Eurographics Conference on Visualization (EuroVis) 2018 (PREPRINT)
J. Heer, H. Leitte, and T. Ropinski
(Guest Editors)

Volume 37 (2018), Number 3 (PREPRINT)

CFGExplorer: Designing a Visual Control Flow Analytics System around Basic Program Analysis Operations

Sabin Devkota and Katherine E. Isaacs

Computer Science, University of Arizona

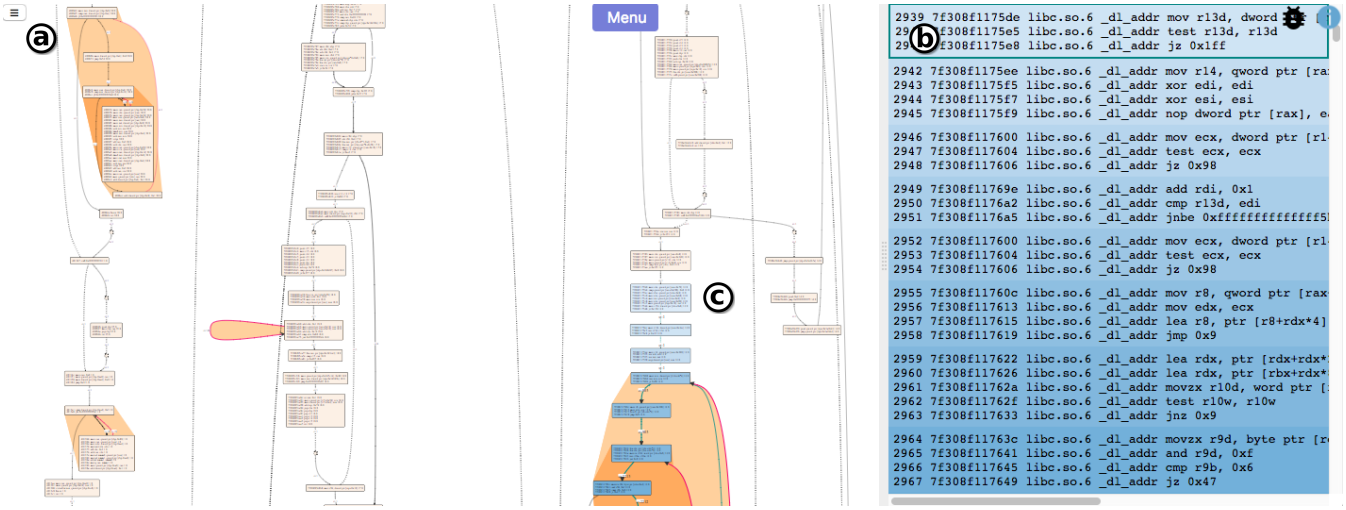


Figure 1: CFGExplorer helps researchers analyze programs and correlate control flow (a) and trace data (b). We designed a domain-specific layout approach to elucidate loop structure (orange). Users can animate execution using the linked blue gradient (c).

Abstract

To develop new compilation and optimization techniques, computer scientists frequently consult program analysis artifacts such as control flow graphs (CFGs) and traces of executed instructions. A CFG is a directed graph representing possible execution paths in a program. CFGs are commonly visualized as node-link diagrams while traces are commonly viewed in raw text format. Visualizing and exploring CFGs and traces is challenging because of the complexity and specificity of the operations researchers perform. We present a design study where we collaborate with computer scientists researching dynamic binary analysis and compilation techniques. The research group primarily employs CFGs and traces to reason about and develop new algorithms for program optimization and parallelization. Through questionnaires, interviews, and a year-long observation, we analyzed their use of visualization, noting that the tasks they perform match common subroutines they employ in their techniques. Based on this task analysis, we designed CFGExplorer, a visual analytics system that supports computer scientists with interactions that are integrated with the program structure. We developed a domain-specific graph modification to generate graph layouts that reflect program structure. CFGExplorer incorporates structures such as functions and loops, and uses the correspondence between CFGs and traces to support navigation. We further augment the system to highlight the output of program analysis techniques, facilitating exploration at a higher level. We evaluate the tool through guided sessions and semi-structured interviews as well as deployment. Our collaborators have integrated CFGExplorer into their workflow and use it to reason about programs, develop and debug new algorithms, and share their findings.

1. Introduction

Several research areas in computer science, including compilation, performance engineering, and reverse engineering, seek to analyze and ultimately transform a given computer program into one with more desirable properties. These properties can include decreased runtime (i.e., optimization), portability, or maintainability. Once developed, these automatic techniques allow software developers to work at higher levels of abstraction, expending less effort to construct responsive, portable, and maintainable programs.

Depending on the particular application a researcher is targeting, the transformation they are developing may be required to operate on the program source code, program executable, or even a log of observed instructions. To gain insight into their problem, researchers often analyze example programs at a low level. Two artifacts commonly used in program analysis are instruction traces and control flow graphs (CFGs). Instruction traces list the executed instructions in a program. They represent ground truth but are difficult to interpret without structure. A CFG is a directed graph with sequential code as nodes and execution paths as edges and can be constructed from an instruction trace. They are commonly visualized using node-link diagrams resulting from a Sugiyama [STT81] style layout. However, researchers struggle to interpret and use these diagrams for even modestly-sized programs. Despite this challenge, we observed researcher, trying to understand if his technique worked, wait 36 hours for a general layered layout to complete and render. Our goal is to determine how visualization can better support the analysis needs of computing researchers.

We have been collaborating with a compilation and program analysis research group for over fifteen months to understand their use of visualization, their analysis needs, and how to design a system that supports them. We augmented our data collection by interviewing researchers in similar areas outside this group. Through these observations, interviews, and questionnaires, we developed a data and task abstraction for dynamic program analysis with CFGs. We found researchers typically visualize CFGs to "debug" their own mental models as well as their proposed transformation algorithms. The operations they perform correspond to the basic operations that compose their proposed algorithms. We used this insight to design a visual analytics system that incorporates these domain-specific operations and structures to improve our collaborators' analysis workflow.

Our visualization prototype incorporates program structures through its graph layout, interactive highlighting, and coordinated navigation. We developed a novel graph layout approach and graph encoding that emphasizes loop constructs matching the models used by both our collaborators and their algorithms. Similarly, we design encodings and interactions for basic program analysis operations that detect loops, instructions, or addresses of interest. We facilitate exploration by linking the temporal information of the trace with the structure information of the CFG and provide quick navigation to program structures such as functions and loops.

In summary, our contributions are:

- a data and task analysis and abstraction for dynamic control flow graphs (Section 4),
- a domain-specific layout for control flow graphs (Section 5.1),
- the design of a visual analytics system for dynamic program analysis (Section 5), and
- the evaluation of the system (Section 6) through user sessions and deployment.

Before discussing these contributions, we first provide a brief overview of the domain (Section 2) and related work (Section 3). We conclude with a discussion of lessons learned that may be transferred to related problems (Section 7).

2. Background

The ultimate goal of our collaborators is to develop automated methods for transforming and enhancing programs. In developing these methods, researchers often perform some sort of manual program analysis, deeply examining the operations of a small set of programs by consulting artifacts such as the program source code, dynamically collected logs of executed instructions ("*traces*"), and the program's *control flow graph*. We describe these artifacts and associated program analysis terminology.

An *instruction trace* is a sequence of instructions executed by a program during its execution. Each line of the trace refers to a single instruction. Included in our traces is the *address* of each executed instruction and its representation in assembly language. An instruction's address refers to its position in storage when the program is loaded. It uniquely identifies the instruction in the context of the rest of the program and may be used to correlate the low level instruction to higher level source code. In addition to addresses and assembly, our traces sometimes include extra information regarding the state of the memory or CPU flags during execution. Figure 1 (right) shows a raw instruction trace in ASCII format.

We use the term *control flow graph* (CFG) to refer to a directed graph representing execution paths that a program can take [FOW87]. The vertices of the graph are *basic blocks*: consecutive instructions that must be executed in order. We use the term *jump* to refer to instructions that can move execution to an instruction at a non-adjacent address. Typically a basic block ends with a jump. The edges in a CFG show the relationships between basic blocks determined by their jumps. Typically, CFGs are relatively sparse graphs, but high degree nodes can occur due to `switch` statements or functions that are called from multiple locations.

Loops, often declared in source code with keywords such as `for` and `while`, are of interest to program analysts. In the CFG, a loop has a single entry node called the *header*. A node is said to *dominate* another if all paths between the program entry node and the other node must go through the dominating node. A *back edge* is an edge whose sink node dominates its source node, which is then referred to as the *tail node*. Thus, the back edge induces the looping behavior. We refer to non-loop nodes that succeed the loop as *outer nodes*. Nodes representing basic blocks that can exit the loop are called *exit nodes* and their corresponding exiting edges are *exit edges*. Figure 2

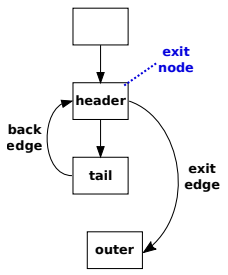


Figure 2: Loop.

shows these components. In the figure, the header node is also an exit node, because it has an edge to a successor outside the loop.

Our CFGs are derived from instruction traces. Unlike statically generated CFGs, they include interprocedural edges and omit basic blocks that were not executed. Our CFGs also include counts of the number of times a basic block or a control flow edge appears in the instruction trace. Note that while the CFG represents the possible control flow executed, it does not retain the exact order of execution as the trace does.

3. Related Work

While CFGs contain cycles, they are typically close enough to directed acyclic graphs to which Sugiyama [STT81] type layered layouts are frequently employed. In particular, the `dot` [GKNpV93] layout algorithm of GraphViz [GN00] was used by our collaborators at the start of this project, as well as other computer science research groups we consulted.

Visualization for Program Analysis Graphs. Several program analysis-specific visualizations use a Sugiyama-style layout, such as VCG [San95], CFGFactory [Con], and the works of Balmas [Bal04] and Wurthinger et al. [WWM08]. None of these layouts incorporate internal loop semantics into their layout as we do, though Balmas pre-processed the graph to group hierarchically by function and loop. We use this hierarchy as well, but for navigation. VCG produces a static visualization.

Toprak et al. [TWS14] use a linear layout of aggregated basic blocks formed by converting the graph to a regular expression. Their visualization is for single function CFGs, while ours are interprocedural. Furthermore, they found that although their approach is better for navigating along specific execution paths, CFGs are better for getting an overview of the whole graph. We require both.

Instead of showing a node-link diagram, Krinke et al. [Kri04] highlight source code to show (possibly multi-hop) connectivity and distance from a variable of interest. Our collaborators are interested in how the source code is transformed into instructions and thus a source code view is not sufficient.

Related Graph Solutions in Other Domains. Following paths is an important task in our work (Section 4). We consider customized visualizations emphasizing this task. Borkin et al. [BYB*13] created a visualization for directed graphs of file system provenance. They forgo the typical node-link diagram for a time-aggregated drill-down radial diagram to improve provenance search tasks, but the abstraction does not apply to our tasks, which do not have a time analogue of similar behavior. Pathfinder [PGS*16] provides a query interface to explore paths between two nodes, returning a rank-ordered list and subgraph. Both solutions rely on one or both nodes being known beforehand, but identification of nodes of interest must also be supported by our visualization.

Wongsuphasawat et al. [WSW*18] visualize dataflow graphs of deep learning models, simplifying the graph by hierarchical clustering, edge-bundling, and filtering. We allow hierarchical grouping of nodes, with different semantics due to differing characteristics of our graphs that have fewer repeated sub-structures and support different analysis goals.

4. Task Analysis

Our goal is to improve visual tools for computer science research. We examined why and how control flow graph visualization is used in practice and identified four major goals served by six tasks.

We collaborated with a group of computer scientists researching binary instrumentation, analysis, and compilation techniques. During this collaboration, the group varied in size from seven to twelve with two to four faculty and post-doctoral researchers, two to three graduate students, and two to eight undergraduates, not including the authors. We attended their regularly scheduled (at least weekly) meetings to observe their use of visualization as a group. Initially, the researchers worked solely with traces in a text editor and CFGs rendered as static PDFs.

We interviewed two of the researchers who used CFG visualization individually and observed another two as they used a CFG for technical analysis. Each of these researchers also examined instruction traces. We then developed a brief reflective questionnaire asking what their goals, operations, tools used, and insights found were during a single analysis session. The purpose was to collect data about how they use visualization as they find the need, rather than scheduling a session and disrupting their workflow. Our collaborators submitted six such completed questionnaires.

During the course of our investigation, we identified four major goals our collaborators had in consulting a visualization:

G1: Debug Mental Model. Our collaborators have a mental model of what the control flow graph or instruction trace of a program will contain. The automatic techniques they develop rest on that mental model. However, they expect their mental model is incomplete or incorrect—part of their research effort is developing enough understanding to construct a new solution. They use the visualization to help develop that understanding and find inconsistencies, either at a large structural or more focused scale. When examining an instruction trace, the corresponding CFG visualization is used to provide structure and context to the otherwise flat data file.

G2: Locate a Feature of Interest. A researcher believes some feature may exist in the execution trace or control flow graph. The definition of the feature may be fuzzy or the feature may be quicker to find visually than through other methods such as writing and debugging a custom script. For example, one of our collaborators wanted instruction addresses associated with high degree nodes in the control flow graph. He consulted the node-link diagram to find those nodes and thus the addresses they contained rather than writing a script to list the high degree nodes because performing the task with the visualization required less effort.

G3: Debug an Algorithm. Once a researcher has an algorithm in mind, they may use the visualization as a reference while simulating the algorithm by hand. Should they find an example that does not work, they use the visualization to help identify the cause. They may then refine their mental model as well (G1).

G4: Present, Explain, and Share. Researchers use the visualizations as an explanatory aid in correspondence, during research group meetings, and in papers and proposals.

In the framework of Lam et al. [LTM17], most goals (G1, G3,

and G4) fall under the category "Collect Evidence (Single Population)," specifically the assessing hypotheses step, with G4 being performed by multiple analysts. Goal G2 is is of type "Discover Observation (Single Population)," specifically noting observations and examining their attributes.

To identify smaller tasks in support of these goals, two authors independently coded the questionnaires and observation notes to identify tasks performed by our collaborators. We then used affinity diagramming [BH99] to split, merge, and group tasks. We discovered the following tasks, some of which support multiple goals. We describe these tasks and their relation to low level visualization tasks using the task taxonomy for graph data of Lee et al. [LPP*06] where applicable.

T1: Follow Control Flow. In a raw execution trace, control flows instruction-by-instruction, line-by-line, but it is difficult to relate each instruction to the higher level structure of the program. When viewing the CFG as a node-link diagram, researchers follow edges or paths to understand how the instructions relate to the basic block structure of the program. These tasks are thus "Browse - Follow Path" in the Lee et al. taxonomy. To determine an exact ordering of events, the researchers would then relate the node-link diagram back to the raw trace. Following control flow is performed in both G1 (debug mental model) to gain an understanding of the program and G3 (debug algorithm) as part of simulating an algorithm. Frequently this process is repeated during presentation (G4) as well.

T2: Identify Known Structures. To understand the relationship between the instructions and higher level program constructs (G1) or in search of a particular structure (G2), researchers typically want to identify common constructs such as loops, functions, and conditionals. Within a loop, they will identify sub-structures of interest, including the entry and exit basic blocks and the loop back edge. To correlate with knowledge about the code, they may be interested in the loop nesting depth. While identifying exit nodes via their exit edges can be considered a "Browse - Follow Path" task, for the most part these are what Lee et al. describe as "High-Level Tasks." Once a structure is found, the instructions associated with it are often queried—this is an "Attribute - Nodes" task.

T3: Examine Specific Regions. Once a region of interest is identified, either visually (T2) or by some other method such as a custom script, the researchers may want to narrow their exploration to that region. This task occurs both when trying to refine their mental model (G1) or apply a planned algorithm to that region (G3). Often the researcher searches for a specific instruction or address ("Attribute - Nodes"). To simplify their search, they sometimes preprocess the data to filter out unimportant data.

T4: Identify Outliers. When debugging their mental model (G1), researchers search the visualization for anomalies—pieces of the visualization that "don't look right." Features of interest (G2), such as high degree nodes, are also often assumed to appear as outliers. The latter falls under "Attribute - Nodes" and the former as "High-Level" in the Lee et al. taxonomy.

T5: Identify Feature Repetition. To optimize a program, researchers target frequently executed instructions. Thus they want to identify highly traveled elements in the CFG. They may also be interested in branch divergence—the proportion of iterations that follow each flow from a conditional. Infrequently executed instructions are important to note for maintaining program correctness after transformation. These queries are done when debugging their mental model (G1), searching for features related to iteration count (G2), and debugging their proposed algorithms (G3). This data is usually modeled as weights on the nodes and edges, making them "Attribute - Nodes" and "Attribute - Links" tasks.

T6: Alter Visualization. When sharing findings (G4), our collaborators would sometimes annotate a static image to highlight certain instructions, nodes, or edges. Sometimes they would pre-filter or aggregate the data to obtain a more simple visualization for presentation or for exploration (G1, G3). A few times, they re-drew a sub-graph of interest manually either for presentation or their own understanding, usually "fixing" the layout.

Summary. Many of these tasks, such as loop identification and following control flow, are operations that are basic building blocks for the algorithms the researchers are trying to develop. We use this observation as an additional guide to our design process.

5. CFGExplorer

The design of CFGExplorer [†] is supported by our task analysis (Section 4) and an iterative design process that included weekly meetings with our collaborators as well as short informal interviews with researchers outside the group. We observed our collaborators would frequently present and discuss both the CFG and the raw trace, attempting to show the same instructions in both at the same time. Therefore, we designed our system around two main linked views: a CFG view and a trace view. These are displayed side by side as shown in Figure 1.

Our collaborators perform both path-following and attribute look-up tasks on the CFG, so node-link diagrams, which they were using, are more appropriate [GFC04, KEC06] compared to matrix-based representations. Furthermore, our collaborators frequently drew node-link diagrams when analyzing data, indicating this representation matches their mental model. While node-link diagrams can be difficult to interpret at large scale, we can take advantage of the sparseness of CFGs as well as domain-specific structures to produce a more interpretable layout. We describe this layout and our graph encoding choices in Section 5.1.

Both the CFG and the trace can be inputs to the algorithms our collaborators are developing. However, the trace represents ground truth and thus our collaborators would refer back to it after finding insights in derived structures. Thus, we left the trace in its raw text format but added several features to link it with the CFG and provide visual cues about basic operations their algorithms might perform. We describe linking and highlighting in Section 5.2.

5.1. CFG View

We designed a node-link CFG view which emphasizes loops. Algorithms developed by our collaborators often use the identification

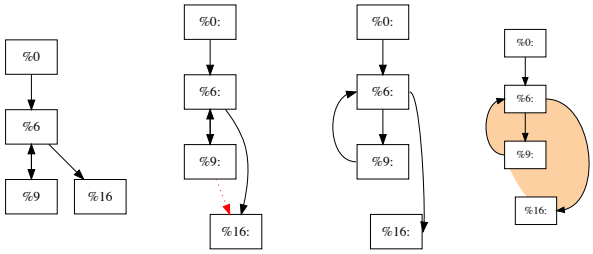
[†] http://github.com/hdc-arizona/cfgexplorer

of loops and their components so we design our CFG layout to preserve these structures. Panning and zooming is supported through standard 'click and drag', 'pinch zoom,' and 'mouse scroll' interactions. We describe the layout, style, and interaction choices below.

Domain-specific Layout. While CFGs contain cycles, the overall structure is generally layered, making a Sugiyama [STT81] style layout appropriate. We chose a vertically layered layout as the vertical direction was associated with sequence by our collaborators to the point they vehemently objected to a horizontal layout of a simple five-node graph in a grant proposal, concerned it would confuse their community. In another instance, a researcher expressed confusion with the node placement in a general layered layout being presented by a colleague. The colleague responded that it was an artifact of the layout and did not imply sequence as the first researcher had inferred.

We chose to draw the graph with the instructions associated with each basic block visible in the nodes to avoid researchers having to perform a look up operation and to support tasks T1 and T3. We also provide a hover lens for each node which shows the instructions at a fixed size using a tooltip, even when the node is small.

Our collaborators struggled to interpret general layered-graph layouts as the number of nodes increased. We decided to incorporate the domain-specific semantics into the layered-graph style, specifically the expected structure of a loop and its components as described in Section 2. We forced non-loop nodes reachable by a loop's tail node to appear after (below) the loop by adding invisible edges. The invisible edges increase the layer (rank) of the non-loop nodes as viewed by the layout algorithm. We experimented with adding invisible edges from all the loop nodes instead of just the tail node but found the produced layouts distorted the positions of the other loop components of interest. Figures 3a and 3b show a before and after layout using this approach. These changes support tasks T1, T2, and T4. Though we provide this layout, we also provide functionality to move nodes while maintaining connectivity in support of task T6.



(a) Original. (b) Altered. (c) Back edges. (d) Boundaries.

Figure 3: CFG of a simple `while` loop as generated by LLVM [LA04]. To modify the results of a general layered layout (a), we add an invisible edge (red) from the tail node to all outer nodes (b). In `dot`, we also route the back edge using ports (c) and color the loop based on its bounding nodes and edges (d).

When using `dot` as our base layout, we found it necessary to explicitly route the back edge using ports (Figure 3c), but not so when using the Dagre [cpe] library. We ultimately chose Dagre to layout

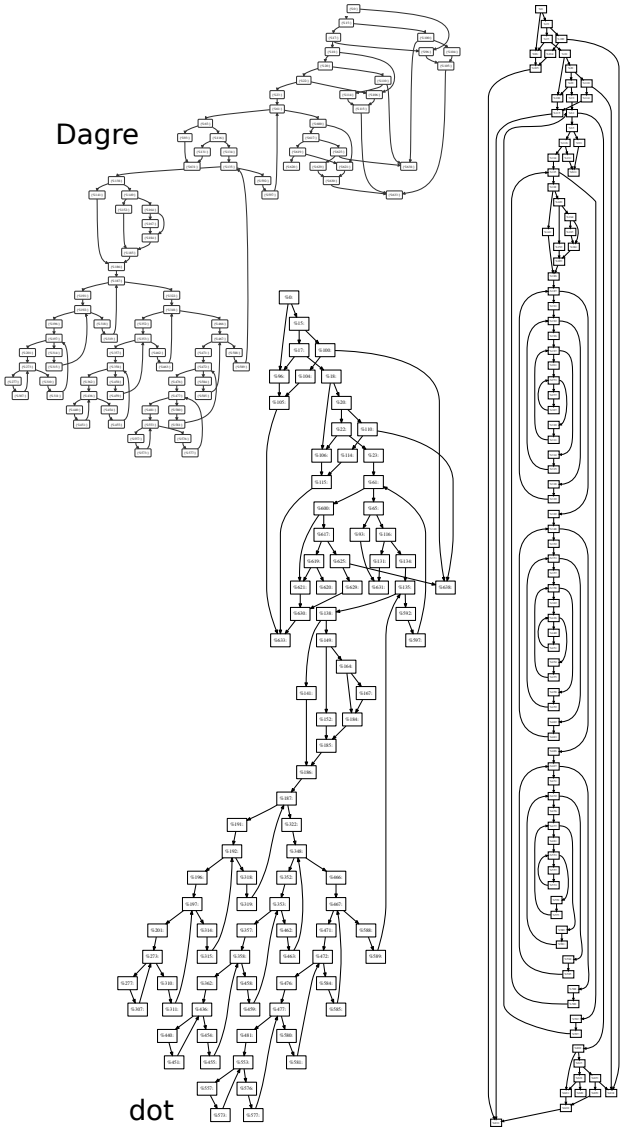


Figure 4: CFG of the mini-flux-div benchmark [OSG*14] as rendered by Dagre, `dot`, and our domain-specific approach. Our layout reveals multiple simple, but deeply nested loops.

our modified graph in CFGExplorer due to its time performance, but maintain the `dot`-based version as well to aid in prototyping and presentation. Figure 4 compares the layouts of Dagre and `dot` to ours (created using `dot` on the modified graph) on a static CFG of the mini-flux-div benchmark [OSG*14].

One of the researchers outside our primary collaborator group said he preferred orthogonal edges like in VCG [San95]. He found them easier to follow and label. We showed both orthogonally routed and spline edges to our primary collaborators. They indicated a preference for the spline edges, in line with the use of curved edges in domains such as metro-map layouts [RNL*13, KNT13].

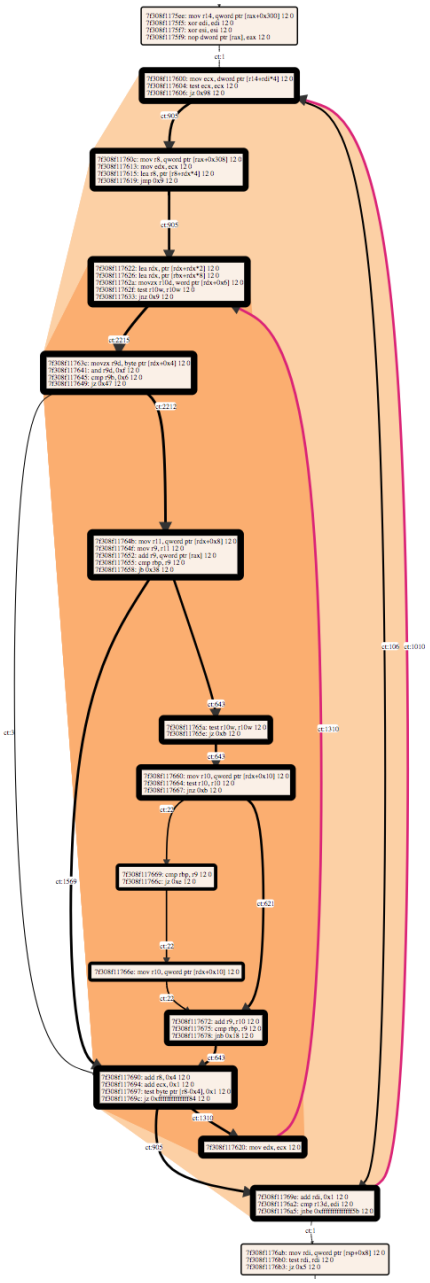


Figure 5: Loop boundaries (orange) are drawn as the convex hull of the loop nodes and back edge, darker indicates loop nesting. This approach does not capture all loop edges (left) as doing so can create difficult-to-interpret results in more complicated graphs.

Loop Boundaries. To further aid researchers in quickly identifying loops (task T2), we draw a boundary enclosing all nodes in the loop as a solid-colored background. The boundaries are computed as the convex hull of all loop node boxes and the back edge using the Graham scan algorithm.

We considered including all loop-internal edges in the boundary calculation. We checked each pair of adjacent nodes in the convex

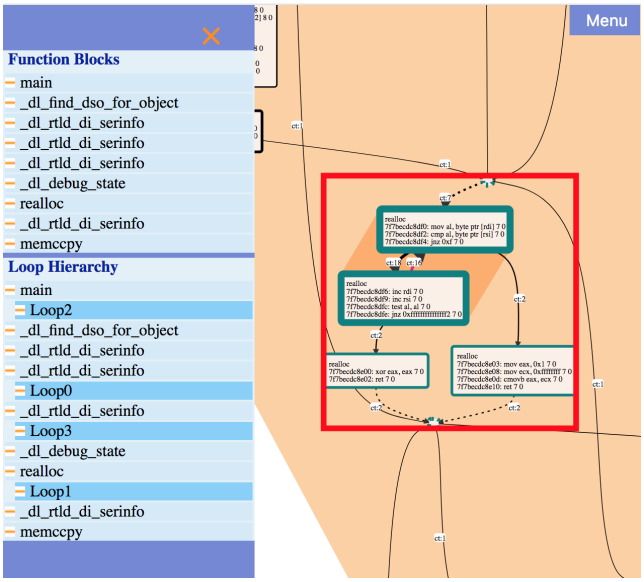


Figure 6: Functions and loops can be automatically centered via the slide-out navigation menu. A bounding box is drawn around the function or loop and all member nodes are selected (teal border).

hull. If an edge existed between them, we replaced the previously calculated convex hull edge with the graph edge. While including all loop-internal edges in the convex hull worked well for simple CFGs (Figure 3d), it was difficult to interpret in larger graphs with more edge crossings and complex code behavior and it was computationally expensive. However, since the back edge is of special importance in defining the loop, we included it in our convex hull calculation, ensuring it is within the loop boundary background.

Non-loop nodes may be included in the boundary. We decided against a non-convex approach as it would not be "loop-like" and would be computationally expensive. We include a feature to explicitly highlight nodes of a given loop to ameliorate this issue.

We chose orange for the loop boundaries to not be confused with the highlighting in Section 5.2. We further encode loop nesting depth with the lightness of the orange coloring—more deeply nested loops appear darker as shown in Figure 5.

Encoding Loop Components. To support task T5, we encode the number of times a node was executed using border thickness and the number of times an edge was traveled using line width, where higher is thicker. As there is a large amount of variation, both are calculated on a log scale. While most edges are drawn in black, we draw all loop back edges in magenta to make them quickly identifiable as desired by task T2. These encodings are shown in Figure 5.

Some of our CFG data is augmented with extra nodes and links added by our collaborators for their analysis. These are marked as such in the data. We use dotted lines and decrease the size of these nodes to communicate they are not basic blocks or dependencies from the trace.

Function and Loop Interaction and Navigation. We provide a slide-out navigation menu for functions and loops, as shown in Fig-

ure 6. Clicking on an entry will center the graph view on the corresponding structure and highlight all member nodes. We experimented with hovering, but found our collaborators used the menu for directed search rather than browsing and thus vastly preferred the view only change on deliberate command. The menu also serves a visualization purpose, summarizing the structure of the data in terms of functions and loops and showing containment and nesting via indention.

Users can double-click on an entry to aggregate all member nodes into a single "super-node," thus decreasing the number of nodes shown in the graph. Super-nodes maintain all in and out edges of the function or loop they represent. The node aggregation is similar to the functionality in GrouseFlocks [AMA08] and ASK-GraphView [AHK06], as applied to a layered layout instead of a force-directed one. We place the super-node at the centroid of its child nodes to preserve the existing layout. We maintain the ability to move and delete (super-)nodes from the view. Execution counts on edges between super-nodes are also aggregated.

5.2. Linked Navigation and Highlighting

As we observed our collaborators generating insights from a CFG visualization and then consulting the corresponding portion of the raw trace, we designed our system to aid in these operations by linking the navigation and highlighting of both data structures. We also describe two highlight encodings, temporal highlighting and analysis highlighting, which support common operations performed by our collaborators as well as building blocks for the algorithms they design.

Positional linking and highlighting. Navigation in one view causes the other view to center on the selection. Scrolling the trace view will center the CFG on the first basic block visible in the trace. We do not alter the zoom as users found doing so disorienting. Hovering over a node in the CFG cause the raw trace to re-center so the hovered basic block is placed at the top. Hovering also causes the node boundary to be drawn in teal and the corresponding basic block in the trace to be outlined in teal. Clicking on an edge shows a preview of the incident nodes of the edge. Similar to that of CGV [TAS09], this feature is designed for long edges where one or both of the incident nodes are not visible.

Temporal highlighting. A CFG does not contain the exact order of instructions executed, which is needed to understand the exact sequence of events. This data is included in the trace. For a higher level understanding of the iterations (tasks T1 and T5) and for insight on whether a proposed algorithm can operate on the CFG alone or requires the trace, we use linked highlighting to show sequence information from the trace. In particular, we highlight the basic blocks visible in the raw trace view and their corresponding nodes in the CFG, using a gradient to denote their order. The gradient is from light to dark blue, with lighter meaning earlier in the sequence. Scrolling through the trace thus produces an animation of the flow of control. Due to the positional linking, the highlighting portions of the CFG remain centered. Figure 7 demonstrates this highlighting.

Analysis highlighting. Our collaborators frequently develop analysis programs for finding specific instructions, addresses, or registers both to help understand program behavior or as part of a larger solution in their research. We incorporate a feature that highlights elements returned by such programs as they appear in both the trace and the CFG. We implemented this highlighting for two such analysis techniques supplied by the researchers, *backtainting* and *upward exposed read (UER) detection*. The backtaint of an instruction finds all other instructions that may have affected it through data flow. It is helpful for understanding the provenance of a particular instruction. A UER is a read instruction in a loop to a location that has not yet been written in that iteration of a loop. Detecting UERs is helpful in determining if a loop can be parallelized. The program supplied by our collaborators allows filtering of returned UERs to those where the write occurs within the loop (in-UERs) and those where it does not (out-UERs).

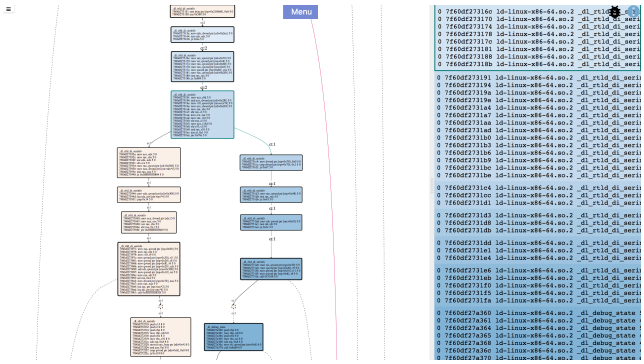


Figure 7: The visible portions of the trace and corresponding nodes in the CFG are colored using a blue gradient. Researchers can see the sequence of a portion of the trace in the CFG. Scrolling acts as animation to understand execution.

When a distance metric is available, such as the number of instructions executed between the queried instruction and the found instruction in backtainting, we use the shade of purple to indicate that distance, where darker means closer to the query. The backtaint analysis does not take control dependency into account and hence the instructions returned by the analysis are not arranged in the exact order of closeness to the queried instruction. Figure 8 shows an example of backtaint highlighting, the dark purple line is closer to the query than the lighter purple line.

Highlighting design considerations. Our design has several forms of highlighting that can activate simultaneously which could cause confusion. Non-color channels such as region and size would alter the layout and we did not want to add shape markers to the text. We tried several colors for highlight and gradient with our collaborators and selected the one they found most visible. We interpolated in perceptual color space. We determined highlight precedence by considering how our collaborators used each highlight. Backtaint, UER, and loop highlighting, as well as future analytics that would use the same semantics (e.g., adding similar functionality for program slicing), are very deliberate user choices, and thus take precedence over the navigational highlighting of hovering and scrolling. As hover highlighting is a quick and basic action, we chose teal outlines for the encoding so it could be composed with the other area-based highlights. To give users more flexibility in avoiding in-

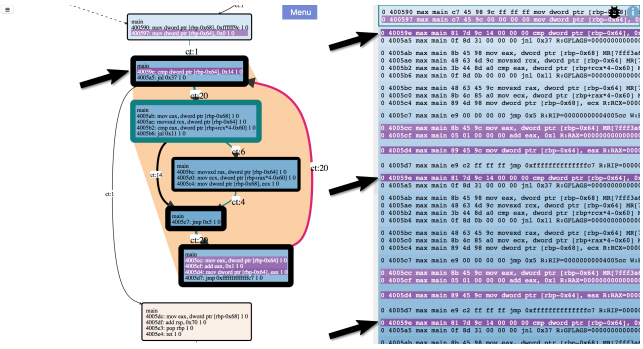


Figure 8: The backtainted addresses from the query address 40059E (denoted here with a floating arrow) are highlighted in purple. Darker implies a closer address in the data dependency chain.

terference and in focusing on particular attributes, we allow any highlight feature (color, thickness, background) to be turned off.

5.3. Implementation

CFGExplorer is a browser-based client-server application. A Flask server runs program analyses (e.g, loop-finding, backtainting) and communicates them in a RESTful manner. The client is written in Javascript using d3js [BOH11] with Dagre [cpe] as the base layout. Users can export the CFG view as an SVG (task T6). Though CFGExplorer was designed for both CFGs and traces, a CFG-only mode without trace-dependent features exists to support a broader range of uses, such as examining statically-generated CFGs.

6. Evaluation

To evaluate the effectiveness of the CFGExplorer, we conducted hour long evaluation sessions with six participants, four of them from our group of collaborators. Of the four collaborators, one had recently joined the project and was thus not part of the formative design process. We also conducted short demonstrations at a computing conference to seek feedback outside the research group. We deployed CFGExplorer to our collaborators and observed its use over the course of our development.

6.1. Evaluation Sessions

We scheduled evaluation sessions with six participants. Four of them are our collaborators of which three are students and one holds a Ph.D. degree. One of the students joined the project a month before the evaluation session as was not involved in the formative design process. We refer to them as P1, P2, P3, and P4 respectively, with P4 being the participant new to the project. P1 had been using our tool for analysis during deployment, while the other two participants had seen the tool but not interacted with it. The other two participants, P5 and P6, had not seen or used CFGExplorer before. P5 is an undergraduate student who had been working on a different program analysis research project for one semester. P6 is a graduate student whose research is in the area of security, including reverse engineering. For these sessions, the participants used a laptop provided by the authors rather than their usual work machines.

Our evaluation sessions started with a ten minute tutorial explaining the encoding, interactivity, and features of CFGExplorer. Next, the participants were asked to complete the tasks described below. We then encouraged free exploration, which all participants did. We closed the session with a semi-structured interview and debriefing. Each session was conducted individually, with one author directing the participant and another author taking notes.

Participants were asked to perform evaluation tasks of increasing difficulty and to "think aloud" while doing so. The evaluation tasks E1 - E5 are listed below along with their corresponding tasks from the task analysis of Section 4.

E1. Find a specific address in the trace and center the node containing it in the CFG. (Task T3)
E2. Find a specific function. (Task T3)
E3. Find a specific loop and identify its back edge and loop exits. (Tasks T2 and T3)
E4. Find the loop with the most iterations. (Tasks T2 and T5)
E5. Free Exploration: Explore instructions marked by automated analysis. (Tasks T1 and T4)

The evaluation dataset was generated by other members of the collaborator group and was one of a set of datasets collected for their analysis. The programs we evaluated CFGExplorer on were different from the programs our participants were working on at that time. Furthermore, we did not time participants. We describe our observations of task performance below.

E1: Find a specific address. Participants P1, P3, P4, and P6 immediately used the browser's search feature to find the given address in the trace. They then scrolled the trace to center the node in the CFG view. P2 needed to be reminded of the command for the browser search, but then followed in the same manner. P3 remarked they would have preferred dragging to scrolling in the trace view.

P5 started by performing a backtaint on the address and then explored the results. After being reminded of what the backtaint feature does, the participant said "Usually I would just use Ctrl-F" and questioned if it was okay. Once given approval, the participant completed the task using the browser's search feature.

E2: Find a specific function. Participants did not exhibit difficulty with this task. They opened the navigation menu and located the specified function in the functions block and used it to highlight and center the function in the CFG view before closing the menu.

E3: Find a specific loop and its components. Similar to E2, all participants used the navigation menu, this time with the loop hierarchy block, to find and center the specific loop. After closing the navigation menu, they identified the back edge as the one colored magenta. To find the exit nodes, they scrolled to the bottom of the loop, loosely centering the bottom of the background color and identified edges leaving the loop.

E4: Find the loop with the highest iteration count. The participants used different strategies for this task. P1 and P5 first looked for loops in the CFG view that had thick borders and edges because line width encodes trip count, with P1 appearing to use the orange background to verify they were loops. After finding candidate loops, they used the navigation menu to center the graph on each loop and zoom in to read the trip counts. P2, P3 and P4

started from the navigation menu, centering and zooming on every loop and reading the trip count on the back edge. P6 located all the loops using just zooming and panning by identifying orange loop boundaries. He remarked he would use the navigation menu if there was a large number of loops.

P4 took some time to recognize the largest loop, which extended beyond the screen at his zoom level. P5 incorrectly assumed the order of loops listed in the navigation menu was based on loop iterations. (The branch order of the loop hierarchy is arbitrary.) None of the participants used the "Show Loops" feature to color the loops.

E5: Free exploration. Participants were encouraged to analyze the graph and to try the backtainting (P1, P2, P3, P5, P6) or upwards exposed read (UER) detection (P4) features, so we could observe how to better integrate more complicated program analysis techniques into the visualization.

E5: Free exploration - Backtainting. P1 decided to explore a dataset of his choosing. The dataset was from a linked list program which he had last looked at six months beforehand. He wanted to work with it again to think through how his in-development algorithm would handle the dataset. From the graph view and his knowledge of the source program, he identified a loop of interest. He backtainted on an address from one of the instructions late in the loop, resulting in most loop addresses being highlighted. He remarked that the backtainted instruction therefore belonged to one set of instructions used in his research. He typed another address and saw that fewer addresses were tainted, meaning the instruction belonged to the other set. He continued with several more instructions, writing down which group each address belonged to on a sheet of paper.

When asked, P1 explained that he wanted to understand direct dependencies and that those will be the ones tainted first ("the immediate taint"). Using the difference in the taint highlighting color, he would locate such an address and use it as input for the next taint. He suggested a limit on the number of tainted addresses that would be highlighted to focus on the closest ones. Using the sheet of paper from earlier, he drew a subgraph showing the taint dependencies.

The other participants explored the original dataset. P2 operated similarly to P1—starting with an address near the end of a loop and polling which addresses resulted in many versus few highlights.

Participant P3 asked for a reminder on how to use the tainting feature. She first found some instructions of interest and explained what operations the instructions were performing and how she believed that related to the original program. She first tainted an instruction she identified as being a loop increment because she was curious what would happen. After tainting, she pointed out where the loop initialization was with respect to the increment. She discovered an untainted instruction in the loop and wondered why it had not been tainted, saying: "Is this iterating over an array? Yes, max iterates over an array, so it's looking for the boundary of an array." She was thus able to determine why the instruction was not part of the taint with the help of knowledge of the program.

Participant P5 wanted to verify her understanding of the backtaint feature, hypothesizing that the order of addresses in the backtaint does not follow the execution order of instructions in the trace. She used the backtaint analysis feature on an address inside a loop. The result showed that some addresses were highlighted after the input instruction. She further hypothesized "so the lightest color would be the initial value... the very first one that starts affecting the others ones."

Participant P6 first searched for a print function using both the browser search and the navigation menu, because print functions indicate program output. The program did not have a print function, so he instead searched for return statements, immediately locating one in the program exit node. He tainted the address and noted it can be hard to visually find all the highlights when zoomed far out. He instead scrolled through the trace to find the addresses, remarking the process gave him a sense of execution count.

Focusing on a single loop, P6 identified an instruction which causes the loop to exit and performed a taint on that address. Using the taint similarly to P3, he explained what the loop is doing, identifying the iteration instructions and those that handle arguments.

Finally, P6 noted that there is a lot of Python start up code in the data. He used the function list in the navigation menu to collapse probable Python functions into single nodes. He said of the function collapsing: "I think that's the most helpful feature especially for the type of things like reverse engineering."

E5: Free exploration - UER Detection. Participant P4 experimented with the UER analysis, having previous experience performing such analysis manually on smaller traces in a different assembly language. Using the feature that returns only the UERs caused by writes within loops (in-UERs), he quickly identified the source of the UER by pointing to the bottom node of a loop and noted all highlighted accesses of the value happen before (above) his identified instruction. He inferred: "So that has the back edge, which means that value might be updated when we use it in the next iteration of the loop. So we have to be careful between iterations."

The UER analysis was in-development by another member of P4's research group. P4 decided to verify its behavior. He first highlighted out-UERs, those that use values not defined in the loop. He identified a highlighted register in one loop instruction. He then visually searched the other highlighted lines in the loop, verifying that register is never assigned and thus is correctly considered an out-UER. P4 said the visualization is helpful in checking correctness of the UER detector. He suggested adding a feature to filter the UERs so he can focus on inspecting just a few at a time.

P4 was curious as to whether the UER detection used trace information. We brought up a dataset from a simple array summation program with which he was familiar. P4 hypothesized that if the UER detector uses the trace, it should show no UERs, but if it does not, it might have false positives. P4 highlighted the detected UERs and examined their corresponding lines in the trace. He concluded the UER feature is not using trace information.

6.1.1. Semi-structured Interviews

After the tasks were completed, we conducted semi-structured interviews with the participants starting with specific questions on the ease or difficulty of performing certain tasks. Then we asked more general questions about usability and desired features. We summarize themes in the responses below.

Participants liked the linking of the CFG and execution trace representations. P1 remarked "So what I see when I spit it out on the command line is just a list of instructions. That doesn't give me any context to how the instructions are related." P4 said CFGExplorer was useful for his research and "it really simplifies navigating through a trace because it identifies the loops for you." P5 said "This [CFGExplorer] is so convenient." P2 said "It's really nice to be able to explore and connect information especially between the CFG and the trace. There's nothing else we have that does that. It's really invaluable" and "...it will help me solidify what information we can get from the trace and we can get from the CFG." P3 also discussed the need to connect the trace and control flow when requesting more tainting features: "...it seems like there needs to be control flow tainting as well. We all need to think about how the forward expression tree building interacts with the trace."

The graph layout was also appreciated by participants. P1 said "If the instruction is higher up, it's probably executed first" and also remarked he used the highlighted back edge as a cue to ascertain loop extents. P3 noted that "the jumps are already pretty clear visually." Participants P2, P3, and P4 preferred the layout be static or locked rather than having the ability to alter the graph visualization. P1 said being able to move the nodes was useful when making figures for presentation, but had not needed to use it for analysis.

Participants generally found the tool usable. P1 stated "It was intuitive. I feel like I could open this up and figure out how to zoom" and "Scrolling and centering is pretty much perfect now." P6 said both looking at the program and performing the analysis was very easy. P4 liked the ability to disable some of the encoding, noting that node thickness is useful when zoomed out but unneeded when zoomed in. P5 liked the ability to search and highlight everything. Both P1 and P3 stated they preferred clicking to hovering when centering the graph from the navigation menu. P3 did not find the hover lens useful and turned the feature off. P1 said he did not currently use the loop and function collapsing in CFGExplorer, but anticipated needing it soon. P6 emphasized that the loop and function collapsing was an important feature to him.

When asked about annotation, P1 and P2 said they preferred to take a screenshot and then use a separate program, e.g., one that allows the user to draw with the stylus. In contrast, P3 suggested being able to save the visualization's state along with a note, highlight features, and add tags. P4 stated annotations would be useful, noting "most of the communication in our group is taking a screenshot in your tool", and then sending it around with comments. P1 and P2 said it was possible built-in annotation would be useful, but would prefer priority be given to other features.

Participants suggested adding the ability to filter columns in the instruction trace (P3), a list view of backtainted addresses (P1, P6), heuristically collapsing some functions on startup (P6), and automatic highlighting of other domain-specific constructs such as upward exposed reads and slices (P3). P3 also asked for a bright and larger highlight color for the linked node highlight. P6 suggested more color difference between successively highlighted blocks in the linked gradient coloring.

Both participants from outside our collaborator group (P5 and P6) asked if the tool was available for their own use and were given a link to the demonstration site. P6 suggested CFGExplorer could be used to help students on their reverse engineering assignments, noting limitations of other tools. He also was interested in using it with other assembly languages. He suggested the dynamic support could help with obfuscated code, as the dynamic CFG may remove some of the obfuscation.

6.1.2. Limitations in Evaluation Sessions

The findings of the evaluation sessions are limited by the small number of participants, three of whom were regularly in attendance at research meetings used in our task analysis and subsequent design iterations. Additionally, four of the participants (those from our collaborator group) had some familiarity with the evaluation dataset, though none had recently been analyzing it when the session was run. This familiarity may have enhanced their productivity with CFGExplorer.

The evaluation sessions were designed to last one hour, including the interview portion. The time constraint limited the kind of analyses we could observe. As CFGExplorer was designed to aid researchers, a more realistic workflow using CFGExplorer would take significantly longer and might involve using other analysis tools and methods in tandem.

6.2. Conference Feedback

We sought feedback regarding CFGExplorer at a parallel computing research conference. Three researchers (R1, R2, and R3) with an interest in CFGs were given either a short demonstration or a link to the prototype and sample data. None had previously seen or heard about our work. All three researchers responded positively to the domain-specific layout and interactivity and asked for the URL. R1 particularly liked the loop and function collapsing capabilities.

The researchers also made several suggestions. R1 and R3 requested more flexibility with multivariate data associated with the CFG, such as time and branch count for vectorization. R3 in particular suggested the ability to color edges by more types. R1 wondered about the scalability of CFGExplorer because he looks at CFGs from big applications.

6.3. Deployment

CFGExplorer has been in development for over an year and has been deployed among our collaborators concurrently with development for more than 6 months. Informally, the visualization received positive feedback during meetings with focus on the linked graph and trace highlighting and loop and function navigation. Prior to the deployment of CFGExplorer, our collaborators would show PDFs generated using a general layered layout. Since deployment, some collaborators now present using CFGExplorer instead, though one still uses a static PDF from a general layered layout.

During research meetings, we observed collaborators simulating the steps of their proposed algorithms on example data in CFGExplorer for the rest of the group. Screenshots from CFGExplorer were also used to share findings in emails, presentations, and proposals. In one meeting, a collaborator shared an annotated screenshot from the tool, marking particular instructions with colors. We followed up with the collaborator asking how he had generated the

screenshot. He said he used the navigation menu to navigate to his loop of interest and then took the screenshot and annotated it using a drawing app on his tablet.

During another session, a researcher pointed out long chains of degree-2 nodes visible in CFGExplorer. The insight was that the data contained an unexpectedly large number of untaken jumps, which would significantly slow down their proposed algorithm. They began considering additional approaches to mitigate the problem. Though datasets with this feature had been viewed with the general layered layout in static form, they had not noticed this feature before using CFGExplorer.

Case Study: Developing a Specialized UER Detector. The researchers are in the process of developing a specialized program to detect upward exposed reads (UERs) of interest to their larger project. They initially programmed the detector to use only the CFG, but believe they will need some information from the trace to refine the results. They used CFGExplorer to inspect the output with the goals of (A) determining if the detector is behaving as expected (correctness) and (B) determining the minimal amount of information they need from the trace. One senior collaborator remarked "I think bringing things up in CFGExplorer is the right way to do this," because it shows the correspondence between the trace and the CFG.

The researchers had a sub-group meeting, limited to stakeholders in the UER detection. The researchers used CFGExplorer to highlight the results of the UER detector in a sample program. The researchers verified the correctness of the results by checking the addresses touched by the UERs. While doing so, they noticed the register `rbp` was returned frequently. After discussing why the program detects the register, they decided it could be ignored by their detector program.

Another discovery made while looking at the UER results came from examining repeated basic blocks in a short loop. Using CFGExplorer, they quickly recognized a repeated portion of the trace corresponding to the loop. One researcher noted that despite a nonzero value being added to the same register each iteration, the value in the trace did not seem to change between iterations. The researchers hypothesized an error in the tracing utility and made a note to consult with the collaborators in charge of those tools.

7. Discussion and Lessons Learned

Our evaluation shows that computer science researchers were able to perform the Section 4 tasks using CFGExplorer and found it useful enough to integrate with their existing workflow. The domain-specific layout and interactivity in the form of correlation with the trace as well as navigation to loop structures emerged as their most well-regarded components of the visualization system.

During the free exploration task (E5) in the evaluation sessions and in research group meetings, the use of analysis highlighting (e.g., due to backtainting) seemed to help researchers more quickly orient themselves in the assembly. We frequently observed them explaining what a group of instructions did and relating them to program constructs such as the loop iterator, thus supporting task T2 (Identify Known Structures).

However, researchers' thoughts on potential integrated annotation capabilities, which might further help them mark known structures, were mixed. We observed the researchers who preferred using general stylus-based drawing tools on screenshots were also likely to draw graphs on paper, as P1 did during our evaluation sessions. We infer annotation capabilities that incorporate creating subgraphs, in addition to the highlighting proposed by P3 and R3, could be beneficial, but would need to be designed in a flexible enough manner and may need to support touch and stylus inputs.

Though the ability to alter the layout was a task we identified during our initial analysis, the feature seemed detrimental during our evaluation sessions, causing us to add the ability to lock the layout. In part, we believe the ability to move nodes got in the way because layout alteration is a relatively short part of the analysis process. Our domain-specific layout may have also lessened the need to "fix" the layout in performing the analysis. However, P6 and R1's enthusiasm for collapsing functions and loops indicates that more support for altering the layout in a domain-specific fashion, i.e., locally updating the layout to preserve more of the structure in terms of edge routing and node placement, is needed to aid researchers when they want to make changes.

No participants used the "Show loops" feature, which highlights the nodes of each loop with a different color, to identify loops in the evaluation. We infer that coloring the background using the convex hull of each loop combined with the loop hierarchy in the navigation menu was sufficient for loop exploration.

Our design was based on data collected from computer science researchers via observation, interviews, and questionnaires as well as consideration of existing approaches. On reflection during our analysis, we noted that many of the tasks performed by the researchers are analogous to common building blocks used by algorithms in their community. This insight opens another avenue for task and requirements analysis—the APIs of their common libraries and toolkits. Once we brought this observation to one of the principal investigators of the research group, they remarked framing their needs in those terms also helped them think about their own uses of visualization. We expect the design of other visualizations for computing research and well as further design to increase the capabilities of CFGExplorer can benefit from careful examination and classification of features available in common libraries.

8. Conclusion

We have presented a design approach, task analysis, and prototype system for program analysis with control flow graphs. We observed that computer science systems researchers often approach a CFG visualization in the same way the algorithms they develop do. We used this observation to guide the design of our graph layout, analysis features, encodings, and interactions. We incorporated the identification of loops and their components, a basic operation, directly into the graph layout and encoding. Correspondence between the CFG and the trace was done with coordinated views. We included backtaint and UER analysis as highlighting features and expect their design to transfer to similar analyses. We evaluated our prototype design through deployment, informal interviews, and formal evaluation sessions. Researchers particularly liked the domain-specific layout and interactivity and found the prototype usable.

Our collaborators have integrated CFGExplorer into their workflow, using it for analysis and collaboration.

We plan on providing a general methodology for incorporating basic analysis operations in a manner similar to our backtaint and UER highlighting, targeting formats available from common tools such as LLVM. We foresee this requiring multivariate techniques [KPW13]. We also plan to closely examine issues of scalability. While CFGExplorer increased the size of graphs that our collaborators could examine and improved upon their analysis process, applicability to data larger than a few thousand nodes is untested and may require new techniques. We expect hierarchical layout approaches that integrate more domain information will be needed as well as support for then altering the graph, similar to [GSE*14].

9. Acknowledgements

We thank the individuals who participated in our evaluation and our collaborators in the Science Up To Par project in the Department of Computer Science at the University of Arizona. We further thank the National Science Foundation for funding this research under award III-1656958.

References

[AHK06] ABELLO J., HAM F. V., KRISHNAN N.: Ask-graphview: A large scale graph visualization system. *IEEE Trans. on Vis. and Comp. Graphics 12*, 5 (Sept 2006), 669–676. doi:10.1109/TVCG.2006.120. 7

[AMA08] ARCHAMBAULT D., MUNZNER T., AUBER D.: Grouseflocks: Steerable exploration of graph hierarchy space. *IEEE Trans. on Vis. and Comp. Graphics 14*, 4 (July 2008), 900–913. doi:10.1109/TVCG.2008.34. 7

[Bal04] BALMAS F.: Displaying dependence graphs: a hierarchical approach. *J. Soft. Maint. and Evolution: Res. and Prac. 16*, 3 (2004), 151–185. 3

[BH99] BEYER H., HOLTZBLATT K.: Contextual design. *interactions 6*, 1 (Jan. 1999), 32–42. doi:10.1145/291224.291229. 4

[BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D3: Data-driven documents. *IEEE Trans. on Vis. and Comp. Graphics 17*, 12 (Dec 2011), 2301–2309. doi:10.1109/TVCG.2011.185. 8

[BYB*13] BORKIN M. A., YEH C. S., BOYD M., MACKO P., GAJOS K. Z., SELTZER M., PFISTER H.: Evaluation of filesystem provenance visualization tools. *IEEE Trans. on Vis. and Comp. Graphics* (2013). 3

[Con] Control flow graph factory. http://www.drgarbage.com/control-flow-graph-factory/. (Accessed on 12/10/2016). 3

[cpe] cpettitt/dagre: Directed graph renderer for javascript. https://github.com/cpettitt/dagre. (Accessed on 12/10/2016). 5, 8

[FOW87] FERRANTE J., OTTENSTEIN K. J., WARREN J. D.: The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349. doi:10.1145/24039.24041. 2

[GFC04] GHONIEM M., FEKETE J.-D., CASTAGLIOLA P.: A comparison of the readability of graphs using node-link and matrix-based representations. In *Proc. IEEE Symp. on Info. Vis.* (2004), pp. 17–24. 4

[GKNpV93] GANSNER E. R., KOUTSOFIOS E., NORTH S. C., PHONG VO K.: A technique for drawing directed graphs. *IEEE Trans. on Soft. Eng. 19*, 3 (1993), 214–230. 3

[GN00] GANSNER E. R., NORTH S. C.: An open graph visualization system and its applications to software engineering. *Software – Prac. and Exp. 30*, 11 (2000), 1203–1233. 3

[GSE*14] GLADISCH S., SCHUMANN H., ERNST M., FÃIJLLEN G., TOMINSKI C.: Semi-automatic editing of graphs with customized layouts. *Comp. Graphics Forum 33*, 3 (2014), 381–390. doi:10.1111/cgf.12394. 12

[KEC06] KELLER R., ECKERT C. M., CLARKSON P. J.: Matrices or node-link diagrams: which visual representation is better for visualizing connectivity models? *Information Visualization 5* (2006), 62–76. 4

[KNT13] KOBOUROV S. G., NÖLLENBURG M., TEILLAUD M.: Drawing Graphs and Maps with Curves (Dagstuhl Seminar 13151). *Dagstuhl Reports 3*, 4 (2013), 34–68. doi:10.4230/DagRep.3.4.34. 5

[KPW13] KERREN A., PURCHASE H. C., WARD M. O.: *Multivariate network visualization: Dagstuhl Seminar 13201, May 12-17, 2013: revised discussions*. Springer, 2013. 12

[Kri04] KRINKE J.: Visualization of program dependence and slices. In *Proc. 20th IEEE Int'l Conf. on Soft. Maint.* (Sept 2004), pp. 168–177. doi:10.1109/ICSM.2004.1357801. 3

[LA04] LATTNER C., ADVE V.: LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Gen. and Optimization* (2004), CGO '04, pp. 75–. 5

[LPP*06] LEE B., PLAISANT C., PARR C. S., FEKETE J.-D., HENRY N.: Task taxonomy for graph visualization. In *Proc. 2006 AVI BELIV Workshop* (2006), BELIV '06, ACM, pp. 1–5. doi:10.1145/1168149.1168168. 4

[LTM17] LAM H., TORY M., MUNZNER T.: Bridging from goals to tasks with design study analysis reports. *IEEE Trans. on Vis. and Comp. Graphics* (2017). doi:10.1109/TVCG.2017.2744319. 3

[OSG*14] OLSCHANOWSKY C., STROUT M. M., GUZIK S., LOFFELD J., HITTINGER J.: A study on balancing parallelism, data locality, and recomputation in existing pde solvers. In *Proc. Supercomputing* (2014), SC '14, pp. 793–804. doi:10.1109/SC.2014.70. 5

[PGS*16] PARTL C., GRATZL S., STREIT M., WASSERMANN A. M., PFISTER H., SCHMALSTIEG D., LEX A.: Pathfinder: Visual analysis of paths in graphs. In *Proc. Eurographics / IEEE VGTC Conf. on Vis.* (2016), EuroVis '16, pp. 71–80. doi:10.1111/cgf.12883. 3

[RNL*13] ROBERTS M. J., NEWTON E. J., LAGATTOLLA F. D., HUGHES S., HASLER M. C.: Objective versus subjective measures of paris metro map usability: Investigating traditional octolinear versus all-curves schematics. *Int'l J. Human-Comp. Studies 71*, 3 (2013), 363 – 386. 5

[San95] SANDER G.: Graph layout through the vcg tool. *Proc. Graph Drawing, GD'94, LNCS 894* (1995), 194–205. 3, 5

[STT81] SUGIYAMA K., TAGAWA S., TODA M.: Methods for visual understanding of hierarchical system structures. *IEEE Trans. on Systems, Man, and Cybernetics 11*, 2 (1981), 109–125. 2, 3, 5

[TAS09] TOMINSKI C., ABELLO J., SCHUMANN H.: Technical section: Cgv-an interactive graph visualization system. *Comput. Graph. 33*, 6 (Dec. 2009), 660–678. doi:10.1016/j.cag.2009.06.002. 7

[TWS14] TOPRAK S., WICHMANN A., SCHUPP S.: Lightweight structured visualization of assembler control flow based on regular expressions. In *Proc. IEEE VISSOFT* (2014), pp. 97–106. 3

[WSW*18] WONGSUPHASAWAT K., SMILKOV D., WEXLER J., WILSON J., MANÉ D., FRITZ D., KRISHNAN D., VIÃL'GAS F. B., WATTENBERG M.: Visualizing dataflow graphs of deep learning models in tensorflow. *IEEE Trans. Visualization & Comp. Graphics* (2018). 3

[WWM08] WÜRTHINGER T., WIMMER C., MÖSSENBÖCK H.: Visualization of program dependence graphs. In *Proc. Joint Euro. Conf. on Theory and Prac. of Soft. / 17th Int'l Conf. on Compiler Construction* (2008), CC'08/ETAPS'08, pp. 193–196. 3