DISCOVER: Detecting Algorithmic Complexity Vulnerabilities[*]

Payas Awadhutkar
Iowa State University
Ames, Iowa, USA
payas@iastate.edu

Ganesh Ram Santhanam
EnSoft Corp.
Ames, Iowa, USA
ganeshramsanthanam@ensoftcorp.com

Benjamin Holland
Suresh Kothari
bholland@iastate.edu
Iowa State University
Ames, Iowa, USA
kothari@iastate.edu

ABSTRACT

Algorithmic Complexity Vulnerabilities (ACV) are a class of vulnerabilities that enable Denial of Service Attacks. ACVs stem from asymmetric consumption of resources due to complex loop termination logic, recursion, and/or resource intensive library APIs. Completely automated detection of ACVs is intractable and it calls for tools that assist human analysts.

We present DISCOVER, a suite of tools that facilitates human-on-the-loop detection of ACVs. DISCOVER's workflow can be broken into three phases - (1) Automated characterization of loops, (2) Selection of suspicious loops, and (3) Interactive audit of selected loops. We demonstrate DISCOVER using a case study using a DARPA challenge app. DISCOVER supports analysis of Java source code and Java bytecode. We demonstrate it for Java bytecode.

Demo Video: https://youtu.be/LtaOYxo7AWI
Tool:https://ensoftcorp.github.io/loop-comprehension-toolbox

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Algorithmic Complexity Vulnerability, Cybersecurity, Software Analysis

ACM Reference Format:
Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. 2019. DISCOVER: Detecting Algorithmic Complexity Vulnerabilities. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341177

[*]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.

1 INTRODUCTION

Algorithmic Complexity Vulnerabilities (ACV) are a class of vulnerabilities that can be exploited to mount a denial of service (DoS) attack. [5]. 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." [14]. 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.

Listing 1: XML-bomb

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol↩
     ;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&↩
     lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&↩
     lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&↩
     lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&↩
     lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&↩
     lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&↩
     lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&↩
     lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&↩
     lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Let's illustrate an ACV with an example. We will use the example of a billion laughs attack on an XML parser [1]. A standard XML file contains a "contents section" with instances of predefined entities 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 by value 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. Unchecked expansion allows an attacker to specify a huge XML document using a small number of entities by repetitively referencing the entity definitions. Listing 1 shows a standard exploit [1]. 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. It contains only one instance of `lol9`. The parser will expand it to 10 `lol8`s, each of which is expanded to 10 `lol7`s and so on. It results into 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. 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. 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 1, then Word will attempt to load the expanded document in the memory, and in most cases will hang. At the time of writing this paper, Word still contains this ACV.

In recent years, exploits using ACVs are on the rise. Crosby and Wallch [5] coined the term ACV in 2003 and theorized an attack on hash tables. In 2011, Klink and Walde [12] 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 [2]. It is imperative that we find ways to mitigate the risks posed by ACVs and hence develop technology to detect ACVs.

This motivated the DARPA Space/Time Analysis for Cybersecurity (STAC) program [6]. It called for a novel human-on-the-loop approach to detect ACVs as completely automated detection of ACVs is intractable [6]. The program artifacts that may contain an ACV include loops, recursion, or resource-intensive library APIs [8]. We focus on loops and library calls. As completely automated and precise analysis of loop termination is intractable, ACVs need automated analyses which can assist human analysts. ACVs also require tools that can help the analyst visualize interactions with library calls. In this paper, we present the tool DISCOVER, the output of our experience on the STAC program. DISCOVER is a suite of loop analysis tools which are geared at detecting ACVs. Its automated loop characterizations help in filtering the loops and library calls likely to lead to an ACV, while its interactive capabilities help a human to verify the presence of an ACV in the filtered program artifacts.

The rest of the paper is organized as follows. Section 2 describes the workflow used by analysts to detect ACVs using DISCOVER. Section 3 describes infrastructure of DISCOVER. Section 4 describes a case study to demonstrate DISCOVER.

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 [4].
(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.

Table 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.
Collection	java.util.collection
Stream	java.util.stream
Iterator	java.util.Iterator
Spliterator	java.util.Spliterator
Functional	java.util.Function

(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.

3 DISCOVER INFRASTRUCTURE

DISCOVER was developed using Atlas [7] and is capable of analyzing Java bytecode and Java source code. It uses Soot [18] to convert Java bytecode into Jimple for analysis. DISCOVER infrastructure is divided into four parts: (1) Loop Abstractions, (2) Subsystems, (3) Loop Catalog, and (4) Interactive Views.

3.1 Loop Abstractions

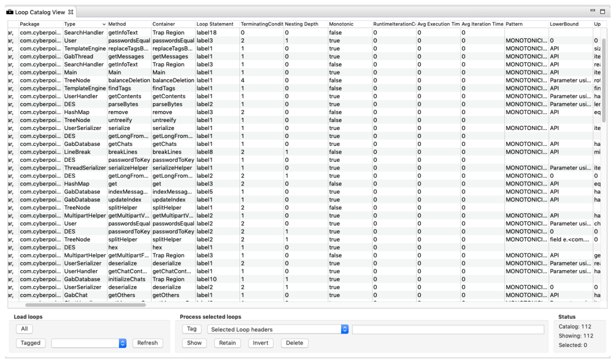
DISCOVER uses two Loop Abstractions to characterize loops: Termination Dependence Graph (TDG) and Loop Projected Control Graph (LPCG). [4]

Termination Dependence Graph (TDG): TDG of a loop is an intraprocedural data flow slice which captures: (a) the data flow that influences the termination condition of the loop, (b) modification of data within the loop. A summary of the interprocedural data flow dependencies is computed along with the TDG to capture the complete picture.

This abstraction serves as the foundation for developing loop termination patterns, each pattern implying a specific mode of termination for the loop. TDG is used to compute Loop Monotonicity [4] which is a complexity metric for loop termination. Loop Termination Patterns implemented in DISCOVER are defined using Loop Monotonicity.

Loop Projected Control Graph (LPCG): LPCG of a loop is a compact representation of relevant control flow within the loop. The relevant control flow includes loop termination, loop control variables and callsites within the loop.

The compaction is derived from *Projected Control Graph* (PCG), a projection of the CFG that retains only the relevant execution behaviors and elides duplicate paths. [16]. A mathematical definition of the PCG and an efficient algorithm to compute PCGs



	Package	Type	Method	Container	Loop Statement	TerminatingCondit	Nesting Depth	Monotonic	RuntimeIterationC	Avg Execution Tim	Avg Iteration Time	Pattern	LowerBound	Up
ar,	com.cyberpoi...	SearchHandler	getInfoText	Trap Region	label18	0	0	false	0	0	0			
ar,	com.cyberpoi...	User	passwordsEqual	passwordsEqual	label3	2	1	true	0	0	0	MONOTONICI...	0	0
ar,	com.cyberpoi...	TemplateEngine	replaceTagsB...	replaceTagsB...	label1	1	0	true	0	0	0	MONOTONICI...	API	siz
ar,	com.cyberpoi...	GabThread	getMessages	getMessages	label1	1	0	true	0	0	0	MONOTONICI...	API	ite
ar,	com.cyberpoi...	SearchHandler	getInfoText	Trap Region	label3	1	0	false	0	0	0	MONOTONICI...	API	rea
ar,	com.cyberpoi...	Main	Main	Main	label1	1	0	true	0	0	0	MONOTONICI...	API	ite
ar,	com.cyberpoi...	TreeNode	balanceDeletion	balanceDeletion	label1	4	0	false	0	0	0	MONOTONICI...	Parameter usi...	rot
ar,	com.cyberpoi...	TemplateEngine	findTags	findTags	label1	1	0	false	0	0	0	MONOTONICI...	API	fin
ar,	com.cyberpoi...	UserHandler	getContents	getContents	label1	1	0	true	0	0	0	MONOTONICI...	Parameter usi...	ha
ar,	com.cyberpoi...	DES	parseBytes	parseBytes	label2	1	0	true	0	0	0	MONOTONICI...	Parameter usi...	ler
ar,	com.cyberpoi...	HashMap	remove	remove	label3	2	0	false	0	0	0	MONOTONICI...	API	eq
ar,	com.cyberpoi...	TreeNode	untreeify	untreeify	label1	1	0	false	0	0	0			
ar,	com.cyberpoi...	UserSerializer	serialize	serialize	label1	1	0	true	0	0	0	MONOTONICI...	API	ite
ar,	com.cyberpoi...	DES	getLongFrom...	getLongFrom...	label1	1	0	true	0	0	0			
ar,	com.cyberpoi...	GabDatabase	getChats	getChats	label1	1	0	true	0	0	0	MONOTONICI...	API	ha
ar,	com.cyberpoi...	LineBreak	breakLines	breakLines	label8	2	1	false	0	0	0	MONOTONICI...	API	mi
ar,	com.cyberpoi...	DES	passwordToKey	passwordToKey	label1	1	0	true	0	0	0			
ar,	com.cyberpoi...	ThreadSerializer	serializeHelper	serializeHelper	label1	1	0	true	0	0	0	MONOTONICI...	Parameter usi...	ite
ar,	com.cyberpoi...	DES	getLongFrom...	getLongFrom...	label2	2	1	true	0	0	0	MONOTONICI...	0	0
ar,	com.cyberpoi...	HashMap	get	get	label3	2	0	false	0	0	0	MONOTONICI...	API	eq
ar,	com.cyberpoi...	GabDatabase	indexMessag...	indexMessag...	label5	1	0	true	0	0	0	MONOTONICI...	API	ha
ar,	com.cyberpoi...	GabDatabase	updateIndex	updateIndex	label1	1	0	true	0	0	0	MONOTONICI...	API	ha
ar,	com.cyberpoi...	TreeNode	splitHelper	splitHelper	label1	2	0	false	0	0	0			
ar,	com.cyberpoi...	MultipartHelper	getMultipartV...	getMultipartV...	label2	1	0	true	0	0	0	MONOTONICI...	API	ha
ar,	com.cyberpoi...	User	passwordsEqual	passwordsEqual	label2	2	0	true	0	0	0	MONOTONICI...	Parameter usi...	ch
ar,	com.cyberpoi...	DES	passwordToKey	passwordToKey	label2	2	1	true	0	0	0	MONOTONICI...	0	0
ar,	com.cyberpoi...	TreeNode	splitHelper	splitHelper	label2	2	1	true	0	0	0	MONOTONICI...	field e.<com....	0
ar,	com.cyberpoi...	DES	hex	hex	label1	1	0	true	0	0	0			
ar,	com.cyberpoi...	MultipartHelper	getMultipartF...	Trap Region	label3	1	0	false	0	0	0	MONOTONICI...	API	ge
ar,	com.cyberpoi...	UserSerializer	deserialize	deserialize	label1	2	0	true	0	0	0	MONOTONICI...	Parameter usi...	rea
ar,	com.cyberpoi...	UserHandler	getChatCont...	getChatCont...	label1	1	0	true	0	0	0	MONOTONICI...	Parameter usi...	ha
ar,	com.cyberpoi...	GabDatabase	initializeChats	Trap Region	label10	1	1	true	0	0	0			
ar,	com.cyberpoi...	UserSerializer	deserialize	deserialize	label2	2	1	true	0	0	0	MONOTONICI...	0	0
ar,	com.cyberpoi...	GabChat	getOthers	getOthers	label1	1	0	true	0	0	0	MONOTONICI...	API	ha

Catalog: 112
Showing: 112
Selected: 0

Figure 1: Loop Catalog View

are presented in [17]. DISCOVER uses COMB, a toolbox of PCG utilities [11].
LPCG [4] is a PCG with following considered as *relevant*: (a) Loop Header, (b) Loop Termination Conditions, and (c) data flow and callsites captured within TDG. LPCGs are useful to understand the loop termination logic and can thus be used by a human to isolate paths with asymmetric resource consumption.

3.2 Subsystems

Dietrich et al [8] point out that ACVs can also be caused due to resource intensive library APIs. This effect is amplified if the APIs are present in loops. High-level understanding of library APIs also helps in characterizing the functionality of loops and help analysts hypothesize ACVs. For example, a loop containing a call to an I/O API may be reading or writing to a file. If it is performing an unchecked write operation, then it may end up consuming a large amount of memory.
With this motivation, we categorized JDK APIs into *subsystems*. A total of 24 subsystems are codified into DISCOVER. Table 1 lists the subsystems and examples of the packages of APIs they contain.

3.3 Loop Catalog

DISCOVER uses the loop abstractions to characterize every loop in the app. This captured information is collected in what is referred to as the *Loop Catalog*. Loop Catalog consists of the following information:

- Source Correspondence: Location of the loop in the app.
- Loop Monotonicity: Whether the loop termination is complex or not.
- Loop Termination Pattern: A triple of {Monotonicity, Type of control variable, Dependency of control variable }. These patterns imply modes of termination. For example, a monotonic primitive local loop is of the form `for(i=0;i<n;i++)`, implying a simple termination behavior. Loop Catalog also reports conservative bounds on the number of iterations.
- Reachability: Whether the loop is reachable from the entry point to the app or not.
- Loop Body Description: This includes information on the branches contained within the loop, branches governing the loop, and lambda expressions (which may hide loops) contained within the loop.
- Subsystem Interactions: The subsystems that were invoked within the loop.

3.4 Interactive Views

In order to make all this information easily accessible to the analyst, we developed interactive views which aid in ACV detection. These views fall into two categories, Loop Catalog View and Smart Views. Loop Catalog View provides access to the information captured within the loop catalog. Figure 1 shows the loop catalog view. The captured information is presented in a spreadsheet format. Each row in this view corresponds to a loop in the app. Each column corresponds to a characteristic of the loop that can be used as a filter to select loops. The view allows the analyst to select loops of interest by clicking on the corresponding rows. Selection of multiple loops is allowed. The analyst can either retain or delete the selected loops. Additionally, they can tag the selected loops to refer them later.
Smart views are part of the interactive visualization infrastructure provided by Atlas [7]. These can be used to invoke canned analyses to on-the-fly. DISCOVER includes smart views which can compute: (1) TDG, (2) LPCG, (3) Loop Body, (4) Loop Subsystem Interactions. We note that these views can interact with each other and can be composed together to audit apps to detect ACVs.

4 CASE STUDY

We present a case study to illustrate how to detect ACVs in an app called Gabfeed_3, developed as a challenge for the STAC program, using DISCOVER. The source code of the app, along with other challenge apps, is available on GitHub [15]. 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 using a custom merge sort. 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 publically available on GitHub [15] and DARPA plans to release the remaining apps in the near future.

4.1 Phase 1: Generate Loop Catalog

DISCOVER automatically characterized and cataloged all loops in the app in the Loop Catalog. Gabfeed_3 consists of 112 loops.

4.2 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.

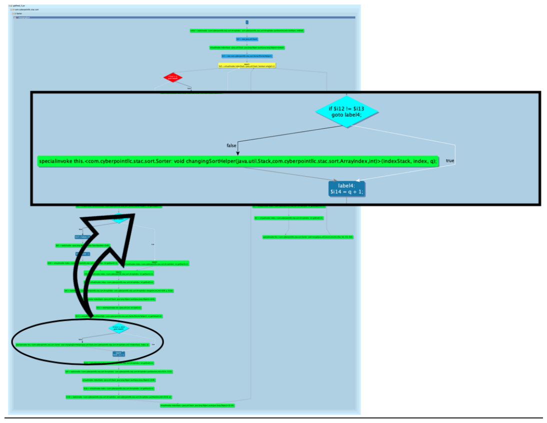

if $i12 != $i13
goto label4;
false
true
specialinvoke this.<com.cyberpointllc.stac.sort.Sorter: void changingSortHelper(java.util.Stack,com.cyberpointllc.stac.sort.ArrayIndex,int)>(indexStack, index, q);
label4:
$i14 = q + 1;

Figure 2: LPCG of the vulnerable loop. The zoomed in branch creates asymmetry with only one with an expensive operation

- 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 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 to search for paths 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 shown in Figure 2. The LPCG reveals the presence of a differential branch. This branch (zoomed in Figure 2) 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.

5 RELATED WORK

This paper is based on our prior work [4]. For a detailed list of related works, we refer to [4], now we will summarize only the most relevant ones.

There is a multitude of techniques available which are aimed at precisely summarizing loops [10, 13, 20]. To assess the usefulness of the existing techniques, we performed the following experiments. We curated 15 representative loop snippets from the challenge apps provided by DARPA. These snippets are available on GitHub [3]. We tried to summarize these loops using Proteus [20] that received the 2016 Distinguished FSE Paper award. None of the 15 loops could be precisely summarized by Proteus. We think that these loops can be used to further improve the existing formal verification approaches and loop summarization techniques. CLAPP [9] has a similar approach to us and can identify loops with calls to a set of high-risk APIs as labeled by Android developers. CLAPP is designed for Android code and not for arbitrary Java code. Also CLAPP does not support interactive audits to facilitate human-on-the-loop approach, which is critical to detect ACVs.

REXPLOITER [19] the only other tool we are aware of that is specifically aimed at detecting ACVs. REXPLOITER detects ACVs by identifying regular expressions that match the vulnerable input strings. These regular expressions are extracted using an NFA-based algorithm. They have successfully employed this approach to detect ACVs in real-world apps. However, how does REXPLOITER fare when there is a singular input, for which the regular expression may not even exist, that triggers the ACV is not made sufficiently clear. Also, REXPLOITER does not provide any interactive capabilities which can make use of human domain knowledge that is often useful in the detection of ACVs.

6 CONCLUSION

Algorithmic Complexity Vulnerabilities (ACV) can lead to denial of service attacks. ACVs are rooted in loops, recursions, and/or resource-intensive library APIs with loops being the likeliest location. A completely automated solution to detect arbitrary ACVs is intractable.

We presented DISCOVER, a suite of tools developed on DARPA Space/Time Analysis for Cybersecurity (STAC) [6] program, that assists a human analyst to detect ACVs. Its interactive capabilities enable a human-on-the-loop audit workflow. We demonstrate DISCOVER using a case study from DARPA challenge apps.

ACKNOWLEDGMENT

We thank our colleagues at the Knowledge-Centric Software (KCS) Engineering Lab at Iowa State University and the colleagues at EnSoft for helping us with our research. Dr. Kothari is the founding President of EnSoft.

REFERENCES

[1] [n.d.]. XML Security: A Billion Laughs. https://cytinus.wordpress.com/2011/07/26/37/.

[2] Jean-Philippe Aumasson, DJ Bernstein, and M BOBLET. 2012. Hash-flooding DoS reloaded: attacks and defenses. In *29th Chaos Communications Congress*.

[3] Payas Awadhutkar. [n.d.]. Curated set of loops illustrating characteristics of loops that may lead to algorithmic complexity vulnerabilities. https://github.com/payas0awadhutkar/ACV-Loops-Repository.

[4] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. 2017. Intelligence Amplifying Loop Characterizations for Detecting Algorithmic Complexity Vulnerabilities. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 249–258.

[5] Scott A. Crosby and Dan S. Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. USENIX Association, 3–3.

[6] DARPA. 2014. *Space / Time Analysis for Cybersecurity (STAC)*. Technical Report. Information Innovation Office, Arlington, VA.

[7] Tom Deering, Suresh Kothari, Jeremias Sauceda, and Jon Mathews. 2014. Atlas: A New Way to Explore Software, Build Analysis Tools. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014)*. ACM, New York, NY, USA, 588–591. https://doi.org/10.1145/2591062.2591065

[8] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[9] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2015. CLAPP: characterizing loops in Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 687–697.

[10] Carlo A Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: Analysis, classification, and examples. *ACM Computing Surveys (CSUR)* 46, 3 (2014), 34.

[11] Benjamin Holland, Payas Awadhutkar, Suresh Kotharti, Ahmed Tamrawi, and Jon Mathews. 2018. Comb: Computing relevant program behaviors. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 109–112.

[12] Alexander Klink and Julian Walde. 2011. Efficient Denial of Service Attacks on Web Application Platforms. In *28th Chaos Communication Congress*.

[13] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. 2013. Loop summarization using state and transition invariants. *Formal Methods in System Design* 42, 3 (2013), 221–261.

[14] MITRE. [n.d.]. CWE-407: Algorithmic Complexity. https://cwe.mitre.org/data/definitions/407.html.

[15] Apogee Research. [n.d.]. Public release items for the DARPA Space/Time Analysis for Cybersecurity (STAC) program. https://github.com/Apogee-Research/STAC.

[16] Ahmed Tamrawi and Suresh Kothari. 2016. Projected Control Graph for Accurate and Efficient Analysis of Safety and Security Vulnerabilities. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*. IEEE, 113–120.

[17] Ahmed Tamrawi and Suresh Kothari. 2018. Projected control graph for computing relevant program behaviors. *Science of Computer Programming* 163 (2018), 93–114.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. IBM Corp., 214–224.

[19] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 3–20.

[20] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. 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, 61–72.