2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation

Statically-informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities

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

***Abstract*—Algorithmic Complexity (AC) vulnerabilities can be exploited to cause a denial of service attack. Specifically, an adversary can design an input to trigger excessive (space/time) resource consumption. It is not possible to build a fully automated tool to detect AC vulnerabilities. Since it is an open-ended problem, a human-in-loop exploration is required to find the program loops that could have AC vulnerabilities. Ascertaining whether an arbitrary loop has an AC vulnerability is itself difficult, which is equivalent to the halting problem.**

This paper is about a pragmatic engineering approach to detect AC vulnerabilities. It presents a *statically-informed dynamic* (SID) analysis and two tools that provide critical capabilities for detecting AC vulnerabilities. The first is a static analysis tool for exploring the software to find loops as the potential candidates for AC vulnerabilities. The second is a dynamic analysis tool that can try many different inputs to evaluate the selected loops for excessive resource consumption. The two tools are built and integrated together using the interactive software analysis, transformation, and visualization capabilities provided by the Atlas platform.

The paper describes two use cases for the tools, one to detect AC vulnerabilities in Java bytecode and another for students in an undergraduate algorithm class to perform experiments to learn different aspects of algorithmic complexity.

Tool and Demo Video: https://ensoftcorp.github.io/SID

I. Introduction

Algorithmic complexity (AC) vulnerabilities [1], [2] are rooted in the space and time complexities of externally-controlled execution paths with loops. In the AC vulnerability commonly known as the "billion laughs attack" or an XML bomb [3], parsing a specially crafted input file of less than a kilobyte creates a string of 10^9 concatenated "lol" strings requiring approximately 3 gigabytes of memory. At the extreme, a completely automated detection of AC vulnerabilities amounts to solving the intractable halting problem [4]. Existing tools for computing the loop complexity [5], [6] are limited and cannot prove termination for several classes of loops [7]. Based on our ongoing research for the DARPA Space/Time Analysis for Cybersecurity (STAC) [8] program, this paper describes a pragmatic engineering approach to detect AC vulnerabilities.

*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.

Our approach to detect AC vulnerabilities involves: (1) *Automated Exploration:* Identify loops, precompute their crucial attributes such as intra- and inter-procedural nesting structures and depths, and termination conditions. (2) *Hypothesis Generation:* Through an interactive inspection of the precomputed information the analyst hypothesizes plausible AC vulnerabilities and selects candidate loops for further examination using dynamic analysis. (3) *Hypothesis Validation:* The analyst inserts probes and creates a driver to exercise the program by feeding workloads to measure resource consumption for the selected loops.

Since detecting AC vulnerabilities is an open-ended problem, our approach strives to combine human intelligence with static and dynamic analysis to achieve scalability and accuracy. A lightweight static analysis is used for a scalable exploration of loops in bytecode from large software, and an analyst selects a small subset of these loops for further evaluation using a dynamic analysis for accuracy.

This paper describes *statically-informed dynamic* (SID) analysis powered by two tools: (a) a static analysis tool called the *loop call graph* (LCG) to explore loops, (b) a dynamic analysis tool called the *time complexity analyzer* (TCA) to measure the resource consumption of loops. The SID analysis is applied using the above 3-step process. The main contributions of the paper are:

- LCG as a compact visual model to explore intra- and inter-procedural nesting structures of loops. The LCG as an interactive *smart view* that updates in response to selections in the underlying source code or other built-in graph models in Atlas such as the control flow graph of a method. The LCG helps human analysts understand the intra- and inter-procedural relationships between loops and how they are triggered, which is crucial to hypothesize AC vulnerabilities.
- TCA as a dynamic analyzer that enables the analyst to automatically instrument the selected loops with resource usage probes. TCA generates a skeleton driver that an analyst can customize and run with a workload to observe the corresponding resource usage. It is laborious to extract an executable loop slice and run it separately to measure its resource usage [5]. TCA enables in situ evaluation of time complexity of loops.
- Case studies illustrating the use of LCG and TCA for (1) detecting an AC vulnerability in a web app; (2) educational

978-1-5090-3848-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SCAM.2016.23

IEEE computer society

Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.

use to learn different aspects of algorithmic complexity.

The work represents a major engineering effort which includes putting together complex static analyses for detecting loops in Java bytecode and computing the crucial loop attributes and nesting structures, instrumenting the Java bytecode for runtime measurements of time complexity, creating drivers to apply different strategies to select inputs, and plotting the time complexity. These individually complex capabilities are integrated and equipped with an interactive and compact visualization so that the analyst can experiment with and understand the structures and behaviors of loops to detect AC vulnerabilities. The LCG and TCA tools are built and integrated using the Atlas platform [9], specifically its APIs for static analysis, program transformations, and interactive program graphs.

II. SID Analysis Tools

This section describes LCG and TCA as the two key enabling tools for SID analysis. LCG helps the analyst to perform automated analysis to explore loops reachable from a method (via a call chain) and select candidates for AC vulnerabilities. After the analyst hypothesizes a vulnerability, TCA enables the analyst to instrument the code, and perform dynamic analysis to confirm or refute a hypothesis by actually observing the resource consumption for the candidate loops. LCG operates on Java source and bytecode, while TCA instruments Java bytecode.

A. Loop Call Graph

The *Loop Call Graph* (LCG) is a succinct representation of crucial information necessary to hypothesize AC vulnerabilities, namely the inter- and intra-procedural nesting structure of loops and how they are reached through a call chain. LCG represents the information compactly as a multi-attributed directed graph. The nodes in the LCG are methods containing loops and methods that call other methods which also contain loops. There is an edge from method m_1 to method m_2 in the LCG if m_1 calls m_2. Further, nodes and edges have color attributes. A node is colored blue if it contains loops, and grey otherwise. An edge is colored yellow if the callsite of m_2 is located within a loop in m_1 (indicates that loops in m_2 are inter-procedurally nested), and black otherwise. Explicit loops as well as loops due to recursion are identified in the LCG. The accuracy of LCG depends on the accuracy of the underlying call graph construction algorithm. Our implementation of LCG allows the flexibility to use different algorithms to manage the accuracy by a suitable choice of an algorithm. Typically, we use a class hierarchy analysis for computing the call graph.

Preprocessing. The loops in the bytecode are identified using the DLI algorithm [10]. The information about loops is computed and added to the program graph database stored by Atlas [9]. This information is used when the analyst invokes the LCG smart view.

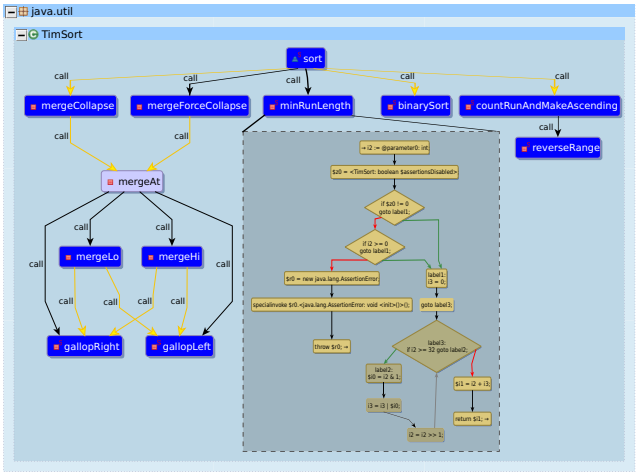

Figure 1: LCG smart view for TimSort along with the CFG smart view (shown as a callout) obtained by clicking on the `minRunLength` method in the LCG. In the CFG, darker shades are used to show deeper loop nesting. LCG shows 11 methods in `TimSort` that contain loops.

LCG Smart View. The LCG *smart view* is an interactive tool that enables the analyst to explore and understand loops nested inter-procedurally.

First, the view itself serves as succinct visual index of loops and their inter-procedural nesting. This is essential for the analyst to hypothesize AC vulnerabilities because if m_1 declares a loop l, and m_1 is called from a loop l' in m_2, LCG helps the analyst to reason that l can be nested in l' in that calling context.

Second, the LCG smart view is interactive. The LCG smart view inherits 2-way source correspondence from Atlas. The analyst can double click on a method in the LCG smart view to go to the code for that method. Alternately, when an analyst clicks on a method name in the source code window, the LCG window instantly updates to show the LCG for the selected method. Since the LCG for real-world programs can be large, to scale the visualization and make large graphs comprehensible, the smart view allows users to incrementally expand and collapse nodes at any level of the graph, and offers a textual search to easily locate methods.

Third, the LCG smart view seamlessly integrates with other Atlas smart views (e.g., control flow and call graph smart views). The analyst can compose analyses by feeding a selection in the LCG smart view as input to another Atlas smart view. For example, if the analyst opens the control flow smart view alongside the LCG smart view and clicks on a method in the LCG, the method's control flow graph (CFG) is shown in another window. This is illustrated in Figure 1. Along with the LCG view for `TimSort` [11] it shows the CFG for a method selected in the LCG window. Overall, the LCG helps the analyst to explore and understand the loops interactively in order to hypothesize AC vulnerabilities.

Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.

B. Time Complexity Analyzer

The analyst can click on a LCG node to select a method and then click the TCA button to perform a dynamic analysis on loops in the selected method. TCA automates the insertion of probes into bytecode, and records the runtime measurements for loops. TCA supports two kinds of probes:

1) *Iteration Counters*. An iteration counter tracks the number of times a loop header (entry point of a loop) is executed. Iteration counter measurements are platform independent and repeatable for deterministic code.
2) *Wall Clock Timers*. A wall clock timer uses timestamps to measure the cumulative time spent in a loop. Time-based measurements, although noisy and inaccurate [12], are nevertheless useful. For example, caching or garbage collection side effects on the runtime are captured by the time measurements but not through iteration counters.

Challenges in Inserting Probes. TCA instruments Jimple [13] (an intermediate representation for Java bytecode) produced from the bytecode of the application. TCA computes the list of probes to be inserted and the corresponding offsets (locations within the method) where they should be inserted. The challenge is that the insertion of a probe changes the Jimple code, and hence changes the offsets for the probes yet to be inserted. To address this challenge, TCA keeps track of the inserted probes, and recomputes the offset for inserting each new probe. Another challenge is to allow the analyst the flexibility to run in parallel different dynamic analysis experiments, each with their own set of probes on possibly different sets of loops. TCA must save and reassemble the instrumented Jimple into bytecode to run each dynamic analysis. To allow multiple dynamic analyses in parallel, TCA clones the original project and saves the updated, instrumented Jimple to the cloned project, while preserving the original project being analyzed. The resulting Jimple code is reassembled into bytecode using Soot [13].

Automation for Generating Driver Skeleton. To allow the analyst to feed custom inputs for running the dynamic analysis, TCA automatically generates a skeleton *driver* program. The driver skeleton contains code to invoke the instrumented bytecode to collect probe measurements. The analyst updates the driver logic with a relevant workload. For example, to measure a sort method's complexity, an analyst may choose the workload as arrays of different sizes containing integers selected from a random distribution.

Automation for Complexity Reports. When the driver is executed, it invokes the instrumented bytecode that collects the measurements made by the probes for each workload. For measurements collected via the wall clock timer, TCA produces a plot of the workload size versus the measured time. For measurements collected via the iteration counters, TCA includes an engine for calculating a regression fit of the observed number of iteration counts. Similarly to existing tools [14], [12], TCA uses linear or power-law models to fit the observed measurements and estimates empirical complexity. In particular, TCA plots a graph of actual measurements against the workload size on a log-log scale using JFreeChart. The plot also reports the slope and the R^2 error for the best fit obtained by the linear regression engine (e.g., see Figure 3). A slope of m on the log-log plot indicates the measured empirical complexity of n^m.

III. Detecting AC Vulnerability Using SID Tools

This section describes a case study of how an algorithmic complexity (AC) vulnerability in a web app is detected by SID analysis using LCG and TCA. The case study app and the underlying vulnerability are modeled after a DARPA challenge from the Space/Time Analysis for Cybersecurity (STAC) program [8]. The analysis for the case study was done on the bytecode, but we include here some snippets of decompiled code for the reader's benefit.

App functionality: *Blogger* is a web application whose Java web server implementation extends the NanoHTTPD open source project. The app's functionality includes user sign in, creation of new blog posts and viewing of posts by other users. The app consists of 2320 lines of Jimple code (translates to 1800 lines of decompiled Java code).

A. Phase I. Automated Exploration

The loops in the Blogger app, their nesting depths and the calling relationships between methods containing loops are precomputed in Phase I of SID analysis.

B. Phase II. Hypothesis Generation with LCG

Domain knowledge for generating hypothesis. The analyst begins auditing with some domain knowledge about web server applications. A server application typically has one or more threads (server listener) dedicated to listening for client connections, and one or more threads (client request handler) for parsing, processing and responding to client requests.

Starting with this domain knowledge, the analyst hypothesizes an AC vulnerability to be in the client request handler thread. For the possibility of an AC vulnerability in the server listener, see Remark 1 at the end of this section. By scanning the source code that initiates the threads, the analyst identifies that each client request is processed in a separate thread of type `ClientHandler`.

Using LCG to identify loops reachable from a method. The analyst selects `ClientHandler.run` and invokes the LCG smart view. As shown in Figure 2, the view shows a graph of methods containing loops and starting from `ClientHandler.run`. Nodes and edges of the LCG are color coded as described earlier (Section II-A). For example, the edge from `ClientHandler.run` to `HTTPSession.execute` is yellow because the callsite to the latter is in a loop. As seen from Figure 2, the LCG shows 16 methods containing loops. Note that each method may contain multiple loops. The analyst formulates the following hypothesis:

> Hypothesis. *An AC vulnerability is caused by one or more loops in the 16 methods in the LCG.*

Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.

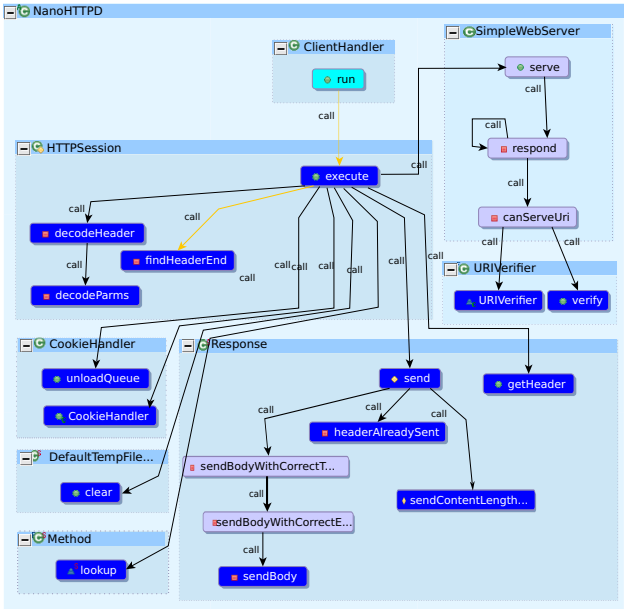

Figure 2: LCG for the Blogger app

Loop Headers by Method	Iterations
NanoHTTPD.HTTPSession.findHeaderEnd.label1	4341
URIVerifier.verify.label5	2148
URIVerifier.verify.label3	1188
URIVerifier.verify.label1	1074
URIVerifier.URIVerifier.label5	270
URIVerifier.URIVerifier.label1	190
NanoHTTPD.HTTPSession.decodeHeader.Trap Region.label10	100
NanoHTTPD.Response.headerAlreadySent.label1	24
NanoHTTPD.CookieHandler.CookieHandler.label1	22
NanoHTTPD.Response.sendBody.label3	22
NanoHTTPD.HTTPSession.decodeParms.label2	16
NanoHTTPD.ClientHandler.run.Trap Region.label2	15
NanoHTTPD.Response.send.Trap Region.label6	12
NanoHTTPD.CookieHandler.unloadQueue.label1	12
NanoHTTPD.Response.getHeader.label1	12
NanoHTTPD.HTTPSession.execute.Trap Region.label6	11
NanoHTTPD.DefaultTempFileManager.clear.label1	11
NanoHTTPD.Method.lookup.label1	11

Table I: Iteration counts for loops exercised in the Blogger app when the URLs "/", "/test" and "/stac/example/Example" are requested from a browser.

C. Phase III. Hypothesis Validation with TCA

TCA is used to confirm or reject the hypothesis as follows.

Probe instrumentation and dynamic analysis using TCA. The analyst selects all methods containing loops in the LCG and invokes the TCA menu option to instrument the loops in the selected methods with the *iteration counter* probe to compare resource consumption of various loops. TCA then compiles the instrumented bytecode for the server into an executable. The analyst starts the server (instrumented server code), and passes three simple HTTP request URLs to the server. Finally, TCA reports the number of iterations of each loop in the LCG to the analyst (see Table I).

Follow-up on TCA results. Table I shows that after the server processes three simple request URLs, two methods show an unusually large number of loop iterations – `HTTPSession.findHeaderEnd` and `URIVerifier.verify`. In particular, `URIVerifier.verify` has three loops, so the analyst clicks on the method in the LCG to go to its code (Listing 1). The method has three loops at lines 5, 9, 11 respectively. The `while` loop pushes to and pops from the list `tuples` in each iteration, and it is not clear whether the loop will always terminate.

Figure 3: Log-log plot of sum of iteration counts of the loops in `URIVerifier.verify` vs. length of String parameter.

Probing a method with custom TCA drivers. The analyst becomes suspicious about `URIVerifier.verify`, and decides to instrument it and compute the iteration counts as a function of systematically generated string inputs of increasing length. From the TCA menu, the analyst selects the iteration counter instrument for `URIVerifier.verify`. TCA generates a default driver with a call to `URIVerifier.verify`. The analyst updates the driver to test the method with input strings of increasing length from 1 to 30 (Listing 2). TCA runs the driver and plots the sum of iteration counts of the three loops (Figure 3) in the method. The plot shows that this count is clearly exponential in the length of the string parameter. The analyst then passes larger URL strings until a point when the server stops responding at URL length 35.

Listing 1: Decompiled code for `URIVerifier.verify`

```
1  public boolean verify(String s) {
2   Tuple peek;
3   LinkedList<Tuple> tuples = new
        LinkedList<Tuple>();
4   tuples.push(new Tuple<Integer, URIElement>(...));
5   while (!tuples.isEmpty() && (peek =
        (Tuple)tuples.pop()) != null) {
6     if (((URIElement)peek.second).isFinal &&
          ((Integer)peek.first).intValue()==s.length())
7       return true;
8     if (s.length() > (Integer)peek.first)
9       for (...)
10        tuples.push(new Tuple<Integer,
              URIElement>(...);
11    for (URIElement child :
          ((URIElement)peek.second).get(-1))
12      tuples.push(new Tuple(...));
13  }
14  return false;
15 }
```


Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.

The analyst has thus identified the complex and hard-to-understand loops in `URIVerifier.verify` as the root cause of the AC vulnerability using SID analysis. The plot and table of measurements produced by TCA collectively serve as the evidence for the analyst to confirm his hypothesis that Blogger has an AC vulnerability triggered by passing long URL strings to the server.

Remark 1: The discussions in Phases II and III for detecting the vulnerability in the Blogger app are based on the analyst's hypothesis that the code for processing client requests has the vulnerability. However, the vulnerability could also lie in the server listener, in which case the analyst would go through similar steps in Phases II and III when auditing the server listener code.

Listing 2: Driver with workload for `URIVerifier.verify`

```
public class CounterDriver {
 private static final int TOTAL_WORK_TASKS = 30;
 public static void main(String[] args) throws
     Exception {
  for(int i=1; i<=TOTAL_WORK_TASKS; i++){
   RULER_Counter.setSize(i);
   URIVerifier verifier = new URIVerifier();
   verifier.verify(getWorkload(i));
  }
  tca.TCA.plotRegression(
   "URIVerifier.verify Workload Profile",
   TOTAL_WORK_TASKS);
 }
 private static String getWorkload(int size){
  String unit = "a";
  StringBuilder result = new StringBuilder();
  for(int i=0; i<size; i++){
   result.append(unit);
  }
  return result.toString();
 }
}
```

D. Search Space Reduction with LCG

Since detecting AC vulnerabilities is an open-ended search through the app code, tools are important if they can reduce the search space effectively. Let us look at the results for this case study to understand how SID tools reduce the search space. The Blogger app consists of 200 Jimple methods, of which 31 contain loops. In Phase II of the detection process, the LCG helped the analyst to narrow down the search to 16 of these 31 methods. These 16 methods are reachable from the entry point of interest (the code executed by the `ClientHandler` thread). The corresponding number of lines of Jimple code reduced from 2320 to 1197. Thus the LCG provided a significant reduction (over 48%) in the amount of code potentially containing an AC vulnerability that the analyst had to look at.

In Phase III of the detection process, the analyst was able to narrow down the search space for the vulnerability even further to 2 of the 16 methods (118 of the 1197 lines of code) identified in Phase II (over 90% reduction) using TCA. Finally, the total amount of code that had to be manually examined by the analyst at the end of Phase III (between the `HTTPSession.findHeaderEnd` and `URIVerifier.verify` methods) was only 118 lines of Jimple code. Hence, using LCG and TCA, the analyst only needed to consider about 5% of the code in the app to find the AC vulnerability.

E. Discussion

To be precise, AC vulnerabilities need to be defined with respect to constraints for the input size and the resource consumption it causes (e.g., an input of less than 100 bytes causes the server to be busy for at least 300 seconds). The current dynamic analysis capability by TCA can be definitive to confirm or refute each hypothesis for time based vulnerabilities. In the future, we plan to extend our tools and techniques to detect AC vulnerabilities due to excessive space usage. Specifically, we plan to augment LCG with information regarding which data structures are accessed in the loops, and TCA with probes to measure space (heap memory) consumed by these data structures.

Although the LCG helps the analyst to form a hypothesis by identifying loops as potential hotspots in the program and their calling contexts, the SID analysis approach still critically relies on the human analyst's expertise and insight in forming an effective hypothesis. The open question is: *what other tools and advances could significantly reduce the human effort for deriving effective hypotheses?*

IV. Educational Use of SID Tools

The SID tools have a broader applicability to study algorithmic complexity. Recently, it has been noted [15] that "a number of important problems and algorithms for which worst-case analysis does not provide useful or empirically accurate results." and the need to study "the performance of an algorithm not only by its worst-case behavior but rather by its behavior on 'typical' inputs." SID tools are valuable in this context. Students can use them to perform experiments to learn about different aspects of algorithmic complexity using a variety of interesting workloads and observe the corresponding empirical complexity. Students can run experiments to compare empirical complexity computed by TCA and theoretically computed asymptotic complexity. We have designed and run such experiments for commonly taught algorithms.

Learning by experiments is especially important for hybrid algorithms. For example, the widely used hybrid sorting algorithm TimSort combines merge sort and insertion sort. It's complexity switches from that of binary insertion sort to merge sort depending on the size of the input. The students can observe this switching of complexity using SID tools. The LCG for TimSort in Figure 1 shows how multiple subroutines contribute to the complexity of the *sort* method. Figure 4 shows TCA plots generated by selectively instrumenting three methods from TimSort. It reveals an abrupt change in behavior for one of the selected methods at the workload size 32.

Empirical complexity can also help find and understand subtle bugs in a given implementation of an algorithm. For

Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.

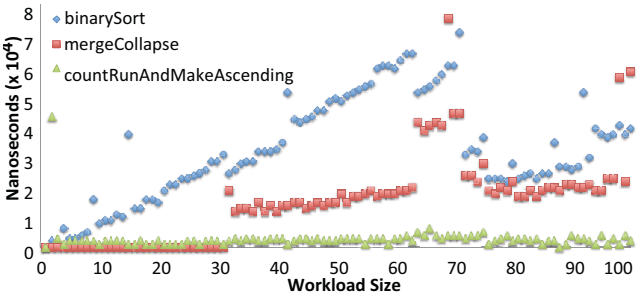

Figure 4: Wall clock time for three methods in `TimSort`

example, implementations of TimSort in OpenJDK, Python and Android were found to crash [16] for a specific input.

V. Related Work

The worst case complexity for arbitrary programs is not always computable (amounts to the halting problem). Static analysis tools such as COSTA [5] and AProVE [6] formulate and solve recurrence relations to compute the asymptotic complexity or termination of loops. However, they only work for a restricted class of loops [7], [17], [18], or do not support Java bytecode [19].

Dynamic analysis approaches typically use benchmarks or fuzzing techniques to provide input workload for the program. Some dynamic analysis tools require the source code (not always available) to add profiling information [12]. Recent tools and techniques for *input-sensitive profiling* [14] automatically estimate the size of the input for generating the workload for individual routines in a program.

The algorithmic profiling tool *AlgoProf* [20] introduced the idea of algorithm profiling as supported by SID tools. *AlgoProf* does not support several important capabilities we have introduced such as selective instrumentation or interactive visualization. The most severe limitation of *AlgoProf* is its overhead, both in terms of space and time. TCA's selective instrumentation helps overcome this limitation.

VI. Conclusion

This paper presents statically-informed dynamic (SID) analysis to detect algorithmic complexity (AC) vulnerabilities in software. Specifically, the paper describes two tools and their case studies. The tools are useful to explore program loops and their structural and behavioral attributes. The Atlas platform [9] is leveraged to integrate the two tools and enable interactive visualization specifically designed to understand programs with respect to their loops. The case studies are: (a) a cybersecurity use to detect an AC vulnerabilities, (b) an educational use to analyze and learn different aspects of algorithmic complexity through experiments.

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 *Usenix Security*, vol. 2, 2003.

[2] E. Adi, Z. A. Baig, P. Hingston, and C.-P. Lam, "Distributed denial-of-service attacks against http/2 services," *Cluster Computing*, pp. 1–8, 2016.

[3] "XML denial of service attacks and defenses," https://msdn.microsoft.com/en-us/magazine/ee335713.aspx.

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

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Costa: Design and implementation of a cost and termination analyzer for Java bytecode," in *Formal Methods for Components and Objects*. Springer, 2008, pp. 113–132.

[6] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl, "Alternating runtime and size complexity analysis of integer programs," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 140–155.

[7] J. Ouaknine and J. Worrell, "On linear recurrence sequences and loop termination," *ACM SIGLOG News*, vol. 2, no. 2, pp. 4–13, 2015.

[8] "Space/time analysis for cybersecurity," http://www.darpa.mil/program/space-time-analysis-for-cybersecurity.

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

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

[11] "TimSort," http://bugs.python.org/file4451/timsort.txt.

[12] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *Foundations of Software Engineering*. ACM, 2007, pp. 395–404.

[13] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a Java bytecode optimization framework," in *Conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1999, p. 13.

[14] E. Coppa, C. Demetrescu, and I. Finocchi, "Input-sensitive profiling," *IEEE Transactions on Software Engineering*, vol. 40, no. 12, pp. 1185–1205, 2014.

[15] M.-F. Balcan, B. Manthey, H. Röglin, and T. Roughgarden, "Analysis of Algorithms Beyond the Worst Case (Seminar 14372)," *Dagstuhl Reports*, vol. 4, no. 9, pp. 30–49, 2015.

[16] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle, "Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case," in *Computer Aided Verification*. Springer, 2015, pp. 273–289.

[17] A. M. Ben-Amram, S. Genaim, and A. N. Masud, "On the termination of integer loops," in *Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 72–87.

[18] O. Olivo, I. Dillig, and C. Lin, "Static detection of asymptotic performance bugs in collection traversals," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 369–378.

[19] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 127–139.

[20] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *Programming Language Design and Implementation*. ACM, 2012, pp. 67–76.

Authorized licensed use limited to: Iowa State University Library. Downloaded on June 08,2024 at 03:24:14 UTC from IEEE Xplore. Restrictions apply.