J-Viz: Finding Algorithmic Complexity Attacks via Graph Visualization of Java Bytecode

Md. Jawaherul Alam* Michael T. Goodrich† Timothy Johnson‡

Dept. of Computer Science, University of California, Irvine, CA USA

Abstract

We describe a security visualization tool for finding algorithmic complexity attacks in Java bytecode. Our tool, which we call J-Viz, visualizes connected directed graphs derived from Java bytecode according to a canonical node ordering, which we call the *sibling-first recursive* (SFR) numbering. The particular graphs we consider are derived from applying Shiver's *k*-CFA framework to Java bytecode, and our visualizer includes helpful links between the nodes of an input graph and the Java bytecode that produced it, as well as a decompiled version of that Java bytecode. We show through experiments involving test cases provided by DARPA that the canonical drawing paradigm used in J-Viz is effective for identifying potential security vulnerabilities for algorithmic complexity attacks.

1 Introduction

The Space/Time Analysis for Cybersecurity (STAC) program [10] at the U.S. Defense Advanced Research Projects Agency (DARPA) aims to develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, and specifically to vulnerabilities based on algorithmic complexity and side channel attacks. STAC seeks to enable security analysts to identify algorithmic resource usage vulnerabilities in software to support a methodical search for them in the software upon which the U.S. government, military, and economy depend [10].

Our Contributions. In this paper, we describe a tool, the JVM abstracting abstract machine (Jaam) Visualizer, or "J-Viz" for short, which is intended for use by security analysts to perform such searches through the exploration of graphs derived from Java bytecode. Thus, we are not attempting to solve the problem of identifying software algorithmic complexity attacks in a completely automated way, but instead we are providing a means for doing semi-automated analysis that increases the efficiency of a human analyst. The workflow for our tool involves taking a given program, specified in Java bytecode, and constructing a control-flow graph of the possible execution paths for this software, using a framework known as *control flow analysis* (CFA) [23]. Our tool then provides a human analyst with an interactive view of this graph, including heuristics for aiding the identification of which parts of the provided program seem suspicious.

One of the main components of our J-Viz tool involves visualizing control-flow graphs in a canonical way based on a novel vertex numbering scheme that we call the *sibling-first recursive* numbering. This numbering scheme is essentially a hybrid between the well-known breadth-first and depth-first numbering schemes, but differs from both in a way that appears to be more useful for visualizing control-flow graphs so as to highlight potential algorithmic complexity attacks. In particular, as we show that with respect to some actual test cases provided to us by DARPA that this approach is effective at aiding a human security expert to find such attacks. We designed J-Viz with the following goals in mind:

- We want users to be easily able to recognize patterns in source code from our visualizations. Thus, similar sections of code should produce similar subgraphs, which should be drawn in a similar way.
- We want to use a hierarchical visualization, in which users can collapse or expand sections of the graph to different levels of detail. But we also want them to be able to build a consistent mental model of the graph. Thus, drawings should not drastically shift the relative positions of the vertices when sections are collapsed or expanded.
- No matter what sequence of actions the user performs, drawings should be consistent. That is, the same view of a graph, in which the same set of nodes are collapsed and expanded, should always be drawn in the same way.
- Our system should rank sections of the graph by how likely they are to produce vulnerabilities, and display this information visually to the user.

Related Work. Although it is using different means to achieve mental map preservation, the J-Viz system follows in a long line of research on techniques directed at preserving the mental map of a graph drawn dynamically. For instance, Misue *et al.* [20] discuss node movement adjustments, including avoiding node overlaps, for preserving the mental map. Diehl and Carsten [7] discuss force-directed approaches for preserving the mental map between instances of a changing graph. Goodrich and Pszona [12] study efficient algorithms for minimizing vertex movements as a graph is incrementally revealed in an online manner. With respect to existing software systems, the Graphviz [8] and GraphAEL [9] systems both include algorithms intended to preserve the mental map as a graph is modified. Bridgeman and Tamassia [3] formally study metrics for characterizing mental map preservation between different instances of a changing graph. So as to provide an empirical basis for such work, a user study of Purchase *et al.* [21] supports the thesis that preserving the mental map for graph visualization is a useful goal to aid users in performing tasks on graphs.

Visualization tools have also previously been applied to source code. Doxygen [25], a tool for automatically generating documentation, can produce various kinds of diagrams for visualizing code, including call graphs. It is generally configured to use the *dot* [11] tool from GraphViz to draw these graphs hierarchically. Similarly, Visual Studio can visualize call graphs to aid programmers in debugging applications [6]. In constrast with these systems, our J-Viz tool provides four main features that these tools do not provide. First, J-Viz shows a greater level of detail, since it analyzes code at the level of individual instructions rather than methods. Second, J-Viz allows the user to interact with a graph and produce multiple views of the same Java bytecode. Third, the layout algorithm

*email: alamm1@uci.edu
†email: goodrich@acm.org
‡tujohnso@uci.edu

978-1-5090-1605-1/16/$31.00

used in J-Viz is designed to draw similar code fragments in the same (canonical) way, so as to highlight portions of repeated code. Fourth, J-Viz guides the user to potential security vulnerabilities, by highlighting nodes that are believed to be risky based on algorithmic complexity (or other factors), whereas these other systems were not focused on software security.

Another tool, Jinsight [5], can be used to profile a Java program to provide various views of resource usage, such as highlighting which instances of a class have taken the most time or used the most memory. This tool does not provide a full graph of the program's possible execution paths, however, which we believe to be essential for detecting security vulnerabilities. In addition to these Java-based tools, there are also tools for visualizing compiled executibles in other languages for malware analysis, such as the tool by Quist and Liebrock [22].

At a high level, our work is also related to recent work on applying graph visualization tools for security visualization. For example, work by Di Battista *et al.* [2] on visualizing flows in the Bitcoin transaction graph and work by Mansman *et al.* [17] on visualizing host behavior in a network are also in this area. (See also the surveys by Tamassia *et al.* [24] and Wagner *et al.* [26].)

2 Graph Generation via Static Analysis

In this section, we review the process that takes Java bytecode as input and produces the graphs that are visualized in J-Viz. These graphs are produced using the *JVM abstracting abstract machine* (*Jaam*) tool [4] developed at the University of Utah based on the work of Van Horn and Might [16], which itself is based on control-flow analysis (CFA) framework known as *k-CFA* [18, 23]. Because there could be exponentially many possible execution paths of any given program, which would be too large to visualize and reason about, the *k*-CFA framework compresses execution paths into a graph of reasonable size that represents possible executions of a Java program at the instruction level. Such a graph is called *sound* if it represents every possible execution path, and *precise* if it excludes every impossible execution path. The *k*-CFA framework is sound, and it has a tunable degree of precision based on the integer parameter, k, albeit at the cost of creating additional states in the graph for larger values of k.

At the lowest level of the hierarchy, 0-CFA, we discard contextual information and generate one state, which forms a vertex in the graph representation, for each line of Java bytecode. Then we add edges for every possible state that could be reached from a given state. For example, a return statement will have an edge to every place from which our current method could have been called. At the next level, 1-CFA, each state also tracks the location from which its method was called. This easily generalizes to higher levels, so that for k-CFA, each state stores the locations of the previous k function calls. This added information allows many of the spurious branches produced by 0-CFA to be pruned. (For additional information, please see more detailed descriptions of k-CFA [16,18,23].)

0-CFA is known to take $O(n^3)$ time to construct a graph for a program with n lines of code, and this is believed to be tight [14]. k-CFA is EXPTIME-complete for functional languages [15], but can be solved in polynomial time for object-oriented languages [19]. Thus, to provide a reasonable balance between soundness, precision, and efficiency, the version of the Jaam static analyzer used for the work of this paper is based on 1-CFA. To summarize, then, the Jaam static analyzer takes as input Java bytecode for a given program and produces an directed graph, G, that represents the results of a 1-CFA performed on this bytecode. This graph is *ordered*, in the sense that the outgoing edges for each node are sorted according to the order in which the corresponding instructions appear in the Java bytecode.

3 Our Sibling-First Recursive Layout Algorithm

Our approach to the layout of graphs produced by the Jaam tool [4] is based on what we believe is a novel graph numbering scheme, which we call a *sibling-first recursive* (SFR) numbering. Intuitively, SFR is a hybrid numbering scheme that combines features of a breadth-first search (BFS) numbering and a depth-first search (DFS) numbering. We show in Figure 1 the difference between algorithms for doing a depth-first search (DFS) ordering of a directed graph and a sibling-first recursive (SFR) numbering. See also Figures 8, 9, and 11 in the appendix, which illustrate SFR spanning trees and their differences with DFS and BFS spanning trees.

Algorithm DFS(v):
Number v as vertex n
Increment n
for each directed edge (v,w) **do**
  if w is not numbered **then**
    Identify w as a child of v
    DFS(w)

Algorithm SFR(v):
for each directed edge (v,w) **do**
  if w is not numbered **then**
    Identify w as a child of v
    Number w as vertex n
    Increment n
for each directed edge (v,w) **do**
  if w is a child of v **then**
    SFR(w)

Figure 1: DFS and SFR algorithms to explore the connected component of a vertex, v, in a directed graph, G. We assume there is a global variable, n, which is used to number the vertices. In the case of DFS, we initialize $n = 1$ and call DFS(v) on a vertex v that is to become the root of the DFS tree. In the case of SFR, we initialize a vertex, v, as the root of the SFR tree, numbering it as vertex 1, and we set $n = 2$ and call SFR(v).

Our motivation for using the SFR numbering is that we feel it produces a rooted spanning tree that corresponds more intuitively with the way that programmers conceptualize the main "backbone" of the control flow of their software. For example, it places the true-false branches of if statements as children of the condition that branches to them. In addition, it places the multiple branches of a switch statement as children of the condition that branches to them, even if some of the branches flow-through to other branches. (E.g., see Fig. 2.) Furthermore SFR numbering also enables the viewer to visually identify isomorphic subgraphs of the graph, corresponding to identical, repeated or equivalent lines of code.

High-Level Description of Our Layout Algorithm. At a high level, there are five steps in our algorithm for producing a drawing of the graph, G:

1. We construct an SFR numbering and rooted spanning tree, T, for our input graph, G, which will be used as the "backbone" of our drawing.
2. We draw the tree T using a recursive placement algorithm.
3. We add the edges of G that are not in the tree T.
4. We highlight in our drawing the sections of our graph that are most likely to contain vulnerabilities, based on various criteria.
5. We automatically group subsets of nodes before displaying the entire graph to the user, in a way that allows the user to expand such collapsed nodes.

In the remainder of this section, we describe in more detail each of the above steps in our layout algorithm.

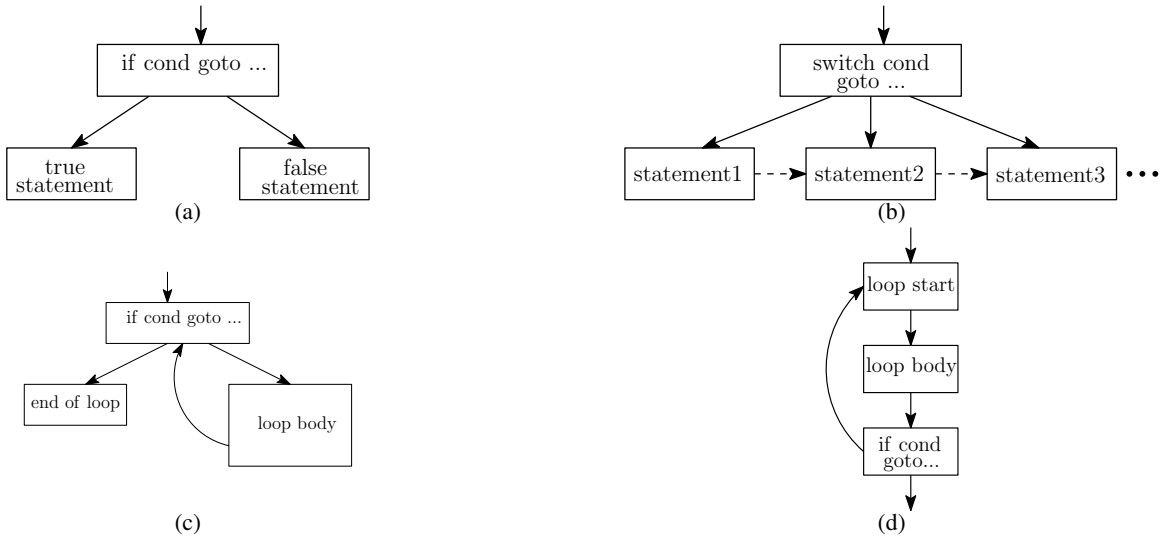

Figure 2: Sample graph layout for (a) if-else conditional statement, (b) switch statement (dashed edges for flow through statements), (c) for and while loop, (d) do-while loop.

Constructing an SFR search tree. In our first step, we construct a rooted (ordered) SFR spanning tree, T, for the graph, G, produced from the Jaam tool [4]. The algorithm we use to perform this construction is exactly the SFR algorithm shown in Fig. 1, with the added detail that as we traverse the graph G to construct our SFR search tree, T, we process the out-edges from each vertex using the ordering for G, consistent with the intuitive way programmers naturally organize branches for different types of software branch points. As we highlight in one of our test cases, this approach tends to produce almost identical drawings for repeated (e.g., cut-and-pasted) software code fragments. It also produces ordered combinatorial layouts for each of the following types of code constructs, as shown in Fig. 2.

- If-else conditional statement: the true and false components are siblings, with the true component coming first.
- Switch conditional statement: the different branches of the switch statement are siblings of the conditional statement, ordered by their appearance in the code (even if there are non-tree edges between them that would be representing flow-throughs from one branch to another).
- While/for loop: the end-of-loop statement and loop body are both siblings of the conditional statement, with the end-of-loop statement coming first.
- Do-while loop: the start statement, loop body, and conditional statements are in a single path, with a non-tree edge leading back to the start statement.

Drawing the Nodes of our SFR Search Tree. Once we have constructed our (ordered) SFR search tree, T, we draw it recursively, starting from the leaves of T. Each parent is drawn on a row above all of its children. Chains of nodes are drawn in a vertical column. When we reach a branch point, we lay out each of the subtrees from left to right. We require that only a direct descendant of a node can be placed directly underneath it. This means that each subtree, no matter its size, will have an entire vertical lane reserved for it from top to bottom in our graph. See Fig. 3.

This requirement might at first seem to waste space in our drawing, but it maintains consistency when a user expands or collapses connected sections of nodes. To see why this is so, suppose that two nodes are placed at the same x-coordinate, but neither is an ancestor of the other. Suppose further that the user then chooses to collapse the set consisting of the path from each of these nodes up to their lowest common ancestor. If this happens, then if we simply shift up that portion of the spanning tree, then we will cause overlaps, which would require shift of nodes to fix. (See Fig. 4.) But such a shift would be detrimental to the mental map. Thus, rather than produce a compact drawing that reuses vertical space, we use the scheme described above, which tends to preserve the mental map even as we would be collapsing or expanding paths in the spanning tree, T, and shifting the remaining portions accordingly.

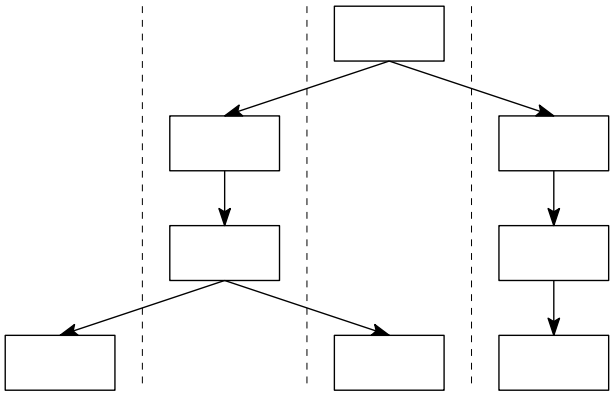

Figure 3: A tree drawn according to our algorithm. The dashed vertical lines show separation of subtrees into unique lanes.

Drawing Edges. After we have placed the nodes of our ordered spanning tree, T, we must draw all of the edges in our graph. In our case we choose to draw downward edges as straight line segments, and we then draw upward edges as curved segments. In addition to drawing arrows at the ends of such segments, this convention provides a visual cue for which direction an edge is pointing. It also prevents upward edges from lying on top of downward edges. For example, this makes the drawing of the graph of software implementing the bubblesort algorithm, shown in Fig. 10 in

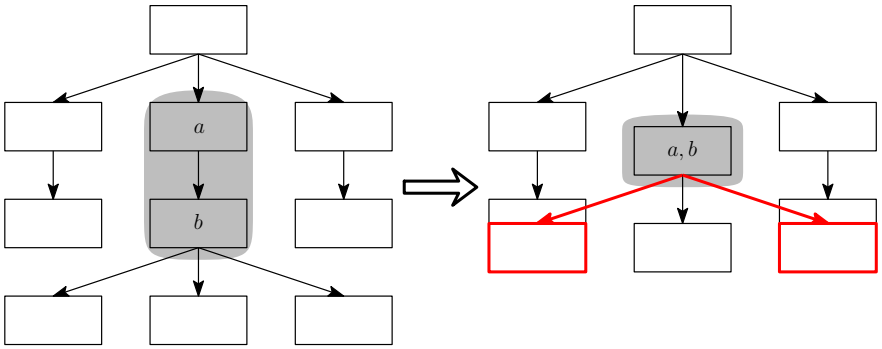

Figure 4: An example of how breaking our drawing style can cause problems. When the user collapses nodes a and b, some of the nodes need to be shifted. To avoid this problem, we forbid a node from lying directly above a node that is not a direct descendant.

Appendix B, more readable.

Highlighting. After the nodes and edges of the graph, G, are placed, we highlight vertices to guide the user on where to begin examining it to discover possible security vulnerabilities. For attacks based on increasing the running time for the software on certain inputs, we want to highlight nodes that are likely to be visited the greatest number of times during an execution. So we color our states from green to red based on how likely each node is to be involved in a vulnerability. We have considered the following three ranking methods:

- We could perform a recurrence analysis that computes an upper bound on how many times each node is visited. Automated recurrence analysis has progressed to the point of being able to provide strong upper bounds on many simple algorithms [1]. But we do not believe that this is yet possible for programs such as the ones we need to examine, which contain thousands or tens of thousands of lines of code, and have a complex loop structure.
- We could profile our code, by providing a sample input and counting how many times each state is visited. But while this may help for honestly written programs, we believe it is unlikely to identify the kinds of deliberate attacks that we need to discover. The programs we are given should in most cases perform well, because otherwise they would not be used at all. But some may have a hidden trigger that causes them to run for much longer.
- We could count the number of nested loops that contain each node. This is a somewhat naive method, but it does seem to give a reasonable heuristic, as will be seen in our experimental results. It is also feasible to compute even for very large graphs. Hence this is the method that we choose to use.

In order to do this highlighting, of course, we need to determine for each node its level of nesting with respect to the loops of the program. We use an adaptation to the SFR tree of a definition by Havlak [13] for DFS trees:

- The outermost loops are the maximal strongly connected components of the directed control-flow graph, G.
- The header for a loop is the first node in the loop that is reached in the SFR tree, T.
- The inner loops are the maximal strongly connected components that remain when the header is removed.

A graph is said to be *reducible* if every cycle has a single entry point [13]. If the graph is reducible, then the loop decomposition does not depend on which rooted spanning is used. But if a cycle has multiple entry points, then the order in which we explore the branches could matter. Thus, in our case, we use the canonical SFR tree, T, that has already been defined for our graph.

To compute loop headers efficiently, we use an algorithm from Tao *et al.* [27]. This traverses the ordered spanning tree and passes loop header information up the tree. While their method could take a long time for artificially complex graphs, it takes linear time for most real-world programs, because the spanning trees for such graphs tend to be reducible or "nearly" reducible.

Grouping Nodes. We have included four different ways in which nodes in the graph, G can be aggregated, and we present the initial drawing of G to the user based on a pre-defined grouping of certain nodes, with some of these pre-collapsed. First, we choose not to explore nodes that correspond to calls to the Java library, since we do not expect it to contain vulnerabilities. Instead, every such call is collapsed to a single line. This prevents us from creating hundreds of thousands of nodes for the Java library. Nevertheless, the static analyzer, Jaam, needs to do this carefully, so that it can approximate the state of Java library objects and predict their later behavior. For example, any object that is added to an ArrayList or a HashMap can later be taken out. Still, we assume that such an identification is given as an annotation to the input graph, G, since this identification is solely the domain of the Jaam tool. Second, we automatically group each connected set of nodes that belong to the same method. That way, if the user is not interested in the details of a given method, they can collapse it to a single node. Third, we automatically group chains of method nodes that were created in the previous step. Generally, having long chains of nodes taking up a large portion of the screen space hinders the user from seeing the branching structure that they need to find. Finally, we allow the user to select a connected set of nodes and collapse them dynamically, along with providing a comment explaining the purpose of the corresponding section of code.

4 Experiments

In this section, we describe some test cases we performed to test the effectiveness of the J-Viz system for visualizing Java bytecode and identifying security vulnerabilities that could be triggered by algorithmic complexity attacks. As input to these test cases, we were provided by DARPA with programs to analyze to test our system, some of which were produced by a "red team" tasked with deliberately creating software that contains vulnerabilities to algorithmic complexity attacks.

Our first test case is for a program for verifying a secret password, without revealing any information about such a password. In this case, J-Viz was effective in leading a security analyst to a nested loop that checks each character in the password one at a time. In part, because of the way that the SFR spanning tree lays out conditional branches in an intuitive manner and draws loop edges as curved segments, the analyst was able to notice that the password-checking program exits as soon as it finds the first character that does not match. (See Fig. 13 in Appendix C.) The analyst then correctly identified this as a vulnerability (inserted by the red team), since, by timing multiple executions of the program, an attacker can easily determine how many correct characters of a password that they entered with each attempt. Thus, such an attacker could quickly crack the password by a simple iterative search.

Our second test case is for a program for analyzing and classifying images based on features, such as the number of edges or the amount of each color that is present. This program is around 1,000 lines of Java code, and produces about 3,000 nodes in our graph. The goal of the security analyst in this case was to determine if this program can be made to take much longer than it should (specifically, greater than 18 minutes to analyze a 70 KB image). For most images of that size, the program takes around 6 minutes. But, through the use of J-Viz, a security analyst was able to create an image that would take over an hour to be analyzed. The key insight for the analyst was to pay attention to the highlighting in our visualizer, which showed the deepest nested loops in dark red. (See Fig. 5.)

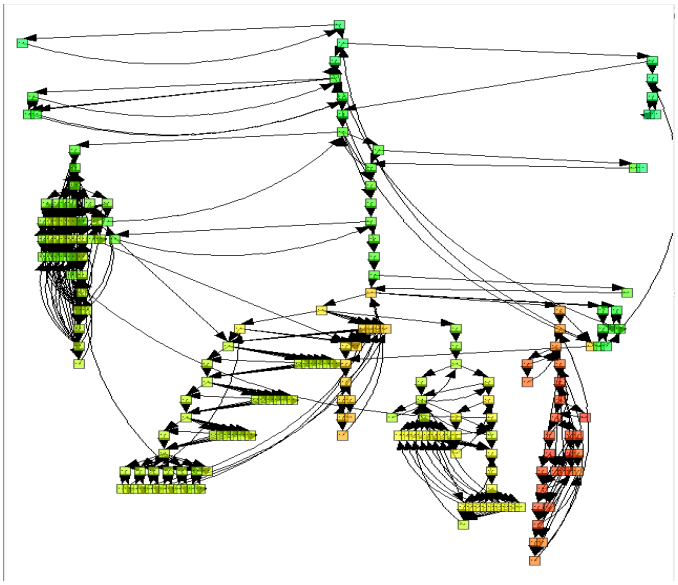

Figure 5: Our visualization for the graph for an image processor program, showing the red section of a deeply nested loop (which contained a security vulnerability) in the bottom right.

In this case, the analyst noticed that many of these nodes were part of the Mathematics class, which provided custom implementations of various mathematical functions. In particular, the exponential function was implemented using a Taylor series, with the number of terms depending on a function of the RGB value of each pixel. This function contained a "spike" which is shown in Figure 6, implying a large number of terms in the Taylor series for an image having a particular RGB value. Given this information, the analyst was then able to create an image that triggered this behavior for every pixel, which took over an hour to process, confirming the vulnerability.

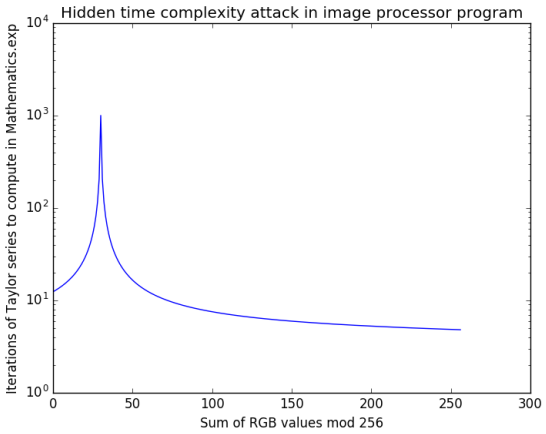

Figure 6: The time complexity of image processor spikes by two orders of magnitude when the sum of the RGB values is exactly 30.

Our final test case is a text analysis program, called *TextCrunchr*, that analyzes documents to compute statistics, such as word frequencies and word lengths, and can process plain text files as well as compressed files (in the case of compressed files, it uncompresses them before analyzing their contents). This program produced about 5,000 nodes in its CFA graph. The goal of the security analyst using our tool was to determine if this program can be made to run for a longer period of time than expected (in this case, greater than 5 minutes to analyze at most 400 KB of data), which could be an indication of a vulnerability to an algorithmic complexity attack.

In using our tool for this task, the key insight for the analyst was to pay attention to the highlighted nodes in our visualizer, which showed the most deeply nested loops in dark red. In this particular case, the analyst noticed that many of these nodes were part of a custom implementation of a HashMap class that uses some simple bitwise operation to hash an input value. This implementation of Hashmap performed poorly when many hashcode collisions occur, and any input containing many words that hash to the same hashcode triggered quadratic behavior in the text analysis. Given this insight, the analyst created an input file within the 400 KB that forced the program to run significantly longer than 5 minutes, thus uncovering the algorithmic complexity attack vulnerability.

In addition to these test cases, we show additional examples and screenshots of our tool in Appendix B and C.

5 Conclusion and Future work

We have described a new software visualization tool, J-Viz, that uses an SFR numbering scheme for graphs produced using the 1-CFA framework so that similar sections of code are drawn similarly and deeply nested code portions are placed well and highlighted.

In keeping with recent graph drawing research, we have argued that our system preserves the mental map of the user as they interact with the graph. We also meet all of the criteria that both programmers and researchers have considered essential for software visualization. As a result, our system has already proven to be useful for human analysts in finding various kinds of software vulnerabilities.

Although our tool is already fairly scalable, in future work, we plan to test our system for even larger programs. We also plan to study ways to provide semi-automated methods for identifying

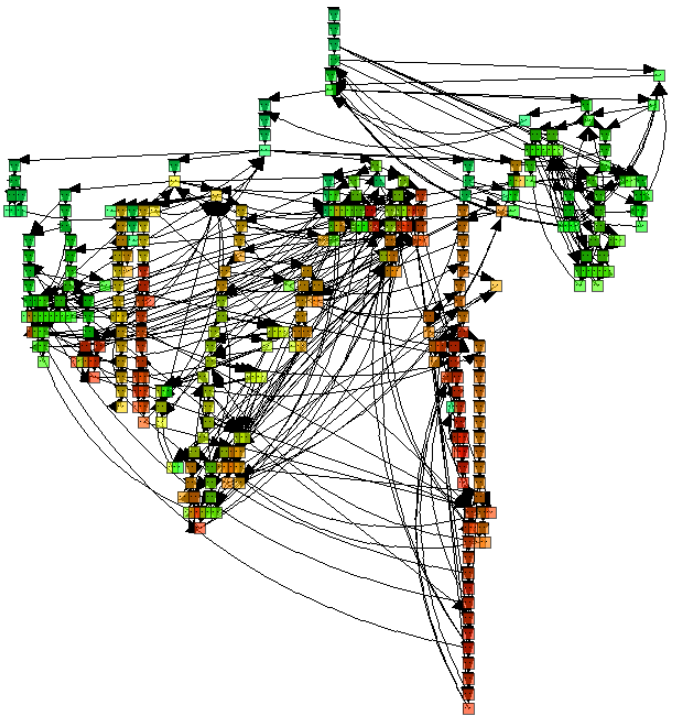

Figure 7: Our visualization for the graph for a text analysis program, showing the red section of a deeply nested loop, which contained a security vulnerability.

other kinds of potential algorithmic-complexity security vulnerabilities, such as those due to over-consumption of memory.

Acknowledgements

This article reports on work supported by the Defense Advanced Research Projects Agency under agreement no. AFRL FA8750-15-2-0092. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work was also supported in part by the U.S. National Science Foundation under grants 1228639 and 1526631. In addition, we would like to thank David Eppstein, Matthew Might, William Byrd, Michael Adams, and Guannan Wei for helpful discussions regarding the topics of this paper.

References

[1] 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*, pages 113–132, 2007.

[2] G. D. Battista, V. D. Donato, M. Patrignani, M. Pizzonia, V. Roselli, and R. Tamassia. Bitconeview: visualization of flows in the bitcoin transaction graph. In *IEEE Symp. on Visualization for Cyber Security (VizSec)*, pages 1–8, 2015.

[3] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In S. H. Whitesides, editor, *6th Int. Symp. on Graph Drawing (GD)*, pages 57–71, 1998.

[4] U. Combinator. Jaam: JVM abstracting abstract machine. `https://github.com/Ucombinator/jaam`. Accessed 2016-06-10.

[5] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Software Visualization*, pages 151–162, 2002.

[6] R. DeLine, G. Venolia, and K. Rowan. Software development with code maps. *Commun. ACM*, 53(8):48–54, 2010.

[7] S. Diehl and C. Görg. Graphs, they are changing: Dynamic graph drawing for a sequence of graphs. In M. T. Goodrich and S. G. Kobourov, editors, *10th Int. Symp. on Graph Drawing (GD)*, pages 23–31, 2002.

[8] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and dynagraph — static and dynamic graph drawing tools. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148, 2004.

[9] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Graphael: Graph animations with evolving layouts. In G. Liotta, editor, *11th Int. Symp. on Graph Drawing (GD)*, pages 98–110, 2004.

[10] T. Fraser. Space/time analysis for cybersecurity (STAC). `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.

[11] E. R. Gansner, E. Koutsofios, S. C. North, and G.-P. Vo. A technique for drawing directed graphs. *Software Engineering, IEEE Transactions on*, 19(3):214–230, 1993.

[12] M. T. Goodrich and P. Pszona. Streamed graph drawing and the file maintenance problem. In S. Wismath and A. Wolff, editors, *21st Int. Symp. on Graph Drawing (GD)*, pages 256–267, 2013.

[13] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[14] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 342–351. IEEE, 1997.

[15] D. V. Horn and H. G. Mairson. Deciding *k*cfa is complete for EXPTIME. In *ACM SIGPLAN international conference on Functional programming, (ICFP'08)*, pages 275–282. ACM, 2008.

[16] D. V. Horn and M. Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54(9):101–109, 2011.

[17] F. Mansman, L. Meier, and D. A. Keim. Visualization of host behavior for network security. In J. R. Goodall, G. Conti, and K.-L. Ma, editors, *Proc. of the Workshop on Visualization for Computer Security (VisSec)*, pages 187–202, 2008.

[18] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[19] M. Might, Y. Smaragdakis, and D. V. Horn. Resolving and exploiting the *k*-cfa paradox: illuminating functional vs. object-oriented program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'10)*, pages 305–315. ACM, 2010.

[20] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, 1995.

[21] H. C. Purchase, E. Hoggan, and C. Görg. How important is the "mental map"? – an empirical investigation of a dynamic graph layout algorithm. In M. Kaufmann and D. Wagner, editors, *14th Int. Symp. on Graph Drawing (GD)*, pages 184–195, 2007.

[22] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In *6th Int. Workshop on Visualization for Cyber Security (VizSec)*, pages 27–32, 2009.

[23] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[24] R. Tamassia, B. Palazzi, and C. Papamanthou. Graph drawing for security visualization. In I. G. Tollis and M. Patrignani, editors, *16th Int. Symp. on Graph Drawing (GD)*, pages 2–13, 2009.

[25] D. van Heesch. Doxygen: Source code documentation generator tool. *Available online: http://www.doxygen.org*, 2008.

[26] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, and W. Aigner. A survey of visualization systems for malware analysis. In R. Borgo, editor, *Eurographics Conference on Visualization (EuroVis)*, pages 105–125, 2015.

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

A SFR Numbering

Fig. 8 illustrates a graph (for the complete code of a bubblesort implementation); the vertices are labeled with their SFR numbers.

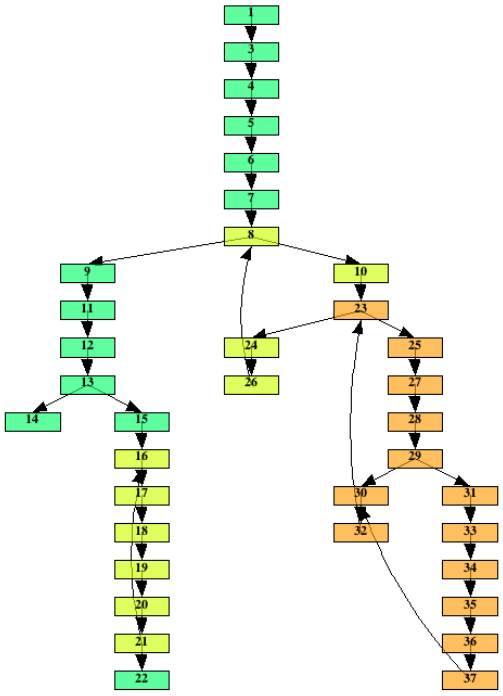

Figure 8: A graph for the bubblesort algorithm; vertices are labeled with SFR numbers.

Fig. 9 illustrates the difference in the spanning trees and the vertex numberings obtained in our SFR search, and a more conventional depth-first search. These are obtained for a code segment containing a switch statement. The tree and the numbering obtained in the SFR shows the structure of the switch statement more naturally.

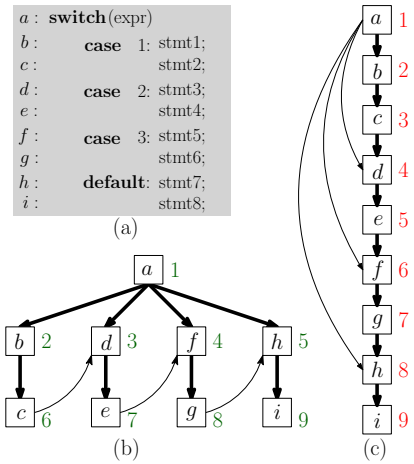

Figure 9: (a) A code segment containing a switch statement, (b) the spanning tree (thick edges) and vertex numbering (in blue color) for the graph obtained from SFR search, and (c) the spanning tree (thick edges) and vertex numbering (in red color) for the graph obtained from depth-first search.

B Additional Figures and Examples

Here we show the graphs for some simple algorithms, as visualized in our J-viz.

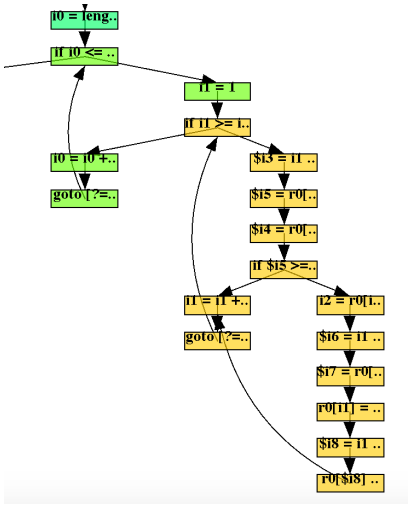

Figure 10: This partial graph of a bubblesort algorithm shows how drawing upward edges as curves and highlighting nested loops can improve readability. It also shows our use of colors for different levels of nested loops.

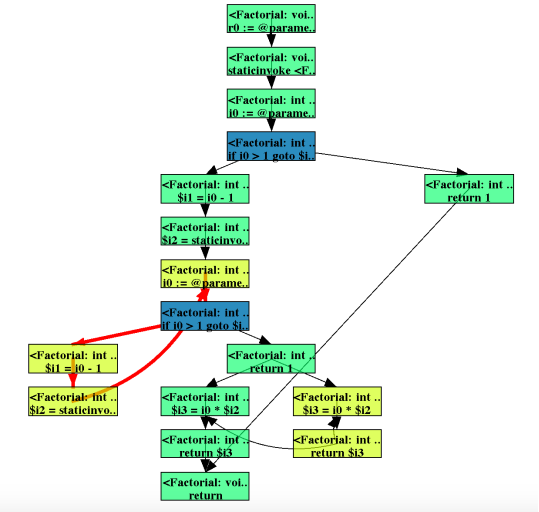

Figure 11: A graph of a recursive factorial function using 1-CFA. The highlighted instruction occurs twice - once in the context of being called from main, and once in the context of the factorial function calling itself. Note, in addition, how this tree is different from a breadth-first search (BFS) tree. Namely, there is a long forward edge from the rightmost node in the drawing. In a BFS numbering of this graph, that edge would force its end-vertex to a higher level, which would distort the natural notion of recursive depth that the SFR spanning tree illustrates better here.

C Illustration of the J-Viz System

Fig. 12 illustrates our J-viz system and its different component.

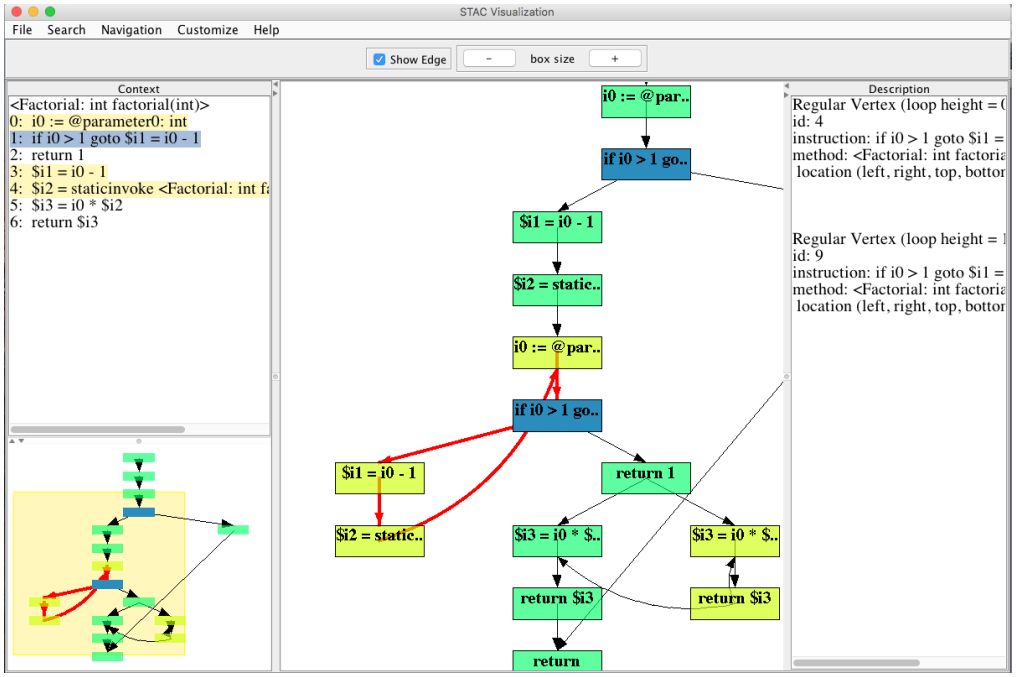

Figure 12: A screenshot of J-Viz with a (clipped) view of an example Factorial program. This shows the left panel with the code for the currently selected method, the right area with the description of each selected node, and the minimap with our current zoom level.

Fig. 13 illustrates how the J-viz system is used in identifying vulnerable code segments in a password checker program.

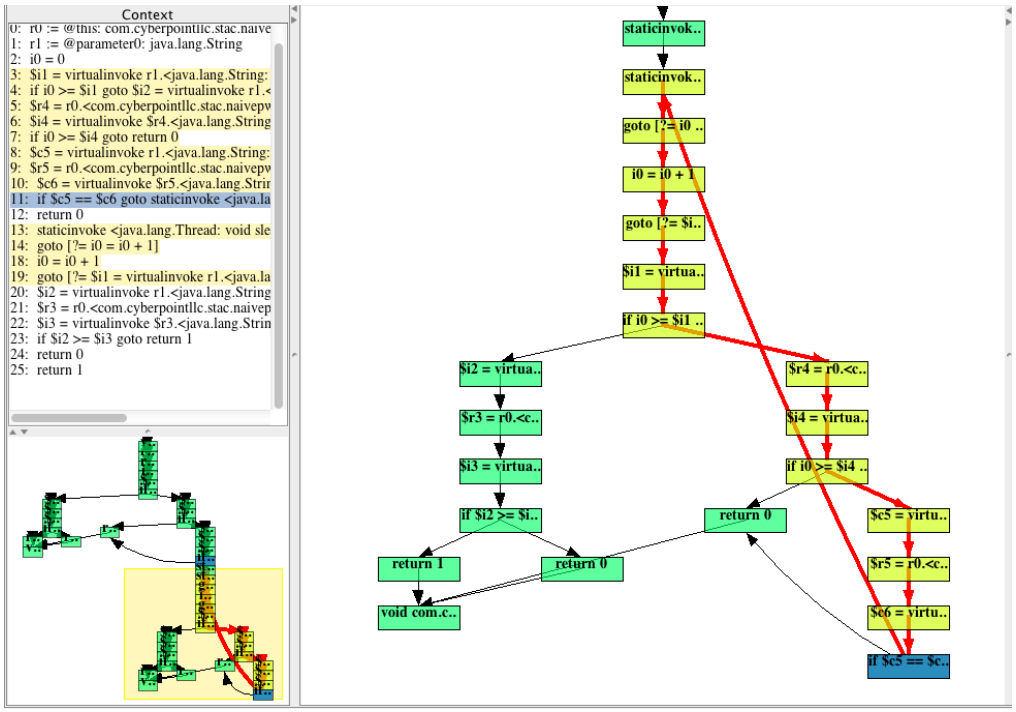

Figure 13: Our layout of the graph for a password checker with the relevant part zoomed, as a part of our first test case. The highlighted node shows a check for each character of a password. If this fails, the program exits the loop immediately, allowing for a side-channel attack (for identifying failed passwords).