Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities*

Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa 50011
Email: {payas, gsanthan, bholland, kothari}@iastate.edu

***Abstract*—Algorithmic complexity vulnerabilities (ACVs) can be exploited to cause denial-of-service. Detecting ACVs is hard because of the numerous kinds of loop complexities that cause ACVs. This renders automatic detection intractable for ACVs. State-of-the-art loop analyses aim to obtain precise loop iteration bounds automatically; they can do so for relatively simple loops. This research focuses on techniques to amplify intelligence so that the analyst can gain a deeper knowledge of complex loops that is necessary to discover ACVs. We describe: (a) loop abstractions and use them to define patterns and other characterizations of loop behaviors which in turn can be applied to create automated filters to isolate complex loops with high likelihood of ACVs, (b) innovative visual querying mechanisms for interactive loop analysis; they enable the analyst to hypothesize ACVs and gather the necessary evidence for targeted dynamic analysis for confirming ACVs. These capabilities are illustrated with an ACV detection case study. We present an empirical study using over 5000 loops from 4 open source libraries, and 18 DARPA challenge apps. The study evaluates the usefulness of the loop characterizations and patterns to enable the analyst to create effective filters to isolate complex loops.**

Tool: https://ensoftcorp.github.io/loop-comprehension-toolbox/

***Keywords*—Algorithmic complexity vulnerability, Loop comprehension, Smart views, Loop termination**

I. Introduction

The *algorithmic complexity vulnerabilities* (ACVs) are about runtime space or time consumption of programs. Adversaries can exploit ACVs to mount denial of service attacks. For example, the denial of service commonly known as the "billion laughs attack" or an XML bomb, is caused by an ACV in the application that creates a string of 10^9 concatenated " `lol`" strings requiring approximately 3 gigabytes of memory [1] when parsing a specially crafted input file less than a kilobyte. A recent study has characterized a class of ACVs in the Java library [2]. Similar to the XML bomb, it is a class of ACVs associated with the serialization and deserialization APIs.

A completely automatic analysis is intractable for detecting ACVs [3]. As discussed later, automatic analyses can determine precise loop iteration bounds only for relatively simple loops. On the contrary, ACVs typically result from complex loop termination logic. The DARPA STAC program [3] has called for a novel human-in-the-loop approach to detect ACVs.

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

We propose a four-step approach to detect ACVs: (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 [4].

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.

We have built an integrated tool chain to detect ACVs. The tool chain serves multiple purposes. We have used it to

discover loop abstractions through empirical studies of loops. It provides the functionality to generate the loop catalog, create the filters, and the visual querying mechanism. The tool chain is built using Atlas [5], a graph database platform for software analysis and visualization.

We present an empirical study based on the totality of more than 5000 loops from 4 open source libraries, and 18 DARPA ACV challenge apps. The purpose of the study is to evaluate usefulness and applicability of the abstractions, the loop behavior patterns, and the characterizations to create filters.

II. Research Gap

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

The program artifacts that can lead to ACVs include loops, recursion, or resource-intensive library APIs [2]. 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 [6].

As an experiment, we tried the state-of-the-art loop analysis tool Proteus [7] 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 [8]) 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* [9], (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 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 [10]. In line with the findings of [11], these smells tend to be either too specific (too many false negatives) or too generic (too many false positives).

To the best of our knowledge, there are no existing tools to help human analysts to detect ACVs. We are also not aware of any tools that allow the analyst to visually query for the relevant program artifacts that affect loop termination. Moreover, in detecting ACVs the added complication is to perform path-sensitive analysis. While a completely automated analysis of loop termination is intractable, a big need is for tools that can perform automated analysis to assist with human reasoning for loop termination. There is also a need for tools to help the analyst to visualize interactions through library calls because as discussed in the recent paper [2], the ACVs can be due to library calls.

We present research that employs automation to amplify human intelligence to address the research gap. It is research motivated by the Intelligence Amplification (IA) vision propounded by Frederick Brooks [12]: *"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."*

Organization: The paper is organized as follows. Section III describes loop abstractions that serve as building blocks for developing loop characterizations relevant to detect ACVs described in Section IV. Section V describes the interactive analysis capability. Section VI describes an ACV detection study. Section VII describes an empirical study of loops from DARPA challenge apps and open source projects. Section VIII discusses related research. Section IX concludes the paper.

III. 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 IV-C.

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 IV-C, 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 [13]. 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 [13]. 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.

IV. 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

A. Loop Control Variable Attributes

A Loop Control Variable (LCV) is a variable that influences any termination condition of a loop. LCVs subsume induction variables [9], 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.

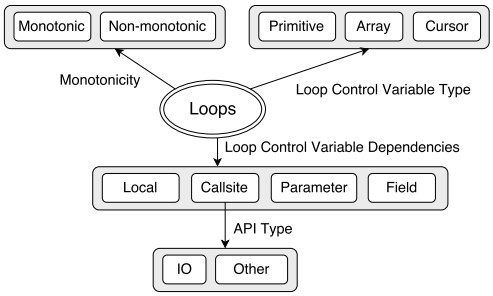

Fig. 1: Loop attributes used for characterizations

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 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'.

B. 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.

C. 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 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 VII these 24 LTPs cover a significant portion of the real-world loops.

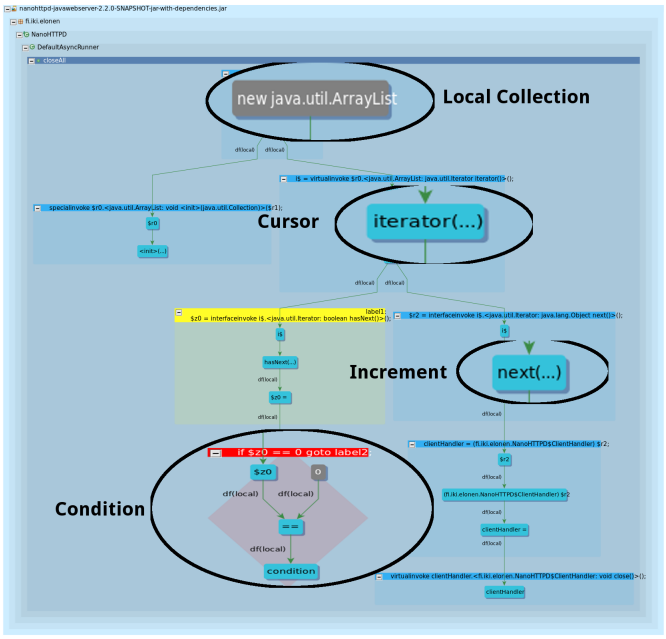

Fig. 2: An example of a loop termination pattern and its key parts

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 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 VI, the complex LTP can serve as a filter to find loops more likely to have ACVs.

D. 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 I lists the subsystems and examples of packages they contain.

E. Automatic generation of loop catalog

We use the Atlas program analysis platform [5] 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 [14] 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,

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.

TABLE I: JDK Subsystems.

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.

V. 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 III. We have also created a filtering framework that enables custom selection of loops using specified constraints on loop characteristics.

A. 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 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 V-B) and ad-hoc Atlas queries [5].

Termination Data Flow Smart View: Displays the Termination Dependence Graph (TDG) (Section III) 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 III) 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 [13], 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.

B. 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. The use of a loop filter to filter loops causing ACVs is illustrated in the case study in Section VI.

Currently, 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 IV-D). 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.

VI. Case study of AC vulnerability detection

We describe an ACV detection case study drawn from our research on the DARPA STAC program. For this case study, we describe the following steps in our approach to detect ACVs: (1) an automated analysis to characterize loops and generate a loop catalog, (2) using the catalog to apply filters to select loops, and (3) scrutinizing the selected loops to detect an ACV.

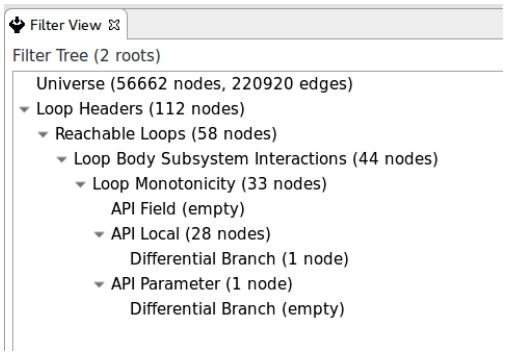

Fig. 3: Using filters to select a loop likely to have an ACV

App Description: *Gabfeed3* is a web forum software which allows users to post messages and search posted messages. The application utilizes a custom merge sort for sorting messages. The application consists of 23882 lines of Jimple, an intermediate representation of Java bytecode.

A. Phase I: Automated Generation of Loop Catalog

A loop catalog (Section IV-E) consisting of the following information about each loop is generated: (1) the TDG and LPCG abstractions, (2) whether it is monotonic, (3) applicable termination patterns, (4) subsystem APIs and the control flow paths on which they are invoked in the loop, and (5) structural characteristics such as the number of nesting levels and the inter-procedural nesting depth for loops in which the nesting is split across multiple functions. This information can be used to create a variety of filters to select loops. This case study of the Gabfeed3 application illustrates five key filters. Using these filters, we were able to isolate one loop out of 112 loops which required further scrutiny. On further scrutiny, an ACV was detected in this loop.

B. Phase II: Automated Filtering of Loops

The goal is to select a subset of loops that are likely to have an ACV. We use the Filter View (described in Section V-B) that enables creation of custom filters using information in the loop catalog. Application of the filters selects a subset of loops for further scrutiny.

The analyst uses a high-level understanding of the app to create appropriate combination of filters. In Gabfeed3, the high-level understanding is that it is a web application, it contains a server component to handle user requests, and thus only the loops reachable from this server component are likely to contain an ACV.

In this case study, the combination of filters used is shown in Figure 3. Gabfeed3 has a total of 112 loops in it. We apply the filters as follows,

- Reachable Loops: First, the analyst applies constraints based on the definition of an ACV: it should be possible for the attacker to control the loops that cause an ACV. As Gabfeed3 is a web application, analyst selects only loops reachable from the web entry point. *#Loops selected - 58 out of 112*
- Loop Body Subsystem Interactions: Second, the analyst applies the domain knowledge that all user input is embedded in a `HttpExchange` object. This prompts the analyst to select loops that use data extracted from this object via

getters, setters, or Java APIs. Thus, the analyst selects loops which interact with a Java API (JAVACORE Subsystem, see Section IV-D). *#Loops selected - 44 out of 58*

- Loop Monotonicity: As termination of non-monotonic loops is more complex than that of monotonic loops, the analyst chooses to select non-monotonic loops using the loop monotonicity filter for further analysis. *#Loops selected - 33 out of 44*
- Loop Termination Pattern: Loops whose termination have inter-procedural dependencies are complex, and typically involve invocation of APIs such as IO or other APIs in the app. Based on this knowledge, the analyst decides to apply the filters that select loops whose termination depends on other APIs in the app. *#Loops selected - 29 out of 32*
- Differential Branch: Finally, knowing that Gabfeed3 stores messages in collections, the analyst applies the differential branch filter to select loops with at least one differential branch influenced by size of a collection. *#Loops selected - 1 out of 28*

Through the filtering process, the analyst has thus selected one loop as a candidate for an ACV, namely `Sorter.changingSortHelper.label1`. He now uses the smart views to scrutinize this loop interactively.

C. Phase III: Detecting ACV by Interactive Scrutiny of Selected Loops

Analyst focuses on the one loop selected using the filters. Because the selected loop has a differential branch associated with it (as indicated by the filters applied in Phase II), the analyst decides to analyze the paths and the behaviors created by this differential branch condition. For that he applies Loop Projected Control Flow Smart View (described in Section V-A) which shows only behaviors of the loop relevant to loop's termination. Figure 4 displays the Loop Projected Control Flow Smart view for the loop - `Sorter.changingSortHelper.label1`. Of the four branch nodes in this loop, only the node labeled DB is a differential branch node. The analyst notices that two different behaviors induced by this branch are based on whether the size of the input is a multiple of eight or not. There is no apparent reason for this distinction and the analysts suspects that it could be a potential ACV.

A closer inspection of the code (available on our curated repository of ACV loops [6]) reveals that if the size of collection is multiple of eight then the loop exhibits quadratic or worse behavior and O(`nlogn`) otherwise. This creates an ACV which an attacker can trigger. We were able to confirm this ACV using dynamic analysis.

Remark: The specific filters and the use of Smart Views would vary depending on the high-level knowledge about the app and how the analyst chooses to apply that knowledge. The above case study illustrated one combination of filters that lead to detection of a particular ACV.

VII. An Empirical Study with 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,

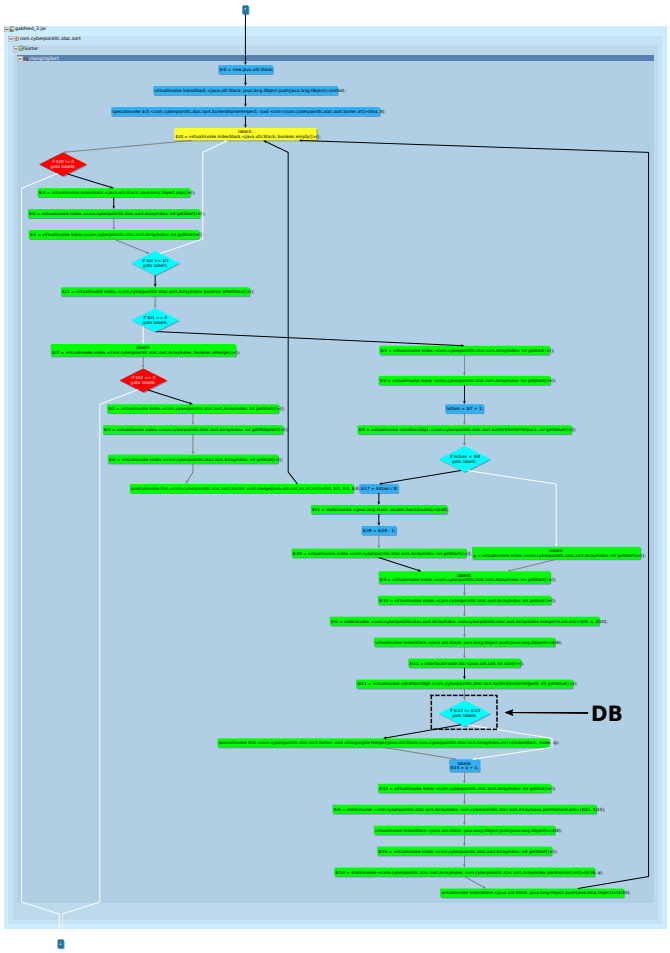

Fig. 4: LPCG shows branch nodes including the termination node in red and the differential branch node (labeled DB) with the ACV on one branch.

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 (of which Gabfeed3 is an app presented as a case study) 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*.

A. Usefulness of Loop Termination Patterns (LTPs)

We evaluate usefulness of LTPs with respect to the following three questions about usefulness of LTPs: (1) What

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

TABLE II: Distribution of C-loops and L-loops with respect to number of termination conditions (TC).

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

TABLE III: Distribution of C-loops and L-loops with respect to number of paths in the loop body.

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

TABLE IV: Distribution of C-loops and L-loops with respect to cyclomatic complexity.

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.

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.

B. 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% *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 II 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 III 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.

Cyclomatic Complexity: As a comparison to our other loop complexity measures we study the cyclomatic complexity [15], 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 IV is computed by using the control flow graph for each loop. The path metric in TableIII is also computed using the same set of graphs. The cyclomatic complexity is an approximation of our path metric.

C. 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.

VIII. 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 [16], [17], [18] 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 a 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 [19], [20], [21], [22] have been used for several applications including WCET (worst-case execution time). Many techniques have been proposed to summarize the loop effect [7], [23], [24], [25] using symbolic analysis. LESE [26] aim to compute iteration count precisely using symbolic execution and use it to infer the loop effect. The technique in [24] generates pre- and post-conditions as loop summaries using dynamic symbolic execution. These techniques focus on single-path loops. Proteus [7] classifies 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) [27], [28] have been traditionally used to abstract data and control dependences of a program. The information in the TDG and LPCG abstractions collectively contain all the information in the PDG that is relevant to the termination of a loop. The TDG uses a kind of taint analysis for reaching definitions [29] 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 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 [30]. 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.

IX. Conclusions

The paper describes foundational research and tools to automatically characterize and interactively reason about loops with an overarching goal of detecting ACVs that emanate from loops. We illustrate use of our analyses and tool chain to detect an ACV in a DARPA challenge app. We present an empirical study to bring out the usefulness of our characterizations in real-world software.

With the major exception of work by Waters [16], the

prevalent research is mostly focused on developing completely automated analyses for algorithmic complexity. However, automatically proving termination of an arbitrary loop is equivalent to the *halting problem*. To the best of our knowledge, this is the first work that that has proposed an integrated human-in-the-loop approach to loop analysis with novel abstractions, patterns, characterizations, and interactive visual querying mechanisms.

A human-in-the-loop approach is aimed at amplifying the intelligence [12] of a human analyst to enable him to detect vulnerabilities. This poses quite different demands on automation, and our contributions provide a first step to address the challenges posed. The automation needs to offer the freedom to the analysts to compose different analyses to drill deeper selectively. In the context of open-ended possibilities for ACVs, the composability is especially critical where the analysis must be adapted to pursue the possibilities that the analyst deems plausible. To this end, our tools enable: (a) creation of filters based on automatically generated characterizations, and (b) visual querying mechanism designed to compose relevant program artifacts interactively.

In the future, we plan to undertake user studies to evaluate how our interactive tools actually impacted the ability of analysts to audit and detect ACVs in the challenge apps.

ACKNOWLEDGMENTS

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

REFERENCES

[1] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003, pp. 3–3.

[2] 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.

[3] DARPA, "Space / Time Analysis for Cybersecurity (STAC)," Information Innovation Office, Arlington, VA, Tech. Rep., 2014.

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

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591065

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

[7] 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*. ACM, 2016, pp. 61–72.

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

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

[10] 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

[11] 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.

[12] F. P. Brooks, Jr., "The computer scientist as toolsmith ii," *Commun. ACM*, vol. 39, no. 3, pp. 61–68, Mar. 1996. [Online]. Available: http://doi.acm.org/10.1145/227234.227243

[13] A. Tamrawi and S. Kothari, "Projected control graph for accurate and efficient analysis of safety and security vulnerabilities," in *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 2016, pp. 113–120.

[14] T. Wei, J. Mao, W. Zou, and Y. Chen, "A new algorithm for identifying loops in decompilation," in *International Static Analysis Symposium*. Springer, 2007, pp. 170–183.

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

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

[30] 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.