of

all the intraprocedural backward data flow slices from the termination conditions of the loop, and (2) F includes the union of all the intraprocedural forward data flow slices from all the variable assignments in S. The S part gathers the data sources that influence the loop termination, and F the part gathers the modifications of the variables that affect the loop termination.

A summary of the inter-procedural data flow dependencies is computed along with the TDG and is included in the loop catalog. For example, the TDG of a loop in a method M whose termination depends on the size of a collection passed as a parameter to method M requires the data flow from all potential methods that call M. The data flow between the formal parameter and the termination condition is shown in the TDG. An analyst can use the parameter in the TDG as an input to an Atlas query to gather the inter-procedural data flow into the parameter.

In Section 3.3.3, we discuss the use of TDG in an empirical study to define the *Loop Termination Patterns* (LTPs). The LTP type of each loop is recorded in the loop catalog. The LTP type indicates the complexity of loop termination and it is an important metric to select complex loops that are more likely to have ACVs.

Loop Projected Control Graph (LPCG): This is designed as a compact representation of relevant control flow within the loop. There is an LPCG per loop.

The compaction is based on the *Projected Control Graph* (PCG), a notion introduced by Tamrawi *et al.* [37]. It is an optimal mathematical abstraction to address the roadblocks to path-sensitive analysis. The PCG is a projection of the CFG to retain only the execution behaviors relevant to a given problem and elide duplicate paths with identical execution behavior.

A mathematical definition of the PCG and an efficient algorithm to transform a CFG to a PCG are presented in [37]. We create an LPCG for each loop by applying the PCG algorithm to the CFG subgraph restricted to the loop. The PCG algorithm requires as input a subset of CFG nodes relevant to behaviors of interest. For deriving an LPCG we use the following nodes: (a) loop header node (entry point of the loop), (b) termination condition nodes, and (c) data flow and callsite nodes from the TDG. In the case of the LPCG, paths reaching the same termination condition are elided by the PCG algorithm unless the paths include different sets of nodes from the TDG.

The LPCG distills the distinct loop termination behaviors in a compact representation. It is especially useful when the number of *control flow graph* (CFG) paths is very large but many paths have identical termination behaviors. LPCG facilitates detection of ACVs by providing an efficient way to focus on distinct behaviors. LPCGs are also useful to isolate paths with respect to the loop functionality (e.g., network, IO, crypto, etc.). To do so, the calls to subsystems are used as relevant nodes for creating the LPCG.

3.3 Loop Characterizations

Loop characterizations are computed automatically and used to create filters to select loops with high possibility of ACVs. We have used for this study 25 loop characterizations organized by categories such as conformance to loop termination patterns, monotonicity, loop control variable attributes, data flow to the loop control variables, the control flow paths inside a loop, and interactions with the subsystems

3.3.1 Loop Control Variable Attributes

A Loop Control Variable (LCV) is a variable that influences any termination condition of a loop. LCVs subsume induction variables [34], which are defined as variables whose modification in every iteration can be expressed as a loop invariant expression. Our definition of LCVs is particularly important for detecting ACVs. For example, consider a variable influencing the termination of a loop passed as an argument to a method, where it is assigned to a value controlled by the attacker's input. Clearly, it is critical to reason about the variable's updates to detect a potential ACV. This variable would qualify as an LCV for the loop. However, it is not an induction variable.

We analyze LCVs over two dimensions: the type of the LCV, and the data flow dependencies of the LCV.

Type of Loop Control Variables: Knowledge of the type of an LCV for a loop can indicate what kinds of resource consumption may cause an ACV. For example, knowing that an LCV is of

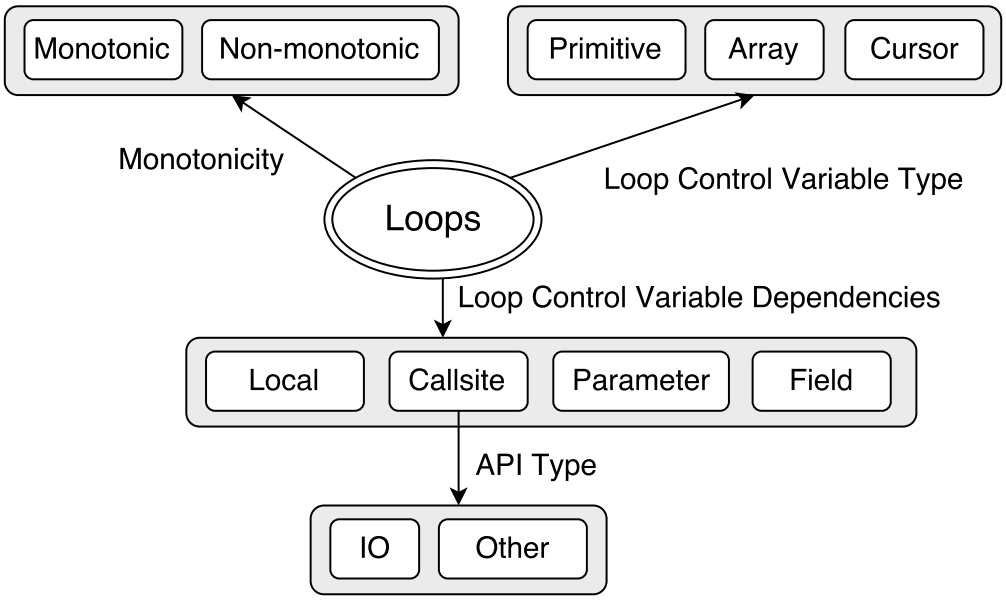


Figure 3.1 Loop attributes used for characterizations

array type, along with the fact the array size can be controlled by the attacker, raises the possibility of ACV involving excessive space consumption. We support the following control variable types:

- *Primitive:* Primitive variables are used to iterate over a range of numeric values. An example of such a loop's header is `for(i = 0; i < n; i++)`, where i is the loop control variable.
- *Array:* Array index variables are used to traverse an array. An example of such a loop header is `for(i = 0; i < length(array); i++)`.
- *Cursor:* Cursors are used to iterate over collections.

  `java.util.Iterator` and `java.util.Enumeration` are the most commonly employed cursors. Cursor APIs come in pairs, one which advances the cursor and other checks existence of a valid next cursor position. For example, loops invoke `Iterator.next()` in each iteration, which returns the current element at the cursor and advances the cursor. This is paired with a call to `Iterator.hasNext()` which checks for existence of a valid next cursor position prior to the next iteration. A similar iteration mechanism is provided by `java.util.Enumeration`'s `hasMoreElements` and `nextElement` APIs.

Data Flow Dependency of Loop Control Variables: In addition to the type, it is also important to know the data flow to the loop control variables. We will refer to this as the data dependence of loop control variables. We track intra-procedural (local) and inter-procedural (global) data dependence. The different forms of global dependence include object field, parameter, or return value from a callsite and these are recorded to assist with interactive analysis and cataloging. The information is particularly useful for detecting ACVs. For example, if the dependence for the size of the array is through a parameter, then passing a large array size value as a parameter creates the ACV possibility of excessive space consumption.

We define four levels of locality of dependencies for every loop control variable in a loop: 1) Local, 2) Callsite, 3) Parameter, 4) Field. We combine this information with the three types of the loop control variables to create a loop control variable attribute vector of size 12 for every loop to comprehend and catalog loops.

Loops whose termination conditions depend on the result of a callsite require an inter-procedural analysis to determine how the loop's termination may be influenced. Since loops invoking APIs in the `java.io` package, e.g., `while((line = file.readLine()) != null)`, are very common programming practices, we specifically identify such loops as belonging to the 'IO API' class. The rest of the loops with callsite loop control variables are classified as 'OTHER API'.

3.3.2 Loop Monotonicity

A monotonic loop is one in which all loop control variable updates go in one direction, either all increments or all decrements. In the simple example `for(i=0; i<10; i++) {...}`, all updates `i++` to the loop control variable `i` are increments. Operator and callsites update complexities can make it impractical to determine some cases of updates as *increment*, *decrement*; we classify them as *neither*. The salient points of how we handle the complex monotonicity cases are:

- We perform an analysis to detect the net effect of modifications of loop control variables. A monotonicity categorization of arithmetic operators on primitive loop control variables cannot simply be based on whether it performs addition, subtraction, or some such operation. For

example, an addition operation cannot always be considered an increment, it could be a decrement if the added number is negative.

- We classify callsite operators by partitioning the relevant APIs into *increment APIs* and *decrement APIs* respectively. For example, `Stack.push()` and `Stack.pop()` are treated as increment and decrement APIs respectively. We have developed a model of increment and decrement APIs that operate on commonly used collection types in the JDK.

We determine monotonicity as follows:

1. Mark all the operators on loop control variables as *increment* or *decrement* (or neither).
2. Mark a control flow path in $LPCG$ (beginning at the loop header and ending at a termination condition) as *monotonic* if includes only the *increment* or only the *decrement* operators. A path with operator of both kind or with a *neither* operator is marked as *non-monotonic*.
3. A loop is marked non-monotonic if has at least one non-monotonic path. Note that monotonicity does not mandate a monotonic operation in *every* iteration.

The above algorithm is sound, i.e., it correctly identifies monotonic loops. Its completeness cannot be guaranteed in cases such as when the LCV is passed as a parameter to a method or the loop invokes an API which we do not model.

3.3.3 Loop Termination Patterns

Loop Control Variable attributes when combined with loop monotonicity gives an idea of how a loop is going to behave. This enables us to define patterns of loop behaviors. We call these patterns Loop Termination Patterns (LTP).

LTP is defined as a triple (M, T, D), where $M \in \{\texttt{true}, \texttt{false}\}$ refers to the monotonicity of the loop, $T \in \{Primitive, Array, Collection\}$ refers to the type of the LCVs, and $D \in \{Local, Field, Parameter, Callsite\}$ refers to the dependency of the LCVs. This gives rise to 24

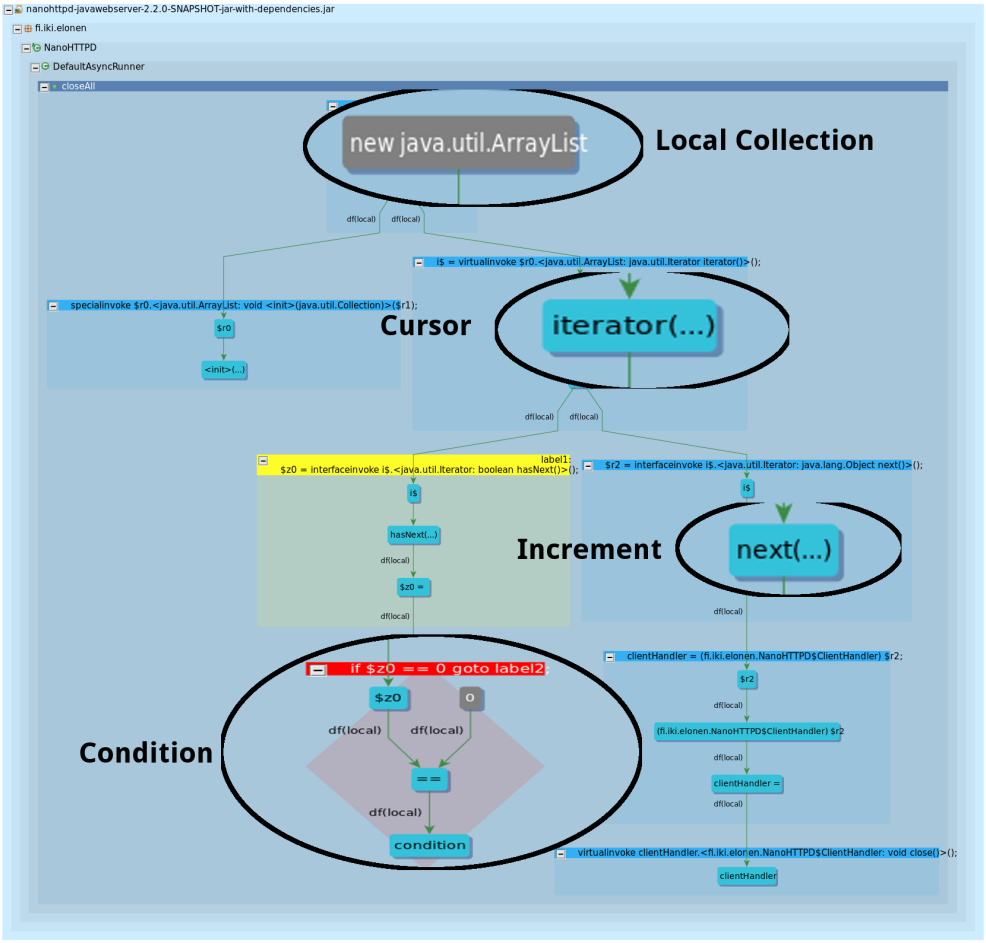


Figure 3.2 An example of a loop termination pattern and its key parts

LTPs. A loop may have match to more than one LTP, in which case we list all possible matches. As shown in our study in section 4.1 these 24 LTPs cover a significant portion of the real-world loops.

The 24 LTPs are divided in two groups as *simple* or *complex* termination patterns. Conformance to a simple LTP implies that no inter-procedural analysis is required to reason about the loop's termination. Figure 3.2 shows the TDG of a loop which conforms to a simple LTP. The highlighted parts of the TDG are captured in the LTP. Loops conforming to complex LTPs may require inter-procedural analysis, and additionally human judgement, in order to reason about the loop's termination. As shown in the case study in Chapter 4.3, the complex LTP can serve as a filter to find loops more likely to have ACVs.

Table 3.1 JDK Subsystems.

Subsystems	APIs belonging to this subsystem
JavaCore	java.util, java.lang
Hardware	javax.sound, javax.sound.midi
IO	java.nio, java.io
Network	java.net, javax.net, java.rmi
RMI	org.omg.CORBA, javax.rmi.CORBA
Database	javax.sql, javax.sql
Log	java.util.logging
Serialization	javax.xml.bind, javax.xml.ws.soap
Compression	java.util.jar, java.util.zip
UI	java.applet, java.awt, javax.swing
Introspection	java.lang.reflect, java.lang.invoke
Iterables	java.util.List, java.util.Vector etc.
Garbage Collection	java.lang.ref
Security	java.security, javax.security etc.
Crypto	javax.crypto
Math	java.math
Random	java.util.Random etc.
Threading	java.util.concurrent etc.
Data Structure	java.beans, java.text etc.

3.3.4 Subsystem Interactions

In addition to knowing how a loop terminates, it is important to have knowledge about the APIs invoked inside a loop's body in order to develop a vulnerability hypothesis. The analysts can deduce the high level functionality of the loop using the knowledge about the invoked APIs. We classify the APIs into 19 categories corresponding to JDK *subsystems*. Table 3.1 lists the subsystems and examples of packages they contain.

3.3.5 Automatic generation of loop catalog

We use the Atlas program analysis platform [38] to generate a queryable, directed multi-attributed program graph whose nodes and edges represent program artifacts and their relationships for a given application. We use an implementation of the DLI [39] algorithm to identify all loops in the application.

Next, we compute the properties related to loop termination (properties of TDG, LPCG, and LTPs) and operations in the loop body (subsystem interactions) for each loop in a given application.

The computed information is saved as a CSV file, which can be used by the analyst as a loop catalog. This includes monotonicity of a loop, the LTPs matched, size of the TDG and LPCG, number of callsites in loop body and their distribution among control flow paths, and subsystem interactions. The analyst can then apply filters with one or more criteria and/or rank the loops in order to select or eliminate loops.

The loop characteristics saved in the loop catalog are also saved as appropriate node and edge attributes in the Atlas program graph. This allows the analyst to filter and query specific loops and perform on-demand visual inspection of program artifacts and properties of loops selected through a query.

3.4 Visual Querying for Interactive Loop Analysis

Visual querying mechanisms aid interactive analysis of loops in order to hypothesize ACVs. We have designed two *Smart Views* for interactive visualization to scrutinize loops for ACVs. These Smart Views are based on the two loop abstractions described in Section 3.2. We have also created a filtering framework that enables custom selection of loops using specified constraints on loop characteristics.

3.4.1 Smart Views

A *Smart View* is designed to display and query a graph abstraction relevant to solving a particular problem. It is an interactive visualization mechanism, and offers the following: (a) a menu to select a type of software analysis to produce the graph abstraction, (b) invocation of the analysis by clicking on a source code object to which the analysis is applicable, (c) an interactive visualization of the analysis result. Atlas [38] comes with basic Smart Views for call graphs, control flow graphs, data flow graphs, etc., and provides APIs to create customized Smart Views.

Smart Views display graph abstractions and incorporate different ways to interact with those graphs: (1) a capability to zoom in and out, (2) a capability for incremental viewing of a graph, (3) several layouts (e.g. hierarchical, orthogonal) to display the graphical result of the analysis,

(4) search facility to look for a node or an edge by their name, (5) saving the graphical result as an image file for offline use, (6) two-way source correspondence between the elements of the displayed graph and the corresponding source code, (7) background colors and border styles to highlight the nodes and edges. Smart Views integrate seamlessly with other Smart Views to enable composition of analyses where one Smart View generates a graph that is used as an input for another Smart View. Additionally, Smart Views can be composed with the filtering framework (Section 3.4.2) and ad-hoc Atlas queries [38].

Termination Data Flow Smart View: Displays the Termination Dependence Graph (TDG) (Section 3.2) for a selected loop header. It enables incremental visualization of a large and complex TDG starting with the TDG roots. It provides color coding of nodes to ease comprehension: *Red* for the termination conditions, *Gray* for roots of the TDG, and *Green* for callsites that point to inter-procedural data flow that affect termination.

Use Case: It serves as evidence to scrutinize whether and how a loop terminates. Especially, it can save significant time and effort to comprehend inter-procedural dependence of a loop termination condition.

Loop Projected Control Flow Smart View: Displays the Loop Projected Control Graph (LPCG) (Section 3.2) for a selected loop header. It extends an LPCG by including control flow paths to callsites that may not affect the termination but could still create an ACV through a resource-intensive call. It provides the following color coding of nodes: *Yellow* for the selected loop header, various shades of *Blue* to display the loop body of nested loops with respect to their nesting depths (darker shades of Blue indicate loops nested deeper), *Red* for termination conditions (and *Cyan* for other branch conditions), *Green* for callsites, and *Magenta or Gray* respectively for the increment or decrement operators or API calls. The control flow edges are displayed as continuous lines, and *event flow edges* as dashed lines. As described in [37], the event flow edges as the induced edges to show the control flow reachability from one PCG node to another.

Use Case: It facilitates comprehension through display of paths that influence termination. With

a separate display it shows paths with callsites that could create an ACV with a resource-intensive call. It also helps scrutinize loops for monotonicity.

3.4.2 Loop Filters

We have developed a filtering framework to select loops matching a combination of the loop characteristics from the loop catalog. The framework currently supports the creation of custom filters by adding constraints on String, primitive and boolean properties. An example of a boolean property is monotonicity – a loop is either monotonic or not; and the two possible constraints based on this property would be "monotonic: true" and "monotonic: false". The nesting depth of a loop is an example of a primitive (integer) property. For example, the constraint "nesting-depth greater than 4" selects all loops having nesting depth of 5 or above within the method. A filter consists of a conjunction of constraints, i.e., a filter consisting of the above two constraints would select monotonic loops with nesting depth over 4. The filtering framework also allows analysts to fork a filter, i.e., create a new filter that includes a subset of the constraints added to an existing filter. This is useful for the analyst to explore multiple hypotheses related to ACVs in the application simultaneously.

We provide filters based on following six characteristics: 1) Reachability, 2) Subsystem Interaction, 3) Presence of branch conditions that affect the resource consumption of the loop, 4) LTPs, 5) Monotonicity, 6) Nesting Depth. We describe first three filters here. The filtering framework currently supports selection of loops, but is extensible and in the future could support selection of other artifacts such as methods or types based on their properties relevant to finding ACVs.

Reachability Filter: This filter selects the loops that are reachable from user input. It supports two boolean properties to enable selection of loops reachable from all the main methods (if supported) and loops reachable from web application handlers such as HTTP request handlers.

Use Case: For a loop to cause an ACV, the input provided by attacker must reach the loop to selectively trigger an execution path or influence its termination. This filter is useful to select the loops which an attacker can influence.

Subsystem Interaction Filter: This filter selects the loops that interact with a given subsystem (see Section 3.3.4). It supports a String property that specifies a subsystem and enables selection of loops which interact with the specified subsystem.

Use Case: Domain knowledge often provides insights to the analyst about how the APIs invoked in loops can cause ACVs. For example, thread creation within a loop indicates the possibility of an ACV due to exhaustion of stack memory. To select loops that may admit this possibility, the analyst may choose to apply this filter with the String property 'THREADING_SUBSYSTEM'.

Differential Branch Filter: This filter selects loops containing a *differential branch*, i.e., a branch condition that affects the loop's consumption of space or time. For example, a branch condition which determines the size of the array being allocated in a loop is a differential branch in the loop. Differential branches are interesting from two perspectives: 1) operations governed by the differential branch can potentially cause an ACV e.g., the branch governs file I/O operations, 2) the differential branch is governed by a parameter controlled by the attacker such as the size of a collection provided by the attacker. We support following kinds of differential branches relevant to ACVs: 1) branches that are governed by size of a collection, 2) branches governing operations that cause network interaction, 3) branches governing file I/O operations.

Use Case: Loops iterating over arrays or collections (e.g., sorting algorithms, matrix multiplication) may be potentially vulnerable to ACVs if the size of the array or collection can be controlled by the attacker. This filter is useful to find such loops when used in combination with reachability filter.

CHAPTER 4. A TOOLBOX TO DETECT ALGORITHMIC COMPLEXITY VULNERABILITIES

In this chapter we present DISCOVER, a toolbox to aid a human analyst in detecting algorithmic vulnerabilities built using the approach discussed in Chapter 3. This was published in our paper *DISCOVER: detecting algorithmic complexity vulnerabilities* by Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari at the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019), Tallinn, Estonia, August 2019 [40]. The chapter is organized as follows, we first present an empirical study to evaluate loop catalog, the cornerstone of DISCOVER, and then present a case study illustrating how DISCOVER helped detect a vulnerability in a DARPA STAC challenge app.

4.1 An Empirical Study with DARPA Challenge Apps and Open Source Software

The artifacts including loop abstractions, the loop classification, and the loop termination patterns are useful if they can provide a good coverage and facilitate understanding and selection of complex loops in real-world software. So, the purpose of this empirical study is to evaluate how well these artifacts serve as a lens to examine and understand loops in real-world software. The evaluation is done from different perspectives. The conformance of a loop to a Loop Termination Pattern (LTP) indicates that the loop terminates according to the pattern. It is then important to evaluate whether these patterns cover a large percentage of loops in real-world software. Abstractions create a compact representation to manage the complexity of a loop and its properties. So, as another perspective it is important to evaluate how significant is the simplification due to the abstraction when it is applied to real-world software.

This empirical study covers 18 challenge apps provided by DARPA and 4 open source libraries (Apache Commons IO (2.4), Apache Commons Collections (4.4.0), Apache Commons Lang3 (3.3.2), JGraphT (0.9.1)). Altogether, they include 5448 loops. While presenting the results, we refer to the challenge app loops as *C-loops* and the loops from the open-source libraries as *L-loops*. There are 3852 *C-loops* and 1596 *L-loops*.

4.1.1 Usefulness of Loop Termination Patterns (LTPs)

We evaluate usefulness of LTPs with respect to the following three questions about usefulness of LTPs: (1) What percentage of loops are covered by LTPs? A large coverage is desirable for LTPs to be considered useful in practice. (2) Among the loops covered by LTPs, what percentage of loops are complex cases for termination? LTPs are particularly useful if they can find complex cases of loop termination. (3) Among the complex loops covered by LTPs, what complexity characteristics are at play? LTPs can be insightful if they reveal the complexity characteristics and the dominance of those characteristics among the multitude of loops. We present results to answer these questions through an empirical study.

For the first question about the coverage of loops using LTPs, we observed that 73.31% of the C-loops and 88.30% of the L-loops are covered by LTPs. Thus, the study shows that LTPS are are useful to prove termination of a large portion of loops in the real-world software.

For the second question about the complex cases of loops, many different factors contributing to the complexity can be studied. We present here the results for a predominant complexity factor. It is especially time-consuming and error-prone for the analyst to examine and understand the cases that require inter-procedural analysis. The inter-procedural analysis can be due to multiple issues such as loops with nesting that goes across functions, or due to an inter-procedural data flow from input to the loop termination condition. Among the loops covered by LTPs, 86.06% of the C-loops and 89.44% of the L-loops require inter-procedural analysis. Thus, the study shows the LTPs are particularly useful because they are useful to prove complex cases of loop termination.

Table 4.1 C-loops and L-loops w.r.t. number of termination conditions (TC)

#TC	**median**	**90^{th} %ile**	**99^{th} %ile**	**max.**
c-loops	1	2	6	14
l-loops	1	2	6	16

For the third question about complexity characteristics handled by the LTPs, we present results for different attributes of the inter-procedural dependencies handled by the LTPs. The dependencies could involve a field of an object, a parameter of a method, or a return value at a callsite. Among the C-loops covered by LTPs with inter-procedural dependencies, 16.81% loops have dependencies through a field of an object, 3.19% loops have dependencies through a method parameter, and 80% have dependencies through a return value at a callsite. The corresponding percentages for the the L-loops are respectively 11.62%, 5.67% and 82.71%.

Unlike the dependencies through a field of an object, dependencies through a return value at a callsite are more challenging because they require an examination of the multitude of control flow paths through the methods invoked at those callsites. Thus, LTPs are significantly useful; not only do they find a large percentage of difficult loop termination cases, but also the cases that are quite complex because of the need for inter-procedural analysis and the path analysis.

4.1.2 Usefulness of Loop Characterizations

Loop characterizations are helpful in bringing out various complexities associated with a loop. We studied distribution of the loops with respect to following five characteristics in order to evaluate the usefulness of loop characterization. 1) loop monotonicity, 2) number of callsites in a loop, 3) number of termination conditions of a loop, 4) number of control flow paths in a loop, 5) cyclomatic complexity of a loop. Collectively, these characterizations help the analyst to isolate complex loops for further scrutiny.

Let us discuss empirical study results that show usefulness of loop characterizations.

Loop Monotonicity: Monotonicity is an indicator of loop complexity, not being monotonic in general reflects that it is hard to show that the loop terminates. The results show that 64.3%

Table 4.2 C-loops and L-loops w.r.t. number of paths in the loop body.

#paths	median	90^{th} %ile	99^{th} %ile	max.
c-loops	1	6	72.7	229432
l-loops	1	3	21	324

C-loops and 55.1% *L-loops* are monotonic. This indicates that monotonic loops are more prevalent than non-monotonic loops and hence can be used as a effective filtering mechanism.

Number of Callsites in Loops: The presence of multiple callsites makes it difficult to reason about loop behavior. The results show that 18.4% *C-loops* and 30.8% *L-loops* do not contain any callsites. This indicates inter-procedural analysis is needed to reason about a majority of the loops.

Number of Terminating Conditions: The number of terminations conditions can be useful as loops with significantly large number of termination conditions are generally difficult to reason about. The Table 4.1 shows the median, 90^{th} percentile, 99^{th} percentile, and the maximum with respect to the number of termination conditions per loop. For both the *C-loops* and *L-loops*, 90% of loops have at most two termination conditions and 99% of loops have six or fewer termination conditions. However, a few loops have a large number of termination conditions. One *C-loop* has 14 termination conditions, and one *L-loop* has 16 termination conditions.

Number of Control Flow Paths: A large number of control paths indicate a large number of different behaviors, which makes it difficult to reason about a loop's termination. The Table 4.2 shows the median, 90^{th} percentile, 99^{th} percentile, and the maximum with respect to the number of control flow paths per loop. For both the *C-loops* and *L-loops*, 50% of loops have one path. The 90^{th} and 99^{th} percentile values are considerably higher 6 and 72 for the *C-loops* compared to 3 and 21 for the *L-loops*. One *C-loop* has 229432 paths, and one *L-loop* has 324 paths. Thus, the *C-loop* collection has a significantly large percentage of loops with a large number of control flow paths compared to the *L-loop* collection.

Table 4.3 C-loops and L-loops w.r.t. cyclomatic complexity.

Cyclomatic	**median**	**90^{th} %ile**	**99^{th} %ile**	**max.**
c-loops	3	14	38	59
l-loops	3	7	16.5	30

Cyclomatic Complexity: As a comparison to our other loop complexity measures we study the cyclomatic complexity [41], which is a quantitative measure of the number of linearly independent paths through a program's source code, computed using the control flow graph. The cyclomatic complexity distribution shown in Table 4.3 is computed by using the control flow graph for each loop. The path metric in Table4.2 is also computed using the same set of graphs. The cyclomatic complexity is an approximation of our path metric.

4.1.3 Usefulness of Abstractions

To detect ACVs, the analyst must comprehend complex loop behaviors including loop termination, whether the termination is affected by input, the multitude of control flow paths with behaviors relevant to ACVs, and the paths with differential behaviors. Abstractions are intended to create compact representations of loops with the goal of simplifying the task of comprehending loop behaviors and enabling efficient automated analysis to characterize loops. The usefulness of these abstractions should be measured by the compactness they can achieve while maintaining the information essential to these behaviors.

The abstractions capture and represent the behavior information through graphs. Using $V + E$, as the size of a graph where V and E are respectively the number of nodes and the number of edges, we measure the compactness by comparing the graph size for the original program graph compared to the abstraction graph. We use the TDG the size of the data flow graph from which the TDG is derived, and we use for LPCG the size of the control flow graph from which the LPCG is derived. The results computed as averages are as follows:

- For each *C-loop* the original data flow graph is 4.3 times bigger than the TDG. For each *L-loop* the original data flow graph is 2.3 times bigger than the TDG.

- For each *C-loop* the original control flow graph is 2.8 times bigger than the LPCG. For each *L-loop* the original control flow graph is 1.5 times bigger than the LPCG.

The significantly higher reductions for the *C-loops* indicate that the developers of the challenge apps may have introduced artificial complexity to make it difficult to locate the vulnerable code segments. However, the abstractions can remove the irrelevant complexity.

We studied the correlation between the number of termination conditions of a loop and the percentage compaction achieved by the TDG and LPCG. The correlation coefficient between the reduction achieved by the TDG and the number of terminating conditions is 0.04 and the correlation coefficient between the reduction achieved by the LPCG and the number of termination conditions is -0.07. Thus, the compaction achieved by these abstractions has very little correlation to the number of termination conditions. Similarly, we found that the compaction has very little correlation to the number of paths. This is expected because compaction depends on the number of relevant nodes and that number is not correlated to either the number of termination conditions or the number of paths.

4.2 DISCOVER Workflow

Detection of ACVs using DISCOVER can be described in three phases:

1. Automated Loop Characterization: In the first phase, the analyst runs the automated loop analysis which characterizes every loop in the app using several pre-defined characterizations. These characterizations are computed using two loop abstractions: Termination Dependence Graph (TDG) and Loop Projected Control Graph (LPCG). The output of this phase is a *Loop Catalog* with their characterization. The details of this phase are described in our previous work [27].

2. Automated Filtering of Loops: ACVs are typically rooted in loops as loops are often the limiting factor of the computational complexity of the program. The Loop Catalog is designed to select

loops more likely to contain an ACV. The analyst combines the information captured by the catalog with a high-level understanding of the app to narrow down the possibilities of ACVs.

3. Interactive Audit of filtered loops: Analyst then makes use of the interactive capabilities of DISCOVER to audit the filtered loops and hypothesizes the presence of ACV, if any. This hypothesis can be checked using dynamic analysis techniques. With this workflow, our team was ranked to have the most accurate analysis on the final two competitive evaluations of the STAC program.

4.3 Case study of ACV Detection

We present a case study to illustrate how to detect ACVs using DISCOVER. We will be using an app called Gabfeed_3, which was developed as a challenge for the STAC program. The source code of the app, along with other challenge apps, is available on GitHub [12]. Gabfeed_3 is a web forum software which allows users to post messages on a server and search the posted messages. The messages are stored in sorted order. The server uses a custom merge sort for sorting messages on the backend. We received bytecode for the app (not the source code), which we converted to Jimple, an intermediate representation of Java bytecode. We use the Jimple code for analysis. Gabfeed_3 consists of $23,882$ lines of Jimple.

Background: DARPA created several challenge apps in order to evaluate the tools developed in the STAC program. Let's first shed some light on these challenge apps. DARPA contracted security professionals to develop apps containing vulnerabilities based on real-world software. These apps are fairly large and the vulnerable code is hardened against detection techniques by obfuscating the code. Each app comes with a description of a vulnerability and analysts are tasked with detecting vulnerabilities that match the given description. This description includes the type of resource consumption (space or time), the threshold for resource consumption, and the constraints on input size. In order to be considered a valid ACV, the detected ACV by a tool must exceed the threshold while staying within the input constraints. Most of these apps are already publicly available on GitHub [12] and DARPA plans to release the remaining apps in the near future.

4.3.1 Phase 1: Generate Loop Catalog

DISCOVER automatically characterized every loop in the app and cataloged every loop in the Loop Catalog. Loop Catalog also includes the information captured by the loop characterizations. Gabfeed_3 consists of 112 loops.

4.3.2 Phase 2: Filtering Loops

The goal is to isolate a subset of loops that are likely to contain an ACV. This filtering is done using the information captured by the Loop Catalog and a high-level understanding of the app. In this case study, we employed the following sequence of criteria to filter loops.

- Reachable Loops: In order to trigger the vulnerability, the loop must be reachable from the Control Flow Entry points of the app. These entry points were identified based on the domain knowledge of the app. Additionally, the attacker input to these entry points must also reach the loop body. Using loop catalog, the analyst selects only those loops which are reachable from the input to the app. *# Loops Retained: 75/112*

- Network Interactions: Gabfeed_3 is a web application. Hence, the inputs provided to the app are processed using network APIs. Thus, the loop must make use of the network subsystem to process the input. Hence, the analyst selects only those loops which interact with the network subsystem. *# Loops Retained: 35/112*

- Loop Monotonicity: Monotonic Loops are loops with simple termination logic and are typically not likely to contain an ACV. Thus, the analyst selects only non-monotonic loops. *# Loops Retained: 14/112*

- Loop Termination Pattern: Loops whose termination is dependent on well-understood APIs have a well-understood upper bound and are not likely to contain ACVs. Gabfeed_3 has 5 such loops which are used to read from files using `readline(...)` API. Loop Catalog captures this information by identifying the loop termination pattern. Analyst focuses on the remaining loops and discards these 5 loops. *# Loops Retained: 9/112*

At this point, the analyst decides to interactively scrutinize these 9 loops.

4.3.3 Phase 3: Interactive Audit

Loop Catalog reveals that these 9 loops are neatly separated in three different components of Gabfeed_3, namely Sorter, HashMap, and TreeNode. The analyst used LPCG smart view to look at LPCGs of these 9 loops. The goal was to search for paths within these loops which may lead to asymmetric consumption of resources. We discovered that out of the 9 loops, the loop in the method `Sorter.changingSort(...)` has a peculiar LPCG. Its LPCG is shown in Figure 4.1. The LPCG reveals the presence of a differential branch. This branch (zoomed in Figure 4.1) creates asymmetry as it has only one path with a callsite. This callsite invokes the method `Sorter.mergeHelper(...)` which handles the merge operation of the sort. This conditional merging is suspicious and further inspection reveals that there is also an unconditional merge before the suspicious callsite. Turns out, that if the number of messages is multiple 8 then this sorting algorithm merges every pair of sublists twice. The second merge is redundant and only adds to the cost of the sort. Thus, analyst hypothesized that if the attacker makes the number of posted messages multiple of 8, then the server is going to take a long time to sort the posted messages. This will increase the response time to any queries made for the posted messages by benign users. Using dynamic analysis, this hypothesis was proved.

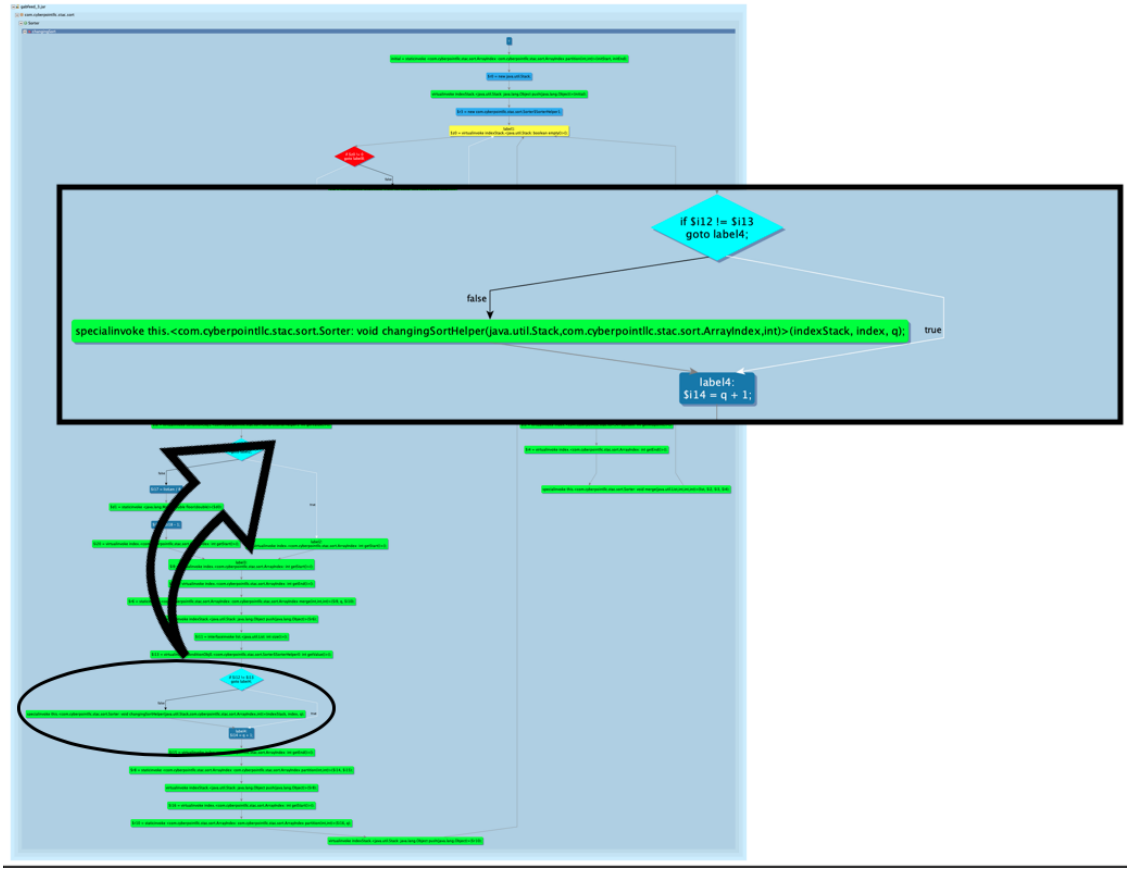


Figure 4.1 LPCG of vulnerable loop showing asymmetric behavior

CHAPTER 5. SOFTWARE ASSURANCE

5.1 Introduction

Software verification is an important but daunting task. The challenge stems from not just the complexity of modern software but also the sheer volume of it. For example, the Linux kernel which forms the backbone of most of the modern technology ranging from web servers, phones, desktops to mission critical components such as processors in aircrafts, is over 20 million lines of code (MLOC). Cyber-physical and embedded systems are becoming increasingly dependent on the software and vulnerabilities in the software can lead to catastrophic disasters. Hence, the ability to verify software efficiently and accurately has become critically important.

Software failures have caused serious accidents that resulted in death, injury, and large financial losses. Without intervention, the increasingly pervasive use of software may bring more frequent and more serious accidents. The National Academy of Sciences report on Software for Dependable Systems [42] points out that existing certification schemes that are intended to ensure the dependability of software have a mixed record; some are largely ineffective, and some are counterproductive. The National Academy Report [42] notes that conventional certification techniques based on conformance to development processes and testing cannot guarantee software compliance with critically important safety and security properties.

DO-178C [43, 44] and UK Def Stan 00-56 [45] are well-known certification standards. Both standards require the submission of a reasoned justification as to why and how the verification and validation techniques achieve the certification goals. Safety arguments are typically communicated in existing safety cases through free-form text. For example, a safety case in UK Def Stan 00-56 is defined as: *"A structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given operating environment."*

Unlike DO-178C, where testing evidence takes the front seat, UK Def Stan 00-56 gives precedence to analytical evidence. Nonetheless, in order to obtain credit for the use of mathematical analysis as the primary means for compliance, the mathematical methods should be accompanied with supporting evidence for reasoned justification. In practice, testing is the dominant technique for producing reasoned justification. All test cases are generated from the requirements. The quality of testing is judged against two coverage metrics: requirements coverage and structural coverage. While the first type of coverage determines which requirements were not tested, the latter determines how well the test cases exercised the code structure. The general approach that one must take in constructing an argument for replacing testing with another technology is to show how to argue that the evidence produced by the replacement technology is at least as convincing as the evidence produced by testing [46].

Security vulnerabilities can undermine the case made for dependability properties in software. The dependability properties must therefore account explicitly for security risks that might compromise its other aspects. It is also important to ensure that security certifications give meaningful assurance of resistance to attacks. Testing in general does not suffice for assuring software security, because even the largest test suites typically used do not exercise enough paths to provide evidence that the software is free from vulnerabilities. New security certification regimes are needed that can provide confidence that most attacks against certified products or systems will fail. For that, these regimes should be amenable to human scrutiny.

In this chapter, we discuss key ideas from our work *Insights for Practicing Engineers from a Formal Verification Study of the Linux Kernel* [47] by Suresh Kothari, Payas Awadhutkar, and Ahmed Tamrawi, published at Formal Verification for Practicing Engineers (FVPE) hosted at the 27th International Symposium on Software Reliability Engineering (ISSRE 2016), Ottawa, Canada, October 2016. These ideas served as the motivation for the work described in rest of the chapters. These ideas are inspired by the visionary paper: "Social Processes and Proofs of Theorems" by De Millo, Perlis (1st Turing Award, 1966) and Lipton [5].

5.2 A Visionary Paper

In 1979, Richard De Millo, Richard Lipton, and Alan Perlis published the visionary paper "Social Processes and Proofs of Theorems" [5]. The paper is about what is a *proof* and the purpose it serves in mathematics. The authors argue that formal verification programs do no play the same role as the proofs do in mathematics and discuss the qualities of a good proof. A brief summary of the central argument of the paper is as follows:

- Mechanisms that make engineering and mathematics really work are obscured in the fruitless search for perfect verifiability. In mathematics, the aim is to increase the human confidence in the correctness of a theorem. Nor does the proof settle the matter, contrary to what its name suggests, a proof is only one step in the direction of confidence. It is a social process that determines whether mathematicians feel confident about a theorem. Verification is nothing but a model of believability. It cannot be a model where proofs are accepted in blind faith. A proof should be amenable to human scrutiny.

- A good proof is one that makes us wiser. With formal verification, we know that our program is formally, correct. We do not know, however, to what extent it is reliable, dependable, safe; We do not know within what limits it will work; we do not know what happens when it exceeds those limits. The verification must provide knowledge that improves our practice.

Perfect verifiability is clearly not possible. The question is what would help to establish more trust in the correctness of formal verification. Because of the low-level at which formal verification operates, the formal proofs are extremely long and not amenable to human understanding. In mathematics, a proof serves the purpose of increasing a human's confidence in the correctness of a theorem. Perlis calls it "a social process that determines whether mathematicians feel confident about a theorem". Perlis argues for a social process to increase confidence in the verification results. That calls for verification proofs amenable to human scrutiny.

As the paper [5] notes, a good proof is one that makes us wiser. The paper makes distinction between formal verification proofs and the proofs used in mathematics and points to the intrinsic

problem that formal proofs do not make us wiser about the software. A good proof must provide knowledge to improve software engineering practice. It should help in determining the extent to which the verified program is *safe*. It should help in determining the limits of the program's reliability. It should help in determining what happens after those limits are exceeded.

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

5.3 A Motivational Example

We undertook a verification case study to get deep insights into the state-of-the-art for formal verification. We did the study using the Linux Driver Verification tool (LDV) [48] which has been the top Linux device driver verification tool in the software verification competition (SVCOMP)[33]. The LDV's developers were generous to help us with the study. We chose to study the problem of Lock/Unlock Pairing (LUP) for the Linux kernel. In this section, we discuss an example that clearly brings out the need to produce artifacts that enable human scrutiny of the verification process, even when the verification instance is reported as *safe*.

We present an example of a complex Linux verification instance where `Lock` is not correctly paired with `Unlock`, but LDV mistakenly verifies it as correct pairing. The PCG of a function shows that `Lock` is not followed by `Unlock`. We expected that LDV verifier would notice it and declare it unsafe. To our surprise, LDV has declared it a safe instance. We did a further investigation and found that it is an unsafe instance but not for the obvious reason. Since the formal verification proof is not revealed, it is not clear why LDV verified this instance incorrectly. This particular instance attracted our attention because of a peculiarity exhibited by its Control Flow. In this instance, the `lock` and `unlock` are on disjoint control flow paths in the function `drxk_gate_crtl` (`f1`) and if some condition C = `true`, the `lock` occurs, otherwise, the `unlock` occurs. We hypothesized that the `lock` and `unlock` can match if `f1` is called twice, first with C = `true` and then with C = `false`. A quick query using Atlas shows that `f1` is not called directly anywhere. Thus, it is either dead code or `f1` is called using a function pointer.

Resolving the function pointers using a tool we have developed using Atlas [49], we find the situation shown in Figure 5.1. The function `tuner_attach_tda18271` (`f2`) calls the function `f1` via function pointer. `demo_attach_drxk` sets the function pointer to `f1`, the pointer is communicated by parameter passing to `dvb_input_attach`, then to `f2`.

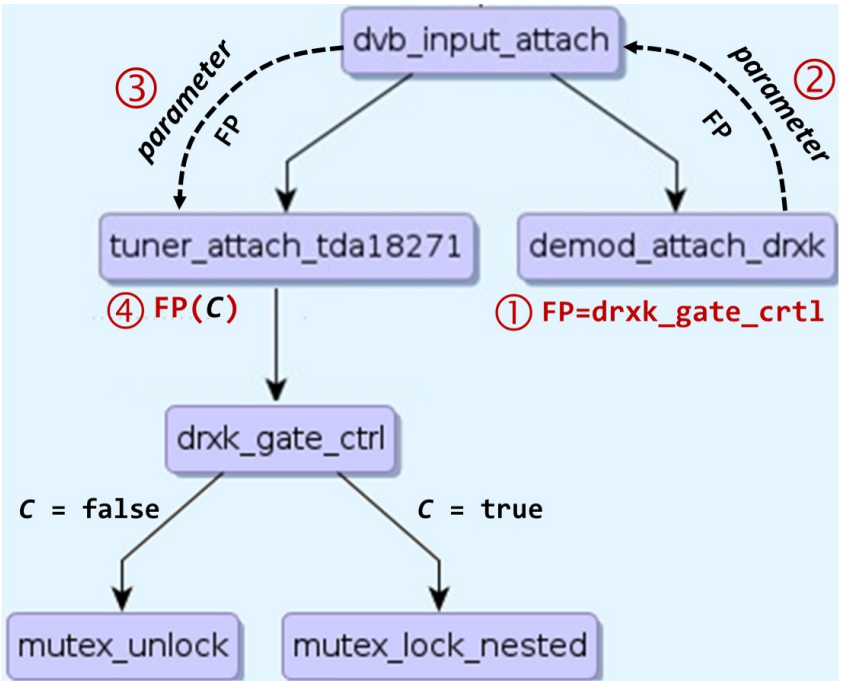


Figure 5.1 Search model for `drxk_gate_crtl` after resolving calls via function pointers

Recall that `f1` must be called twice. The function `f2` has a path on which there is a return before the second call to `f1` and thus it is a bug.

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

5.4 Contemporary approaches to software assurance

Now we discuss the contemporary approaches to generate artifacts which can serve the role of a proof. Several software assurance techniques aim to generate a *proof*. The *proof* in this context, is a series of mechanised deductive arguments that are logically sound, and are used to establish one or more properties of a *formal model* of the system being analyzed [50]. Needless to say, such techniques have helped in discovering security vulnerabilities that otherwise might have gone undetected. Lowe [51] famously used formal methods to uncover and help fix a decades-old security (design) flaw in the Needham-Schroeder public key protocol. There is no doubt that these techniques improve software safety and security in general.

A popular approach is completely mechanised validation of the verification process. The approach taken involves a use of a tool that verifies a software and generates a *proof* that can be checked using a proof checker. This process is referred as *certification* [52]. This squarely puts the onus of assuring the correctness onto the proof checker and they must be held to a very high standard for them to be used to verify mission-critical software. Avionics industry have developed various certification standards, such as DO-178C [44], but these processes can be costly and are generally riskier to implement. Ongoing research aims to improve the cost of adoption and reduce the perceived risk to increase industry level adoption of these techniques [53, 54].

Another popular approach is to generate results which can be validated by a machine while allowing some level of transperancy for a human. A widely used idea in these techniques is *Witness Validation* [55]. Traditionally, a verification tool would report a problem by generating counterexample trace(s). The traces are often tool-specific and cannot be validated by other tools. This would generate false alarms and determining whether a counterexample is a false alarm or not is a more often than not a tedious process [56]. Beyer, Dangl, Dietsch, and Heizmann have tried to address this problem [55, 57]. The key contribution is to create a witness exchange format that can be used by different tools to exchange witnesses. This allows the witness to be validated by multiple verifiers and instill greater confidence in the correctness of the machine generated *proof*. Witness validation techniques are based on the notion of *model checking*. Model checking is an algorithmic technique to

verify a system description against a specification. [58, 59] A witness generated by a model checker is typically an abstraction that captures the system description. These abstractions are typically automata which another model checker can stepwise validate against the specification [56]. The suggestion, if a human wishes to cross-check, is to use a visualization tool to inspect these automata manually. The problem is that these automata can be inordinately large, encapsulating thousands of state transitions in real-world software, which makes it practically impossible for a human to cross-check the witness. Thus, if the system description is incorrectly captured by the model, which again can only be assured by a human using a similar process, then the witness validation will fail to report a false alarm; worse it may miss a vulnerability which simply cannot be allowed in a mission-critical software.

There is a growing interest in human interpretable verification. This is visible in the recent programs launched by DARPA for Explainable Artificial Intelligence (XAI) [60] and Computers and Humans Exploring Software Security (CHESS) [61]. It is not a new trend by any measure and this direction of research has been pursued for a long time. We trace the origin to the landmark paper by DeMillo, Lipton, and, Perlis [5], discussed in Section 5.2, which argues against the notion that software verification proofs should be viewed as a long chain of deductive logic. They point out that a mathematical proof is constructed by elevating concepts instead of breaking down the argument into a chain of low-level arguments. These proofs are not formal in the strictest sense; they are exchanges that increase a *human's* confidence in the correctness of the argument. David Parnas, an early pioneer of software engineering, has been a long time supporter of this direction of research and his work discusses possible approaches to such proofs. [62, 63].

The work that comes closest to implementing Perlis's vision is by Ahmed Tamrawi [64]. Tamrawi developed Projected Control Graph (PCG) [37] which works as a concise evidence that captures the groupings of relevant CF paths required to verify a given instance. Tamrawi studied the Lock/Unlock Pairing problem in the Linux kernel and showed that all the Lock/Unlock instances in the Linux kernel can be verified automatically and evidence for the same can be generated. However, they do not provide a succinct evidence that shows the data flow relation among the relevant functions

and hence it does not work as well with Memory Leak (ML) problem in the Linux kernel where computing the data flow relation is more challenging.

CHAPTER 6. VERIFICATION EVIDENCE REQUIREMENTS

In this chapter, we discuss what program artifacts should be captured by the verification evidence. We also discuss the challenges involved in computing the verification evidence.

6.1 Verification Evidence

We shall use the *memory leak* (ML) problem as an example to discuss the general aspects of the verification evidence. We note that these aspects also apply to other Software Safety and Security problems. The ML verification requires a proof for each memory allocation instance. Thus, a verification evidence must be provided for each instance. We argue that the following program artifacts must be captured by the evidence.

- *Relevant Functions:* The set of functions that are relevant in the verification proof. The set could have one or more functions. For example, it is just one function if a memory allocation in function `f1` is followed by a memory deallocation in `f1` itself, on each feasible control flow (CF) path following the allocation. However, it could be two functions `f1` and `f2` if there is a feasible CF path in `f1` on which another function `f2` deallocates the memory.

- *Relations between Relevant Functions:* The data and control relations between relevant functions. For example, the control and data relations can be respectively `f1` calls `f2` and `f1` passes the pointer to the allocated memory to `f2`.

- *Coverage of CF Paths:* The grouping of all CF paths within each relevant function. The grouping is for the purpose of a concise proof. The number of CF paths grows exponentially with 2^n paths with n non-nested branch conditions. The paths where the same reasoning is applicable should be put in one group. For example, the CF paths following a memory allocation in `f1` may have three groups: (i) paths with deallocation, (ii) paths on which a

pointer to the allocated memory is passed to `f2`, (iii) paths without deallocation on which the pointer is not passed to another function.

- *Path Feasibility and Relevant Conditions:* The evidence must account for the path feasibility verification in order to avoid false positives. For example, feasibility must be checked for paths without deallocation on which the pointer is not passed to another function. It is not a memory leak if these paths are not feasible. The evidence should include the set of conditions relevant for checking the feasibility of a path.

- *Graphical Notation:* In order to enable human scrutiny of the verification process, the verification evidence should be communicated to a human in a format that the human can comprehend. A Graph is a suitable abstraction that can be visualized by a human to comprehend the verification evidence. The evidence should have a graphical notation which represents the artifacts described in this section as nodes and edges of a Graph.

6.2 Fundamental Challenges of Software Verification

There are two fundamental challenges which makes software verification difficult - 1) Control Flow Challenge: Exponential Number of paths, 2) Data Flow Challenge: Backward Data Flow due to Pointers. These are also the challenges involved in computing the verification evidence.

6.2.1 Control Flow Challenge

In order to verify a software safety or security property, each control flow path in the software must be analyzed to verify that the property holds true. If it does not then a path-feasibility check can be performed. Verification evidence must capture all the relevant control paths and group them for conciseness. It should also capture the relevant branch conditions required for the feasibility check. The fundamental challenge presented by Control Flow is that the number of behaviors can be astronomically large. The primary cause of this difficulty is branch conditions and loops.

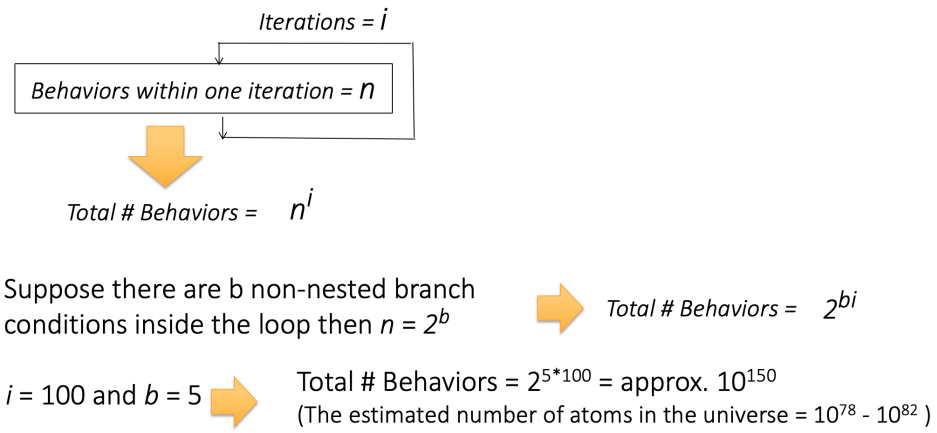


Figure 6.1 Astronomically large number of behaviors due to loops and branches

Figure 6.1 illustrates the challenge presented by branches and loops. The number of paths is 2^n with n 2-way non-nested branch nodes. This causes an exponential growth of the number of paths in the Control Flow Graph. Addition of loops in the mix muddles things further. First, enumerating all the paths in presence of a loop is not possible as it is equivalent to enumerating paths in a cyclic graph. To do that the termination bounds on the number of iterations of a loop needs to be computed. Computing exact loop termination bound is equivalent to the halting problem [65] and thus, undecidable. But even the use of an approximation is not going to help much.

Figure 6.1 shows a loop with n behaviors per iterations (n paths within the loop) and assumes the loop iterates i times. This results in a total of n^i behaviors. Assuming there are b non-nested branch conditions then $n = 2^b$. Which makes the total number of behaviors in this simple loop 2^{bi}. Even for a small loop with 5 branches that iterates 100 times, the number of behaviors are 2^{500} which is way more than the number of atoms in the universe. This challenge must be addressed in order to complete the software verification and compute the verification evidence.

6.2.2 Data Flow Challenge

A data flow analysis is required to map the allocations with the corresponding deallocations so that the Memory Leak (ML) problem can be solved. In general, data flow analysis enables

computation of the relevant functions as it captures the functions which access the allocated memory. In order to perform analysis, the pointer to the allocated memory must be tracked. This is a problem that needs computation of Def-Use (DU) chains to track the definitions of the pointer to the allocated memory (allocation) to their uses (deallocation). The challenge here is to address backward flow created by pointers.

Let's consider two cases - DU chains in absence of pointers and DU chains in presence of pointers.

```
1 int count = 0,x;
2 int p;
3 p = count;
4 count = 5;
5 x = p;
```

Listing 6.1 A simple DU chain

Consider the code in Listing 6.1. Line 1 defines an integer `count` to be 0. This definition is killed at line 4 and the new value is 5. But before that at line 3, `p` gets the old definition of `count`. This is the definition that is used at line 5. Thus, the DU chain that ends at line 5 is (1,3,5). This chain does not require to track what happens to the definition of `count` after line 3. Thus, one does not need to use a worklist-style algorithm [66] and these chains can be computed efficiently.

```
1 int count = 0,x;
2 int *p;
3 p = &count;
4 count = 5;
5 x = *p;
```

Listing 6.2 Backward flow due to pointers

Now consider the code in Listing 6.2. It is very similar to the code in Listing 6.1 except `p` is now a pointer. At line 3, p is made to point to the variable `count`. Then `count` is redefined at line 4. At line 5, `p` is de-referenced and is used to define `x`. This time the the definition at line 4 is used and the DU chain is (4,3,5). Note how one has to go back and forth on the control flow to compute the DU chain in a path-sensitive manner. This is the fundamental data flow challenge introduced

by the pointers. This problem is particularly severe in the Memory allocation-deallocation pairing problem [64].

6.2.2.1 Pointer Chains due to Structures

This challenge is made even more complex by the use of structures. Consider the code in listing 6.3. Lines 1-3 define a structure `st` that has a pointer field `p`. At line 6, a pointer to a structure `s` is defined. At line 7, a pointer to allocated memory is stored in the field `p` of `s`. This is the pointer that needs to be tracked and used for deallocation. Note that the pointer returned by the function is the pointer to the parent structure. Thus, not only the pointer to the allocated memory needs to be tracked, but the pointer to its parent structure also must be tracked.

```
1 typedef struct st {
2     int *p;
3 } st;
4
5 st* foo() {
6     st *s;
7     s->p = alloc();
8     return s;
9 }
```

Listing 6.3 Parent structure of a field also needs to be tracked

6.2.3 Related Work that address the verification challenges

Software verification problems have been studied for a long time. For literature survey, we focus works related allocation/deallocation pairing. The existing approaches to address these problems can be divided into two categories - static analysis techniques and dynamic analysis techniques.

There is a vast body of work [67, 68, 69, 70, 71, 72] on detecting deadlocks and memory leaks using dynamic analysis. However, running the program multiple times to examine all possible behaviors is prohibitively expensive, time-consuming, and it is hard to achieve complete coverage

of all possible behaviors. Hence, static analysis techniques are crucial to complement the dynamic analysis and testing.

There is a rich literature on static analysis tools [73, 74, 75, 76, 77, 78, 79, 80] for lock/unlock pairing and memory allocation/deallocation pairing. Tools such as Saturn [74], LockSmith [81, 82] have been shown to scale for older versions of the Linux kernel. But they suffer from lack of global alias analysis and incomplete function summaries and produce high number of false alarms. Saturn also has a limitation of detecting memory leaks of a specific type: a memory block that is allocated in function p and is never escaped and deallocated in p is considered a memory leak. This forms a very small subset of the instances of memory leak in a recent version of the Linux kernel. They also take one or more days to analyze a software of the scale of the Linux kernel making them impractical. FastCheck [83] uses guarded value-flow analysis to detect memory leaks. It is fast but limited to analyzing allocation sites whose values escapes to (flow into) top-level pointers only. Hence, it also is limited in applicability due to imprecisions in the data flow.

ESP [73] is a path-sensitive analysis tool that scales for large programs by merging superflous branches leading to the same analysis state. However, its analysis for 2-event problems is neither sound nor complete due to incomplete function summaries. Sparrow [84] is a static analysis tool that detects memory leaks using abstract interpretation to compute function summaries. It has been applied to small programs and it is not clear how well it scales for large code bases.

SVF [77], the modern incarnation of SABER [76, 85], is a static detector for memory leaks in C programs. SVF is implemented using LLVM [86] as opposed to SABER, improving its usability. It performs a sparse value flow analysis to track the pointers to the allocated objects. It does not track the pointers which flow into the global variables. Although, SVF is the most recent memory leak detector, it can not be applied to large code bases such as the Linux kernel due to the following challenges: (1) SVF uses the LLVM compiler to transform the codebase into the SSA, which does not scale well for the Linux kernel. (2) The sparse value-flow analysis used by the SVF is based off Andersen's pointer analysis. [66]. Andersen's algorithm has been extensively studied over the years and it is hard to scale to large code bases due to its $O(n^3)$ computational complexity. SVF's

pointer analysis might be improved by mixing it with unification-based analyses [87, 88], however that will drastically affect detection accuracy [89].

The best rated tool for verifying Linux is Linux Driver Verification (LDV) project [48]. LDV uses the BLAST [90] and has successfully found several bugs in the Linux kernel. Test cases generated by LDV are even used as regression tests by other tools such as CPAChecker [75], another static analysis tool used for memory allocation/deallocation pairing. LDV is a very complex tool and setting up LDV to run on a linux kernel is not straight forward. Secondly, LDV does not produce a verification evidence that can be checked by a human and hence it is difficult to validate LDV results. In fact, we found a bug in the Linux kernel which was verified as safe by LDV [91].

Apart from LDV, M-SAP [64] is the only other known work to have undertaken an empirical study of the Linux kernel and is known to have scaled to the size of the Linux kernel. M-SAP suffers from inaccuracies in its data flow and as a result, cannot verify every instance in the Linux kernel.

CHAPTER 7. A NOVEL APPROACH TO COMPUTING VERIFICATION EVIDENCE

In this chapter, we describe our approach to computing verification evidence. We first describe how the fundamental challenges of software verification described in Section 6.2 are addressed. Then, we discuss a novel abstraction, *Evidence Graph* that succinctly captures the verification evidence and can be used to prove the verification. We illustrate our approach using the *Lock/Unlock* and *Memory Leak* (ML) instances.

7.1 Addressing the Software Verification Challenges

In this section, we discuss our approach to handle the fundamental challenges described in Section 6.2.

7.1.1 Using Regular Expressions to handle Loops

As discussed in Section 6.2.1, the challenge presented by the control flow is the explosion in the number of behaviors due to the branch conditions and the loops. Projected Control Graph (PCG) [37, 92] was developed primarily to tackle the growth due to the branch conditions. We use PCG to define a homomorphism on the set of behaviors and map them to a smaller set of behaviors where only the statements relevant to the verification task are retained.

To account for the loop, we make use of regular expressions to capture the loop behaviors. This approach is inspired from the classic work of Robert Endre Tarjan [93, 94]. Tarjan showed how regular expressions can be used to accurately capture all the behaviors in a loop and also provides a fast algorithm to compute the regular expression for a loop. We will now illustrate how this approach works using an example.

Listing 7.1 shows the source code for the example. It has a `lock` instance at line 4 and it needs to be unlocked on every feasible execution path. The code also contains a loop which begins at line 3. C1, C2, C3 are the boolean variables that denote the condition values and can change as the code executes.

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

Listing 7.1 Example from Linux kernel to illustrate the use of regular expressions to handle loops

Let's first separate the behaviors of this program in three categories

1. Execution behaviors where loop iterates: There are two such behaviors (denoted using line numbers)

   (a) (2, C1(T), 4, C2(F), 8, C3(T), 11)

   (b) (2, C1(T), 4, C2(F), 8, C3(F), 13)

   C(T) means the condition C is `true` and C(F) means the condition C is `false`.

2. Execution behavior where the loop breaks: There is one such behavior (2, C1(T), 4, C2(T), 6, 16)

3. Execution behavior where the loop does not iterate: There is one such behavior (2, C1(F), 16)

Let's first count the number of behaviors if the loop executes n times. For the first *n-1* iterations, the behaviors could be either (2, C1(T), 4, C2(F), 8, C3(T), 11) or (2, C1(T), 4, C2(F), 8, C3(F), 13). The last iteration is used to exit the loop. This could be either (2, C1(F), 16) which is the normal exit for the loop or (2, C1(T), 4, C2(T), 6, 16) which is exit using the break. Thus, the number of possible behaviors are $2^{n-1} * 2 = 2^n$. for n iterations. This can be succinctly captured by the regular expression

$$(2, (C1(T), 4, C2(F), 8, C3(11 + 13))*, C1((4, C2(T), 6) + \epsilon), 16$$

The * operator is the standard regular expression operator which denotes that particular subexpression can execute zero or more times. ϵ is the empty symbol in the regular expressions. This expression captures all possible behaviors of the code in Listing 7.1.

Verification Using Regular Expressions: Now we show how to use the regular expressions with the PCG to perform software verification. The first step is to compute the PCG for the given instance using lock and unlock as the relvant events. Figure 7.1 shows the computed PCG. The red node corresponds to the `lock` and the green nodes correspond to the `unlock`. We will denote `lock` as `L` and `unlock` as `U` for the remained of the section.

First thing to note is that the condition `C3` was omitted from the PCG. This is because it has no effect on the `LU` matching. The regular expression for the PCG is:

$$(C1(T), L, C2(F), U(8))*, C1(L, C2(T) + \epsilon), U(16)$$

where U(8) and U(16) represent the `unlock` on line 8 and line 16 respectively. This expression can be broken down into two subexpressions

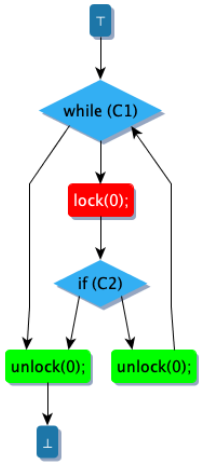


Figure 7.1 PCG for the instance in Listing 7.1

- Loop never iterates - $(C1(L, C2(T) + \epsilon), U(16)$. This shows that either `L` never happens (`C1 = F`) or it is matched with `U(16)`. Hence, there is no bug.
- Loop iterates at least once - Then it is equivalent to the expression where loop iterates exactly once and then exits on the next iteration - $C1(T), L, C2(F), U(8), C1(L, C2(T) + \epsilon), U(16)$. This shows that during the iteration, `L` is matched with `U(8)` and then it is matched with `U(16)`.

Thus, on every possible execution behavior `L` is matched by a `U`. This also implies that there is no need for feasibility check and the verification is complete.

7.1.2 Handling Backward Flow

As discussed in Section 6.2.2, the fundamental challenge presented by data flow is the backward flow created by pointers. We handle the backward flow as follows.

We first compute Data Dependence Graph (DDG) using the algorithm by Ferrante, Ottenstein, Warren (F.O.W.) [26]. This algorithm computes DDG in nearly linear time using Lengauer-Tarjan algorithm to compute dominators [95]. The DDG computed by the F.O.W. algorithm needs to be refined to address the backward flows and needs to made interprocedural. The refinements we apply are as follows:

- For every edge in DDG, associate the variable the definition of which flowed across that edge. This achieved using the node attributes in the Atlas framework.
- If the variable is a pointer, then compute the variables that the pointer can point to. We use the path-sensitive Andersen-style points-to analysis implemented by Atlas [38] to compute the points-to sets. For each non-pointer variable in the points-to set, connect the new definition to directly to the non-pointer variable by adding a new edge. This resolves the pointer chain and makes the tracking easier.
- If the variable being defined is a field of a structure then add an edge from the definition of the structure to the definition of the field. This links the field definitions to the parent structure definitions, thus taking care of the other complication discussed in Section 6.2.2.1
- Add an edge from the callsite to each node in the function that was called where the definition of the parameter is used. This allows tracking of parameters.
- Add an edge from each `return` statement in the function to each callsite to the function if a variable was returned.
- If the definition being used is of a global variable then mark the edge to indicate it involves the use of a global variable.

These refinements allow us to track the pointers to the allocated memory and enable verification of the memory allocation-deallocation pairing problem in the Linux kernel. Figure 7.2 shows the UD chains computed using the our refined DDG for the example in Listing 6.2. It shows exactly two definitions reaching the definition of `x`. The first definition is the definition of the pointer `p`. This would be computed by the F.O.W. algorithm. After that, the analyzer will have to figure out the correct live definition of `count` using a path-sensitive analysis. Atlas's path-sensitive data flow analyzer is used to resolve the pointer flow and to save the traversal computation cost during the traversal of the DDG, a direct link from the resolved definition is added to the use. Thus, if we

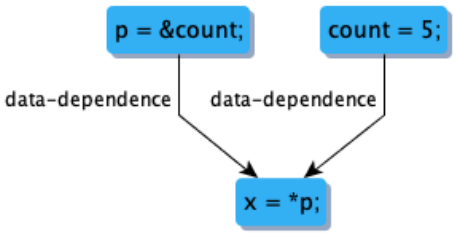


Figure 7.2 DDG refinements to resolve the backward flow

were tracking the value of the variable `count`, we will pay the cost of traversing just one edge to find its use.

The actual tracking to solve ML problem is done as follows,

- Compute the DDG for every function in the software. This is a one-time cost at the beginning of the analysis.
- Given an ML instance A to be verified, begin traversal of the DDG at the node for A. Traverse the DDG in the forward direction until a deallocation site D is discovered or a leaf in the DDG is encountered. Traverse only those edges along which the definition of A flows.
- If at any point, there is an incoming edge from a global variable (marked while creating DDG), then include the global variable in the analysis and traverse the DDG edges originating at the global variable along which the definition of A flows.
- If the definition of A is linked to a parent structure, then include the parent structure definition in the analysis and traverse the DDG edges originating at the definition of the parent structure along which the either definition of A flows or the definition of the parent structure flows.
- The end result will be the set of deallocation sites D that match with the given instance A. Use PCG and regular expressions to verify that the match is present along every feasible execution control flow path.

7.2 Evidence Graph

In this section, we describe a novel *certification evidence* (CE) schema designed to justify the verification proof in a succinct visual form to make it easy to comprehend and cross-check the proof. The ML verification requires a proof for each memory allocation instance. Thus, the CE schema serves to provide the verification evidence for each instance. We use this schema to compute a novel abstraction, *Evidence Graph* that satisfies all the requirements described in Chapter 6 that can be used to prove the ML verification.

7.2.1 A Graphical Notation for the CE Schema

We present a graphical notation as a succinct visual medium to communicate the certification evidence. We shall use the term *macro evidence* for the first two categories of evidence (relevant functions and relations between relevant functions) and the term *micro evidence* for the last two categories of evidence (coverage of CF paths and path feasibility).

The CE schema depicts the relevant functions as nodes and interactions between the functions as edges. Suppose the interaction is: function `f` passes a pointer to the allocated memory as a parameter to another function `g`. Then, the schema has an edge from `f` to `g` with label *par* to denote relevant parameter passing.

The CE schema is extensive enough to describe a variety of data and control flow interactions between functions. It incorporates all three modes of communicating data between functions: (i) passed as a parameter, (ii) passed as return value, and (iii) shared through a global variable. It incorporates all three modes for passing control from one function to another: (a) one function calls another function by name, (b) one function calls another function using a function pointer, and (c) two functions work asynchronously and so the control passes from one function to another because of context switching.

Unification of Macro and Micro Evidence: The micro evidence shows that the control flow paths within a function `f` are grouped into different groups for the purpose of verification. The macro evidence shows interactions of `f` with other functions for the purpose of verification. Consider the

following scenario that calls for unification of macro and micro evidence. Start with function `f` that allocates memory. The control flow paths in function `f` fall into three groups: a group of control paths that include deallocation, a group of control paths that pass a pointer to the allocated memory to function `g`, and a third group of control paths that do not have either, i.e. they do not include deallocation and they do not communicate the pointer to allocated memory to any other function. We need to represent succinctly the grouping of paths (micro evidence) and the relevant passing of parameter (macro evidence).

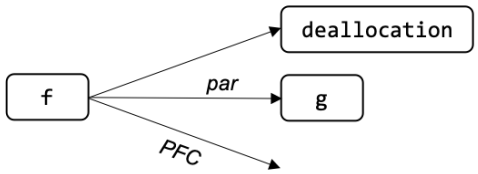


Figure 7.3 Function `f`: control flow paths fall into three groups

Let us now discuss how the micro and macro evidence are unified to communicate the above scenario. As shown in Figure 7.3, the functions `f` and `g` are denoted by nodes. The three outgoing edges from `f` correspond to the three groups of control paths. An edge from `f` to `g` with label *par* denotes the relevant parameter passing along a group of paths. An edge from `f` to `deallocation` denotes the matching deallocation along a group of paths. An edge from `f` with label *PFC* denotes that a path feasibility check (PFC) is performed along a group of paths.

7.2.2 Evidence for a Real-World Example Using the CE Schema

We illustrate the use of CE schema as a verification proof for an ML instance from the XINU operating system which is about 10,000 lines of code [96]. In XINU `getbuf` and `freebuf` are respectively the calls to allocate and deallocate memory. The memory in `dgwrite` is allocated by the `getbuf` call. We will use the CE schema to present evidence that every feasible CF path following this allocation in `dgwrite` has a corresponding `freebuf` call.

CE Evidence: As Figure 7.4 illustrates, the problem starts with memory allocation instance (`getbuf` call) in function `dgwrite`. The paths in `dgwrite` fall into two groups: paths with matching `freebuf`, and

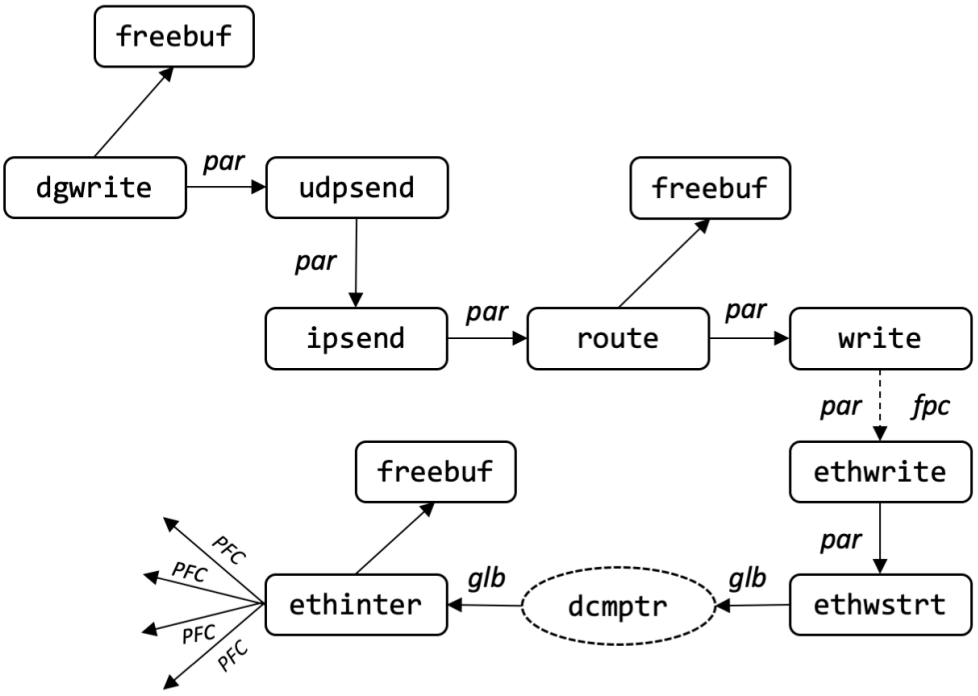


Figure 7.4 Evidence Using the CE Schema for the Allocation Instance in `dgwrite`

paths on which the pointer to the allocated memory is passed as a parameter to function `udpsend`. The parameter passing is shown by an edge from `dgwrite` to `udpsend` with the edge label *par*.

As per the CE evidence shown in Figure 7.4, the verification has taken the following steps:

1. Paths in `udpsend` fall into one group and the pointer to the allocated memory is passed as a parameter from `udpsend` to `ipsend`.

2. Paths in `ipsend` fall into one group and the pointer to the allocated memory is passed as a parameter from `ipsend` to `route`.

3. Paths in `route` fall into two groups. The pointer to the allocated memory is passed as a parameter from `route` to `write` along one group of paths and there is a matching `freebuf` along the other group of paths.

4. Paths in `write` fall into one group and the pointer to the allocated memory is passed as a parameter from `write` to `ethwrite`. `write` calls `ethwrite` using a function pointer. The function pointer call is denoted by the edge label *fpc*.

5. Paths in `ethwrite` fall into one group and the pointer to the allocated memory is passed as a parameter from `ethwrite` to `ethwstrt`.

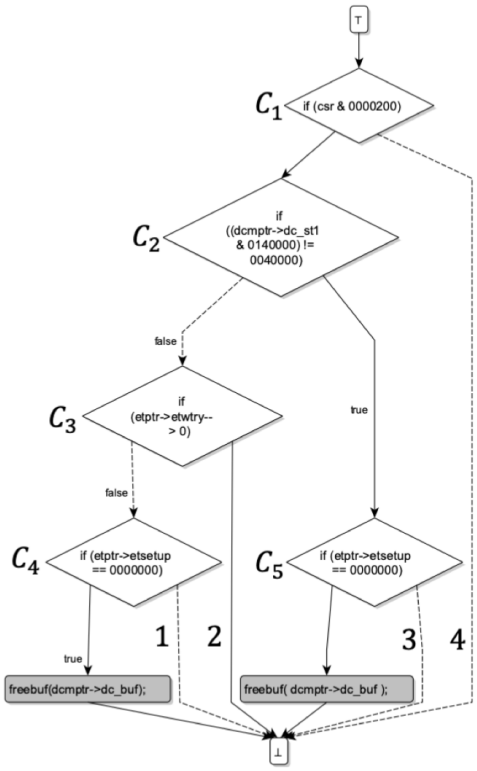
6. Paths in `ethwstrt` fall into one group and the pointer to the allocated memory is assigned to the global variable `dcmptr`. The function `ethinter` reads that global variable.

7. Paths in `ethinter` fall into five groups. One group of paths have a matching `freebuf`. The other four groups of paths do not have a matching `freebuf` and the pointer to the allocated memory is not communicated to any other function. These four group of paths require path feasibility checks as donated by the edge label *PFC*.

Path Feasibility Check: We use *Projected Control Graph* (PCG) [92] as a compact and easy-to-understand abstraction to capture micro evidence; it captures the relevant behaviors and relevant conditions for feasibility check. A control flow graph (CFG) can have many irrelevant conditions and a multitude of paths with identical relevant behavior for the purpose of verification, while the PCG has only one path per relevant behavior and captures only the relevant conditions.

The PCG for `ethinter` is shown in Figure 7.5(a); it shows six groups of paths. Two groups of paths have a matching `freebuf` (collapsed into one in Figure 7.4) and the other four groups of paths requiring feasibility checks are numbered 1 to 4 in Figure 7.5(a). The conditions relevant for checking path feasibility are captured by the PCG. An equivalent representation of the path feasibility conditions using a tabular notation is shown in Figure 7.5(b). The four groups of paths, all require feasibility checks but are still maintained as distinct paths because their corresponding relevant conditions are not the same and thus the feasibility checks are actually different.

Data and control relations between functions: This example brings out two out of the three data relations between functions: parameter passing and global variables. It also illustrates all three control relations between functions: a function is called by name, a function is called using a function



(a) `ethinter` PCG

C_1	C_2	C_3	C_4	C_5	Relevant Behaviors
T	F	F	T	$*$	`freebuf`
T	F	F	F	$*$	-
T	F	T	$*$	$*$	-
F	T	$*$	$*$	T	`freebuf`
F	T	$*$	$*$	F	-
T	$*$	$*$	$*$	$*$	-

(b) `ethinter` behaviors

Figure 7.5 Micro-Evidence for `ethinter`

pointer, and asynchronous control between two functions (the control is passed from `ethwstrt` to `ethinter` by context-switch; `ethinter` is an interrupt-driven function).

7.3 Applicability of the CE Schema

The CE schema is well-suited for verifying safety and security properties, especially *2-event properties*. Many safety and security properties are similar in terms of analysis and verification challenges they pose. A typical safety property can be stated as: verify that an event $e_1(O)$ is followed by an event $e_2(O)$ on every feasible execution path, where the two events are operations on the same object O (e.g., `lock` must be followed by `unlock`). A typical security property can be stated as: verify that an event $e_1(O)$ is *not* followed by an event $e_2(O)$ on every feasible execution path,

where the two events are operations on the same object O. For example, a confidentiality property is: a *sensitive source* must *not* be followed by a *malicious sink* on any feasible execution path.

As a general application, the CE schema is useful for presenting evidence for verifying a property defined by a finite state machine (FSM): verify on every feasible CF path, the occurrence of events that follow the acceptability test defined by a FSM.

7.3.1 Evidence Graph: Implementation of CE Schema

We implemented the CE Schema using Atlas [38]. The evidence presented in form of a graph abstraction, *Evidence Graph*. Evidence Graph constitutes of the following,

1. The allocation instance
2. All the DU chains that originate from the allocation instance. This captures the evidence required to check that the data flow has been captured correctly. These chains are interprocedural and span across functions. They capture transfer of pointer definition via parameters, returns, and global variables.
3. Relevant Functions: These are captured by the DU chains and are included in the Evidence Graph as containers for the control flow nodes.
4. If there are paths in a function where the pointer to the allocated memory is not used, then they are shown by an edge to a node denoted by $\perp$. This captures paths on which the pointer to the allocated memory does not reach a deallocation site and indicates the need for a path-feasibility check. This is achieved by computing a PCG for each of the captured functions.

Figure 7.6 shows the evidence graph for the XINU instance of ML. The red colored node is the allocation instance. It shows that there is a set of paths where it was passed to a deallocation site (`freebuf` callsite). The pointer is also used at to access a field and that is captured in the Evidence Graph. The edge from $\top$ to $\perp$ indicates that there are paths on which the pointer is not allocated

and they should not be analyzed. The deallocation sites are colored green. The evidence graph shows that they are in functions `dgwrite`, `route`, and `ethinter`. The blue node is the function pointer callsite. The node colored white is the global variable which is used to pass the pointer from the function `ethwstrt` to `ethinter`. Whereever there are edges to a $\perp$ node they indicate paths on which the pointer does not reach a deallocation site and they need to be checked for feasibility.

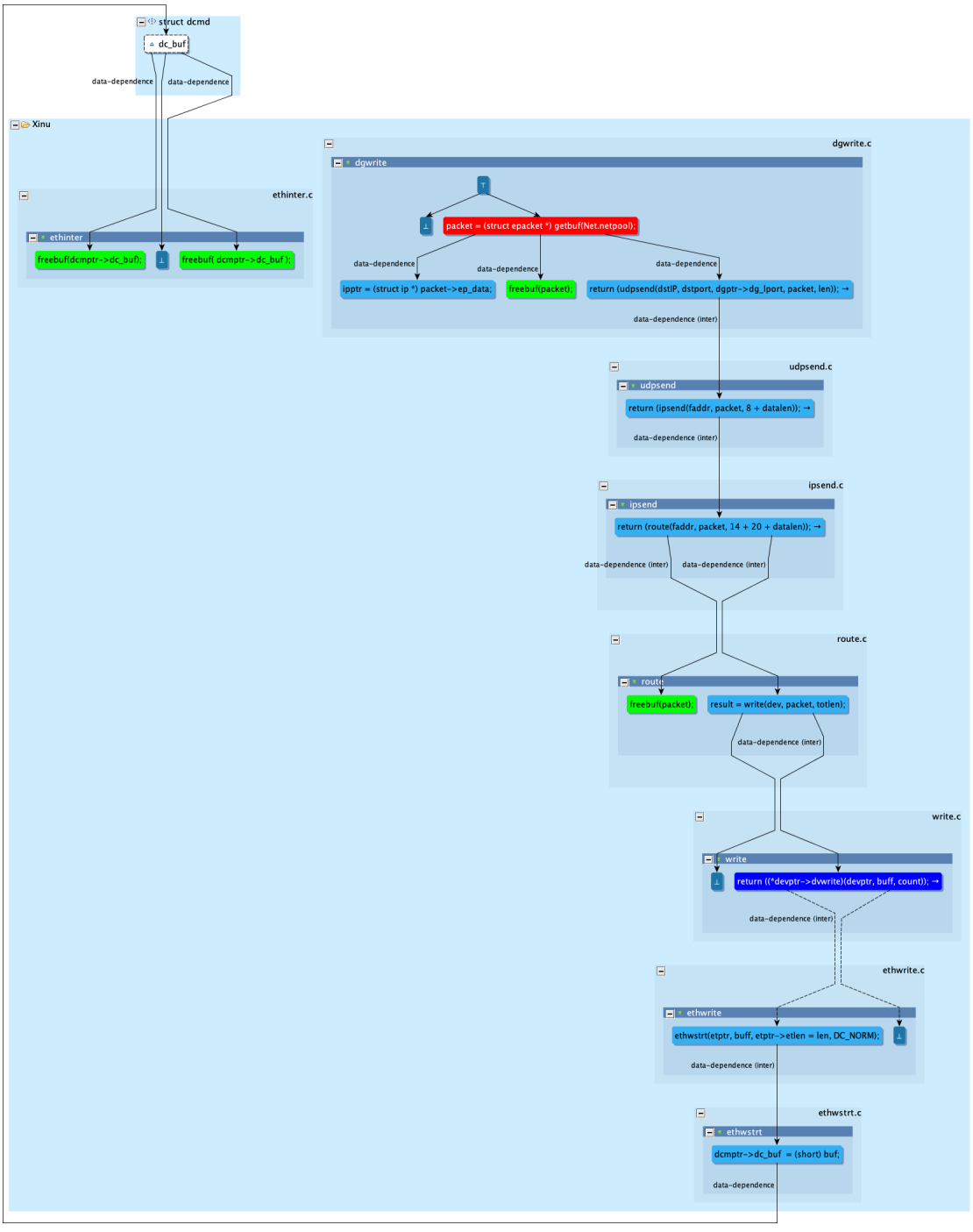


Figure 7.6 Evidence Graph for the XINU instance

CHAPTER 8. EVALUATION

In this chapter, we evaluate our approach using an empirical study of the Linux kernel for Memory Leak (ML) verification. We chose the *defconfig* build of Linux 5.0. This is the subset of the kernel that includes the default system utilities and drivers in the Linux kernel. Importantly, it includes the most critical parts of the Linux kernel such as file system, memory management, and crypto utilities. We first present our detection results and compare against the accuracy rates reported by the state-of-the-art software verification tools. We then present some interesting case-studies from the Linux kernel to illustrate how the Evidence Graph can be used to prove the verification.

8.1 An empirical study of the Linux kernel

We first present the results from our empirical study of the *defconfig* build of the Linux kernel 5.0. We performed the analysis for the ML problem. The default system call to allocate memory in the Linux kernel is `kmalloc` and the corresponding deallocation system call is `kfree`. There are 180 instances of `kmalloc` in the build. Our analysis reports three possible results

1. *Safe*: This means the verifier could match the given allocation instance with a corresponding deallocation site on every feasible execution path.

2. *Unsafe*: This means the verifier could match the given allocation instance with a corresponding deallocation site on some feasible execution paths but there are also paths on which it could not match with a corresponding deallocation site.

3. *Unknown*: This means the verifier could not match the given allocation instance with a corresponding deallocation site on any feasible execution path. In other words, it failed to find a corresponding deallocation site.

Our verifier does not timeout on any of the instances in the Linux kernel and is sound i.e. whenever it reports an instance to be Safe it is a Safe instance and there is no problem. It can raise false alarms due to inaccuracies in path-feasibility check or pointer-tracking due to hard to model programming language features such as pointer arithmetic. But in those cases, Evidence Graph points out the reason for the verifier's failure and allows a human to take over and complete the verification.

Table 8.1 Verification Results for Memory Leak (Linux Kernel 5.0, defconfig)

Kernel	kmalloc Instances	Results		
		Safe	Unsafe	Unknown
5.0	180	116	7	57
Distribution		64.4%	3.9%	31.7%

Table 8.1 show the results of our study. Out of the 180 `kmalloc` instances, the verifier can automatically verify 116 instances as Safe and has a 64.4% detection accuracy. Out of the remaining 64, 7 instances were reported as Unsafe, i.e. there are paths that the verifier thinks are feasible and the instance cannot be matched with a deallocation. For the remaining 57 instances, the verifier could not find a corresponding deallocation and hence reports them as Unknown. We manually inspected the Evidence Graphs for all the 180 instances. All the Safe instances are in fact safe and the verifier has not made any mistakes. The 7 unsafe instances are false alarms and are revealed to be safe with a path feasibility check. The 57 unknown instances revealed complexities related to pointer arithmetic which are currently beyond the scope of this work.

These results compare favorably with the existing state-of-the-art. LDV [97] is a popular Linux Verification tool, which has won various software verification compeititions [33]. A study of LDV conducted by Tamrawi [64] shows that LDV has a detection accuracy of *12%*. The previous best results for Memory Leak verification on any Linux kernel was by Tamrawi [64] that reported a detection accuracy of 35.8%. The other tools discussed in Section 6.2.3 are not able to complete the verification of the Linux kernel due to scalability limitations of the LLVM framework [76, 85, 77]. Our empirical study clearly shows a significant improvement in the memory leak verification results.

8.2 Linux Kernel Case Study - I

We present a case study from the Linux Kernel that involves a loop. The relevant source code is shown in Listing 8.1. The instance `A` is on line 6. It also shows that there is a matching deallocation `D` on line 11. The only thing that remains is whether `ascii_filter(t)` call does something to the pointer and handle the loop.

```
1  static ssize_t get_modalias(...) {
2    for (f = fields; f->prefix && left > 0; f++) {
3      if (!c) {
4        continue;
5    }
6      t = kmalloc(strlen(c) + 1, GFP_KERNEL);
7      if (!t) {
8        break;
9    }
10     ascii_filter(t);
11   kfree(t);
12   }
13 }
```

Listing 8.1 ML instance from the Linux Kernel involving a Loop

Figure 8.1 shows the Evidence Graph. Evidence Graph shows that `ascii_filter` does not modify the pointer only uses to access some value. Thus, the only `D` that can be matched is on line 11 in Listing 8.1. It also shows that there is a path on which the `A` is not matched with `D`. We should analyze the PCG to find that path and perform a path-feasibility check.

Figure 8.2 shows the PCG for the function. The regular expression is:

$$(2, 3, (4 + (A, 7, D)))*, ((2, 3(F), A, 7(T), 8) + \epsilon)$$

Clearly, there is a behavior (2,3(F),A,7(T),8) where `A` is not matched with `D`. A trivial feasibility check shows that this behavior will only happen when the pointer to the allocated memory, `t` is

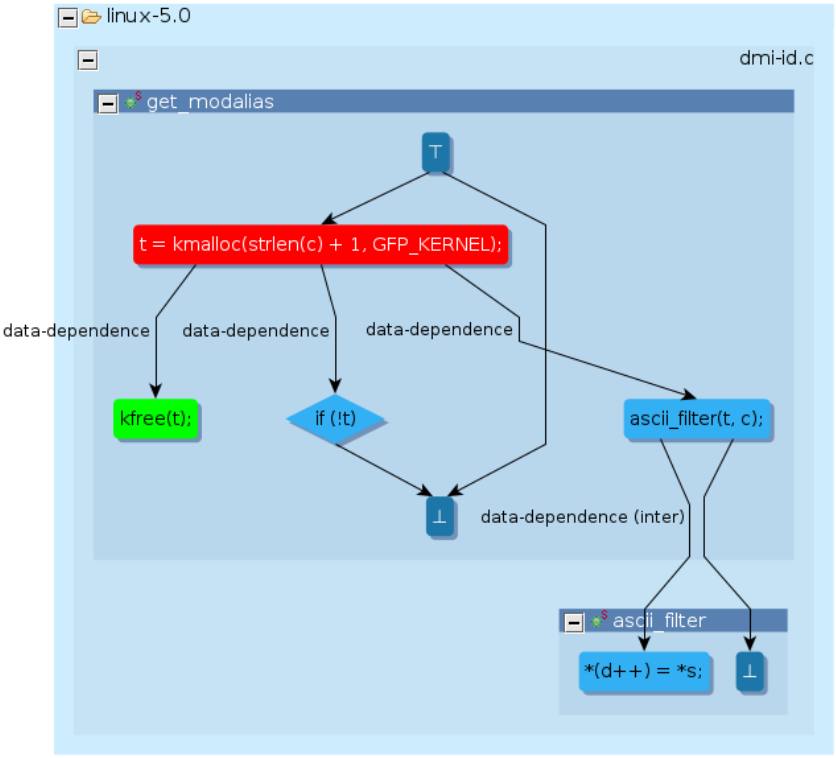


Figure 8.1 Evidence Graph for Case Study I

NULL. In other words, there is nothing to be freed. Hence, there is no memory leak and the instance is declared *Safe*.

8.3 Linux Kernel Case Study - II

We present a case study from the Linux Kernel that involves backward data flow. This is created due to the use of structures. Listing 8.2 shows the relevant code segment. The instance A is on line 4. The pointer to the allocated memory is tmp_buf. It is passed to the function regmap_raw_read on line 8. Further inspection shows that it is a library function (there is no Control Flow to be computed as there is no source code available). Since library functions are not expected to free the memory, the verifier chooses to ignore it. There is a deallocation site D on line 13 but on line 11 A is stored inside a global variable. Now, that structure, specifically that particular field also needs to be tracked.

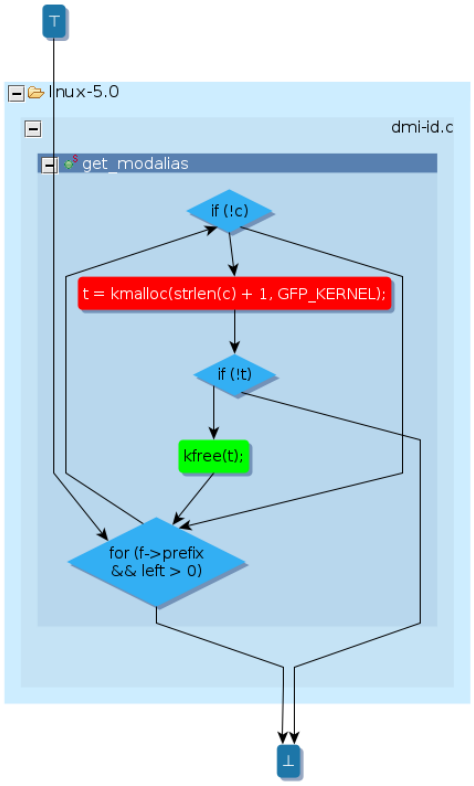


Figure 8.2 PCG for Case Study I

```
1  static int regcache_hw_init(...) {
2    ...
3    if(!map->reg_defaults_raw) {
4      tmp_buf = kmalloc(map->cache_size_raw, GFP_KERNEL);
5      if(!tmp_buf) {
6        ...
7      }
8      ret = regmap_raw_read(map, 0, tmp_buf,
9                map->cache_size_raw);
10     if (ret == 0) {
11       map->reg_defaults_raw = tmp_buf;
12     } else {
13       kfree(tmp_buf);
14     }
```

```
15    }
16    ...
17 }
```

Listing 8.2 ML instance from the Linux Kernel showing backward data flow

Figure 8.3 shows the evidence graph for the instance. The red node is the instance A. Green nodes show the deallocation sites it matched with. The white node shows the field of the global structure. This field is accessed by two functions, `regcache_init` and `regcache_exit` to deallocate it (which is what is shown in the Evidence Graph). It is accessed and used by many other functions but for the sake of clarity, we are not showing all of its uses. Note that the evidence graph is also indicating potential paths on which the allocation is not matched with a corresponding deallocation. We should look at the PCG now.

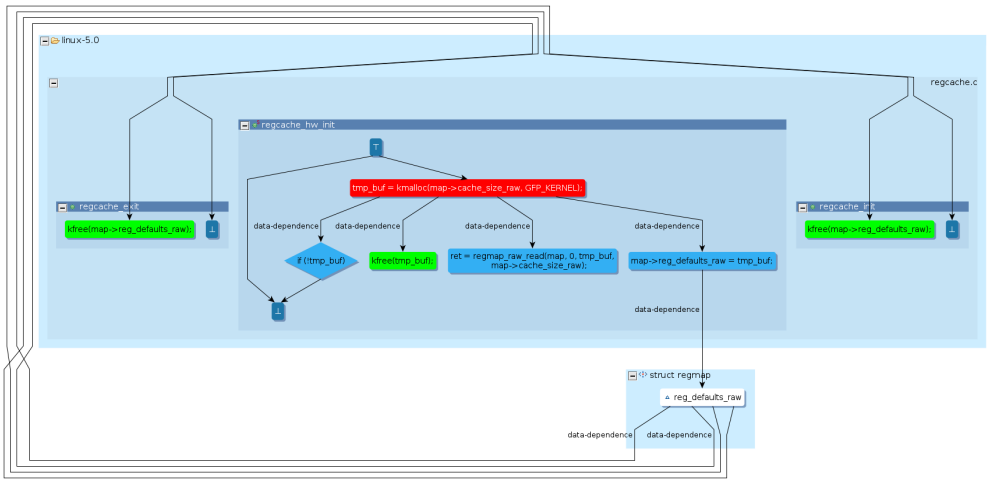


Figure 8.3 Evidence Graph for Case Study II

Let's start with the PCG of the function `regcache_hw_init`. Figure 8.4 show the PCG. It shows that on a path (outgoing edge from `if(!tmp_buf)`), it is neither followed by the deallocation nor the assignment to the global variable. This path needs to be checked for feasibility. A simple feasibility check shows that this path is feasible only if `tmp_buf` is `NULL`, which means no memory was allocated. Thus, it is not a problem and as long as the global variable is freed on all feasible execution paths, this instance is *Safe*.

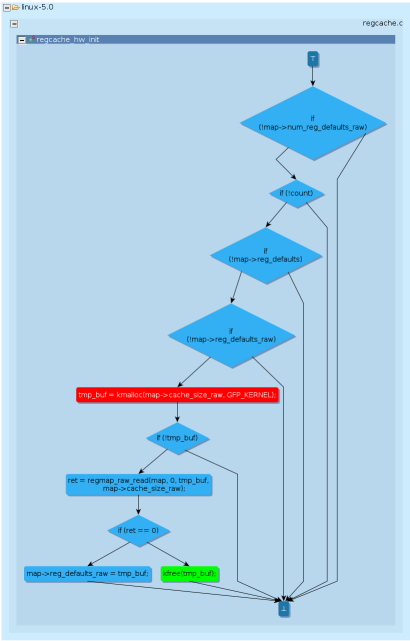


Figure 8.4 PCG for the function `regcache_hw_init`

Figure 8.5 shows the PCG for the function `regcache_init`, one of the function that accesses the global variable to deallocate it. Although the PCG looks pretty complicated, a feasibility check reveals that every path other than the path on which the allocated memory is freed is feasible only if the allocated memory is `NULL`. Thus, the instance is *Safe* in this function.

Figure 8.6 shows the PCG for the function `regcache_exit`, the other function that accesses the global variable to deallocate it. This PCG has three paths and one of them deallocates the memory. The feasibility check reveals that for the other paths to be feasible, the parent global structure needs to be `NULL`. Which again means, there is no memory to be deallocated. Thus, the instance is *Safe* in this function as well.

To conclude, the instance is Safe in all the functions that attempt to deallocate it. The functions access this global variable asynchronously (these are hardware interrupts working on cache registers of the machine). The CE Schema and the verifier based on it, currently do not support asynchronous verification. Thus, as long the hardware itself is not faulty, this instance is *Safe*. The verifier assumes to be the case and declares the instance to *Safe*.

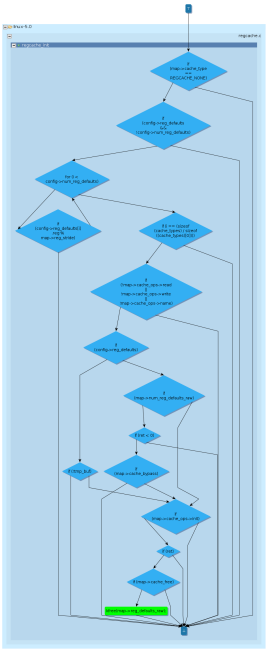


Figure 8.5 PCG for the function `regcache_init`

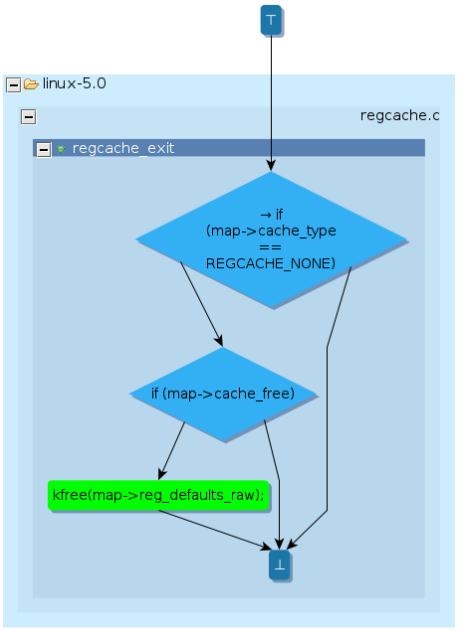


Figure 8.6 PCG for the function `regcache_exit`

CHAPTER 9. CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

Software assurance is necessary to avoid costly software failures and to mitigate exploitation. There is a need to establish confidence in the correctness of software verification. De Millo, Lipton, and Perlis were among the first computer scientists to advocate that the proofs of the verification should be a social process like mathematical proofs. Mathematical proofs increase the confidence of a human in the correctness of the theorem. We argue, along the same lines, that the call for human-machine collaboration to detect safety and security vulnerabilities and scrutinize the verification results is a practical call to raise our confidence in the correctness of the verification results.

We discussed our approach to enable the human-machine collaboration for two classes of software vulnerabilities. The first class, Algorithmic Complexity Vulnerabilities (ACV), is a class of software security vulnerabilities rooted in loops that cause denial-of-service attacks. Often, a description of the ACV is not known a priori, it needs to be hypothesized first before attempting to detect the vulnerability. We developed a tool DISCOVER and an analysis workflow that can help a human analyst hypothesize and detect ACVs in web applications. We used this tool as a performer on DARPA STAC project [4] and achieved 100% accuracy in detecting ACVs to become one of the top teams.

The second class of the vulnerabilities we tackled was where the description of the vulnerability is known a priori. We used the Memory Leak (ML) problem as a representative example. It is important to prove the correctness of the verification process of the ML problem. We developed our approach based on the vision described in the visionary paper [5]. We developed *Evidence Graph*, a novel abstraction that presents all the necessary evidence required to reason about the correctness of the verification. A human can then scrutinize the evidence and "prove" that the verification was correct. This required advances in the computation of the program artifacts needed for the verification, specifically in the data flow analysis. We evaluated our approach against ML instances

in the Linux kernel. We verified 64.44% of the instances which is the best accuracy among the previously reported studies of the Linux kernel for ML problem. We also presented two case studies that show how the Evidence Graph can be used to prove verification.

9.1 Future Research Directions

Now we will discuss the future research directions. Specifically, we point out two areas for improvement in our work.

Programming Language specific challenges: Apart from the fundamental challenges described in Section 6.2, computing verification evidence can be hindered by programming language specific features. We describe the features we encountered during the Linux kernel verification study.

- *Pointer Arithmetic:* Use of pointer arithmetic by C developers obfuscates the data flow and makes tracking of the pointer to the allocated memory difficult. This usage is especially common to get the structure containing a given field. Linux kernel uses the `container_of` macro for this.

- *List Macros:* The pointer to the allocated memory is escapes to a list and obfuscates the data flow. This is done using macros in the Linux kernel which needs to be modeled properly.

- *Function Pointers:* The presence of function pointers makes it difficult to compute the relation between relevant functions.

We used Atlas [38] to implement our approach. Atlas in its current state does not support handling these features and thus our solution also can't handle them. Addressing these features will improve our results further.

Limitations of CE Schema: The CE schema currently does not encompass the evidence to support verification problems that can arise from faulty hardware-software co-design. For example, the CE schema does not include evidence to support the time and space requirements of real-time embedded systems. These are important requirements for avionics, automotive, medical, and other

safety-critical systems. The European Space Agency has set up the COMPASS project (Correctness, Modeling, and Performance of Aerospace Systems) [98]. Its goal is to develop a coherent and multidisciplinary approach towards developing systems at a systems engineering level. The COMPASS project has adopted the Architecture and Analysis Design Language (AADL) and its Error Model Annex [99] as the key formalism. Thus, a worthwhile exercise is to extend the CE schema to incorporate evidence based on AADL models.

BIBLIOGRAPHY

[1] M. Dowson, "The ariane 5 software failure," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 2, p. 84, 1997.

[2] P. Johnston and R. Harris, "The boeing 737 max saga: lessons for software organizations," *Software Quality Professional*, vol. 21, no. 3, pp. 4–12, 2019.

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

[4] DARPA, "Space / Time Analysis for Cybersecurity (STAC)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html, 2014.

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

[6] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks." in *Usenix Security*, vol. 2, 2003.

[7] MITRE, "Cwe-407: Algorithmic complexity," https://cwe.mitre.org/data/definitions/407.html.

[8] "Xml security: A billion laughs," https://cytinus.wordpress.com/2011/07/26/37/.

[9] "Denial of service in microsoft office 2007-2013," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2730.

[10] A. Klink and J. Walde, "Efficient denial of service attacks on web application platforms," in *28th Chaos Communication Congress*, 2011.

[11] J.-P. Aumasson, D. Bernstein, and M. BOBLET, "Hash-flooding dos reloaded: attacks and defenses," in *29th Chaos Communications Congress*, 2012.

[12] A. Research, "Public release items for the darpa space/time analysis for cybersecurity (stac) program," https://github.com/Apogee-Research/STAC.

[13] R. C. Waters, "A method for analyzing loop programs," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 237–247, May 1979.

[14] A. Barua and Y. Cheon, *Finding Specifications of While Statements Using Patterns.* Springer International Publishing, 2015, pp. 581–588.

[15] L. J. White and B. Wiszniewski, "Path testing of computer programs with loops using a tool for simple loop patterns," *Software: Practice and Experience*, vol. 21, no. 10, pp. 1075–1102, 1991.

[16] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 34, 2014.

[17] V. Aponte, P. Courtieu, Y. Moy, and M. Sango, "Maximal and compositional pattern-based loop invariants," in *International Symposium on Formal Methods*. Springer, 2012, pp. 37–51.

[18] S. K. Abd-El-Hafiz and V. R. Basili, "A knowledge-based approach to the analysis of loops," *IEEE Transactions on Software Engineering*, vol. 22, no. 5, pp. 339–360, 1996.

[19] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009, pp. 136–146.

[20] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 61–72. [Online]. Available: https://doi.org/10.1145/2950290.2950340

[21] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger, "Loop summarization using state and transition invariants," *Formal Methods in System Design*, vol. 42, no. 3, pp. 221–261, 2013.

[22] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 23–33.

[23] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen, "S-looper: Automatic summarization for multipath string loops," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 188–198.

[24] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 225–236.

[25] M. Weiser, "Program Slicing," in *Proc. ICSE '81*. IEEE Press, 1981, pp. 439–449.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[27] P. Awadhutkar, G. R. Santhanam, B. Holland, and S. Kothari, "Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 00, 2017, pp. 249–258.

[28] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[29] Y. Fratantonio, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Clapp: characterizing loops in android applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 687–697.

[30] B. Holland, G. R. Santhanam, P. Awadhutkar, and S. Kothari, "Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities," in *Proceedings - 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation, SCAM 2016*. IEEE, 2016, pp. 79–84.

[31] J. Dietrich, K. Jezek, S. Rasheed, A. Tahir, and A. Potanin, "Evil pickles: Dos attacks based on object-graph engineering," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[32] P. Awadhutkar, "Curated set of loops illustrating characteristics of loops that may lead to algorithmic complexity vulnerabilities," https://github.com/payas0awadhutkar/ACV-Loops-Repository.

[33] D. Beyer, "Competition on software verification," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 504–524.

[34] M. Wolfe, "Beyond induction variables," in *ACM SIGPLAN Notices*, vol. 27, no. 7. ACM, 1992, pp. 162–174.

[35] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, "Caramel: Detecting and fixing performance problems that have non-intrusive fixes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 902–912. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818863

[36] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for web services: Benchmarking approach and examples," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 269–283, 2015.

[37] A. Tamrawi and S. Kothari, "Projected control graph for accurate and efficient analysis of safety and security vulnerabilities," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*. IEEE, 2017, pp. 113–120.

[38] 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*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 588–591.

[39] T. Wei, J. Mao, W. Zou, and Y. Chen, "A New Algorithm for Identifying Loops in Decompilation," in *Static Analysis*, no. 2006. Springer, 2007, pp. 170 – 183.

[40] P. Awadhutkar, G. R. Santhanam, B. Holland, and S. Kothari, "Discover: Detecting algorithmic complexity vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 1129–1133. [Online]. Available: https://doi.org/10.1145/3338906.3341177

[41] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[42] N. R. Council *et al.*, *Software for dependable systems: Sufficient evidence?* National Academies Press, 2007.

[43] B. Brosgol and C. Comar, "Do-178c: A new standard for software safety certification," ADA CORE TECHNOLOGIES NEW YORK NY, Tech. Rep., 2010.

[44] F. Pothon, "Do-178c/ed-12c versus do-178b/ed-12b: Changes and improvements," *ACG Solutions*, 2012.

[45] D. Stan, "Stan 00-56," *Safety Management Requirements for Defence Systems", UK Ministry of Defence, Defence Standard 00-56*, no. 2, 1996.

[46] A. Galloway, R. F. Paige, N. Tudor, R. Weaver, I. Toyn, and J. McDermid, "Proof vs testing in the context of safety standards," in *24th Digital Avionics Systems Conference*, vol. 2. IEEE, 2005, pp. 14–pp.

[47] S. Kothari, P. Awadhutkar, and A. Tamrawi, "Insights for practicing engineers from a formal verification study of the linux kernel," in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 264–270.

[48] P. Andrianov and Team, "Linux driver verification (ldv) project," http://linuxtesting.org/ldv.

[49] S. Kothari, A. Tamrawi, and J. Mathews, "Human-machine resolution of invisible control flow?" in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–4.

[50] T. C. Murray and P. C. van Oorschot, "BP: formal proofs, the fine print and side effects," in *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 2018, pp. 1–10. [Online]. Available: https://doi.org/10.1109/SecDev.2018.00009

[51] G. Lowe, "Breaking and fixing the needham-schroeder public-key protocol using FDR," in *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 1055. Springer, 1996, pp. 147–166. [Online]. Available: https://doi.org/10.1007/3-540-61042-1_43

[52] K. S. Namjoshi, "Certifying model checkers," in *Proceedings of the 13th International Conference on Computer Aided Verification*, ser. CAV '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 2–13.

[53] L. G. Wagner, A. Mebsout, C. Tinelli, D. D. Cofer, and K. Slind, "Qualification of a model checker for avionics software verification," in *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, ser. Lecture Notes in Computer Science, C. W. Barrett, M. Davies, and T. Kahsai, Eds., vol. 10227, 2017, pp. 404–419. [Online]. Available: https://doi.org/10.1007/978-3-319-57288-8_29

[54] A. Mebsout and C. Tinelli, "Proof certificates for smt-based model checkers for infinite-state systems," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View,*

CA, USA, October 3-6, 2016, R. Piskac and M. Talupur, Eds. IEEE, 2016, pp. 117–124. [Online]. Available: https://doi.org/10.1109/FMCAD.2016.7886669

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

[56] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, "Witness validation and stepwise testification across software verifiers," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 721–733. [Online]. Available: https://doi.org/10.1145/2786805.2786867

[57] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, "Exchanging verification witnesses between verifiers," in *Tagungsband Software Engineering 2017, Fachtagung des GI-Fachbereichs Softwaretechnik (21.-24. Februar 2017, Hannover, Deutschland)*, ser. LNI P-267, J. Jürjens and K. Schneider, Eds. Gesellschaft für Informatik (GI), 2017, pp. 93–94.

[58] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981*, ser. Lecture Notes in Computer Science, D. Kozen, Ed., vol. 131. Springer, 1981, pp. 52–71. [Online]. Available: https://doi.org/10.1007/BFb0025774

[59] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2000.

[60] DARPA, "Explainable Artificial Intelligence (XAI)," https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-16-53/listing.html, 2016.

[61] ——, "Computers and Humans Exploring Software Security (CHESS)," https://www.fbo.gov/spg/ODA/DARPA/CMO/HR001118S0040/listing.html, 2018.

[62] D. L. Parnas, "Really rethinking 'formal methods'," *IEEE Computer*, vol. 43, no. 1, pp. 28–34, 2010. [Online]. Available: https://doi.org/10.1109/MC.2010.22

[63] ——, "The use of mathematics in software quality assurance," *Frontiers Comput. Sci. China*, vol. 6, no. 1, pp. 3–16, 2012.

[64] A. Y. Tamrawi, "Evidence-enabled verification for the Linux kernel," Ph.D. dissertation, Iowa State University, 2016.

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

[66] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, University of Cophenhagen, 1994.

[67] 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*. ACM, 1998, pp. 298–309.

[68] 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," *ACM Sigplan Notices*, vol. 37, no. 5, pp. 258–269, 2002.

[69] 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," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 326–336, 2014.

[70] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: Effective sampling for lightweight data-race detection," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 134–143. [Online]. Available: https://doi.org/10.1145/1542476.1542491

[71] M. Jump and K. S. McKinley, "Cork: Dynamic memory leak detection for garbage-collected languages," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 31–38. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190224

[72] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[73] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," *ACM Sigplan Notices*, vol. 37, no. 5, pp. 57–68, 2002.

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

[75] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 184–190.

[76] Y. Sui, D. Ye, and J. Xue, "Static memory leak detection using full-sparse value-flow analysis," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 254–264.

[77] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 265–266.

[78] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 237–252. [Online]. Available: http://doi.acm.org/10.1145/945445.945468

[79] 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*. ACM, 2007, pp. 205–214.

[80] T. G. Rossin, J. L. Ashburn, and J. M. Dewey, "Methods and apparatus for fast check of floating point zero or negative zero," Jan. 19 1999, uS Patent 5,862,066.

[81] P. Pratikakis, J. S. Foster, and M. Hicks, "Locksmith: Context-sensitive correlation analysis for race detection," *SIGPLAN Not.*, vol. 41, no. 6, p. 320–331, Jun. 2006. [Online]. Available: https://doi.org/10.1145/1133255.1134019

[82] ——, "Locksmith: Practical static race detection for c," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 33, no. 1, p. 3, 2011.

[83] S. Cherem, L. Princehouse, and R. Rugina, "Practical memory leak detection using guarded value-flow analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250789

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

[85] Y. Sui, D. Ye, and J. Xue, "Detecting memory leaks statically with full-sparse value-flow analysis," *IEEE Transactions on Software Engineering*, vol. 40, no. 2, pp. 107–122, 2014.

[86] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[87] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996, pp. 32–41.

[88] M. Das, "Unification-based pointer analysis with directional assignments," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 35–46, 2000.

[89] M. Shapiro and S. Horwitz, "The effects of the precision of pointer analysis," in *International Static Analysis Symposium*. Springer, 1997, pp. 16–34.

[90] P. Shved, M. Mandrykin, and V. Mutilin, "Predicate analysis with blast 2.7," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2012, pp. 525–527.

[91] S. Kothari, P. Awadhutkar, A. Tamrawi, and J. Mathews, "Modeling lessons from verifying large software systems for safety and security," in *Simulation Conference (WSC), 2017 Winter*. IEEE, 2017, pp. 1431–1442.

[92] A. Tamrawi and S. Kothari, "Projected control graph for computing relevant program behaviors," *Science of Computer Programming*, vol. 163, pp. 93–114, 2018.

[93] R. E. Tarjan, "A unified approach to path problems," *Journal of the ACM (JACM)*, vol. 28, no. 3, pp. 577–593, 1981.

[94] ——, "Fast algorithms for solving path problems," *J. ACM*, vol. 28, no. 3, p. 594–614, Jul. 1981. [Online]. Available: https://doi.org/10.1145/322261.322273

[95] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–141, jan 1979.

[96] D. Comer, *Operating system design: the Xinu approach, Linksys version.* Chapman and Hall/CRC, 2011.

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

[98] "The compass project," http://www.compass-toolset.org/.

[99] "Aadl error model annex," https://www.sae.org/standards/content/as5506/1a/.