SANDIA REPORT

SAND2018-14238
Unlimited Release
Printed December 2018

Creating an Interprocedural Analyst-Oriented Data Flow Representation for Binary Analysts (CIAO)

Michelle Leger, Karin M. Butler, Denis Bueno, Matthew Crepeau, Christopher Cuellar, Alex Godwin, Michael J. Haas, Timothy Loffredo, Ravi Mangal, Laura E. Matzen, Vivian Nguyen, Alessandro Orso, Geoffrey Reedy, John T. Stasko, Mallory Stites, Julian Tuminaro, Andrew T. Wilson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories,
a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC,
a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's
National Nuclear Security Administration under contract DE-NA0003525.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865) 576-8401
Facsimile:	(865) 576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800) 553-6847
Facsimile:	(703) 605-6900
E-Mail:	orders@ntis.fedworld.gov
Online ordering:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2018-14238
Unlimited Release
Printed December 2018

Creating an Interprocedural Analyst-Oriented Data Flow Representation for Binary Analysts (CIAO)

Michelle Leger
Threat Intelligence Center
maleger@sandia.gov

Karin M. Butler
Quality Assurance Center
kbutle@sandia.gov

Denis Bueno
Matthew Crepeau
Christopher Cuellar
Michael J. Haas
Timothy Loffredo
Laura E. Matzen
Vivian Nguyen
Geoffrey Reedy
Mallory Stites
Julian Tuminaro
Andrew T. Wilson

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Alex Godwin
Ravi Mangal
Alessandro Orso
John T. Stasko

Georgia Institute of Technology
North Avenue NW
Atlanta, GA 30332

Abstract

National security missions require understanding third-party software binaries, a key element of which is reasoning about how data flows through a program. However, vulnerability analysts protecting software lack adequate tools for understanding data flow in binaries. To reduce the human time burden for these analysts, we used human factors methods in a rolling discovery process to derive user-centric visual representation requirements. We encountered three main challenges: analysis projects span weeks, analysis goals significantly affect approaches and required knowledge, and analyst tools, techniques, conventions, and prioritization are based on personal preference. To address these challenges, we initially focused our human factors methods on an attack surface characterization task. We generalized our results using a two-stage modified sorting task, creating requirements for a data flow visualization. We implemented these requirements partially in manual static visualizations, which we informally evaluated, and partially in automatically generated interactive visualizations, which have yet to be integrated into workflows for evaluation. Our observations and results indicate that 1) this data flow visualization has the potential to enable novel code navigation, information presentation, and information sharing, and 2) it is an excellent time to pursue research applying human factors methods to binary analysis workflows.

Acknowledgments

We would like to thank Danny Loffredo, Chris Leger, Todd Jones, Doug Ghormley, Tiemoko Ballo, Bryan Kennedy, Ben McBride, Kerstan Cole, Eric Moyer, Chris Wampler, Nasser Salim, Josh Maine, Adam Vail, and the many binary analysts who supported this work. Their regular interactions, thought-experiments, suggestions for references and approaches, and novel ideas have been invaluable. We could not have attempted this research without their amazing support.

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The format of this report is based on information found in [73].

Declaration

All studies within this research were performed with the approval of the Sandia National Laboratories Human Subjects Review Board.

Contents

Appendix

List of Figures

List of Tables

Preface

We analyze binaries to understand and mitigate potential vulnerabilities. We each have our own approaches, our pet techniques, our specialties in source code type and vulnerability type. Some of us are really good at using dynamic instrumentation; some of us prefer static analysis. Some of us know fuzzers like AFL [115], or symbolic execution engines like angr [94], or broad classes of techniques like abstract interpretation [107] or machine learning [39]. We all work together, all the time, because realistic software poses hard problems – often undecidable problems – and we need the skills of our peers to find solutions quickly. And we are always behind.

We have realized that, without some drastic improvement, we will continue to fall further and further behind. For example, take static binary analysis – our only fallback when we have no way to execute some code. For static binary analysis, we have a plethora of reverse engineering frameworks and program understanding tools at our fingertips [55][3][12][33][70]. We have a plethora of techniques and tools implementing them, and we have platforms that combine these techniques in incredibly clever ways [100][27][25][86]. We even have fantastic data flow analyses, usually designed to support compiler optimizations, that iteratively compute to a fixed-point to make high fidelity global models of code [65][45][59][99][103][57]. But when we sit down to an analysis, it still takes us weeks to months to years to come to incomplete conclusions about the binaries we need to analyze, and we are rarely able to use these excellent tools as much as we would like.

Again, take an example: static analysis of code that has a data flow vulnerability, an insidious flaw described in [24] and automatically exploited in [60]. A data flow vulnerability exhibits unexpected data relationships along expected control paths. It is surprisingly difficult to fully understand these vulnerabilities in code you did not write, i.e., to create accurate mental models of the code that help to understand and mitigate the vulnerabilities. The tools and techniques described above often fail to scale to real-world binaries, and we are left to understand the binaries manually. Our binary analysis tools have excellent visualizations for control flow; the control flow graph (CFG) [2] provides a beautiful mid-level abstraction between instructions and the call graph, and the primitives are easy to describe.[1] We use the CFGs all the time to derive information about binary code; nodes with lots of children or "wide" parts of the graph may indicate a dispatcher, a tall and narrow CFG may indicate lots of library calls, and a "c" or "z" shape may indicate a series of initialization code with error handling. However, these heuristics do not help in understanding a data flow vulnerability. How can we recognize and understand such vulnerabilities quickly?

More broadly, many of our vulnerability analysis tasks require deep understanding of data flow and data relationships. Our best data flow visualizations tend to do one of the following:

[1] Though CFG definitions can differ slightly between tools, conceptually each node represents a group of instructions such that all of the instructions are executed if one of them is, and each edge represents the relationship "may transfer control to".

- They display the data flow by overlaying data flow information onto a control flow or call graph [86], where control-based abstractions interfere with graph interpretation at anything other than a very course-grained level. Data flow is tied to instructions, which are difficult to separate at control abstractions above the instruction level.

- They offer understanding through a text-based visualization [56][114][113] where analysts still have to parse the code to discover relationships, or they display limited relationships such as taint [8] or pre-computed metrics [9] (note that more recent visualizations are similar).

- They display the data flow graphs produced for and by automated data flow analyses, on source code, to provide a one-shot global summary [103][112][31]. When binaries can be shoe-horned into the tools that produce these graphs (which often fails due to uncertainty introduced when semantically lifting the binary), and when human analysts try to view these graphs (which may fail, either because visualization tools are not usually tuned to handle the millions of nodes and edges or because layouts are not sufficient to convey the required information), the displays are overwhelming. We do not have ways to filter, organize, and abstract these graphs.

We use these visualizations (and related tools) to get answers to specific sub-problems about explicit relationships, and we stitch those answers together in an end-to-end analysis using additional knowledge about implicit relationships imposed through programming paradigms.

Intuitively we feel that something is wrong with the current state of the practice. In response, we continually identify problems, and we request and build automated tools to make our lives better. However, our intuition is failing us – with few exceptions, the tools we make do not seem to change things all that much. They are not widely adopted, they are difficult to use for non-developers, and they interfere with the limited time we have to get our analyses done.

Can we have a useful representation that helps analysts integrate knowledge from many places (like a symbolic execution engine summarizing path information, an abstract interpretation fixed-point engine providing global summary information, an analyst's own observations, and knowledge from that guy down the hall)? Can we effectively represent data flow to the human brain, allowing us to really exercise our strengths in pattern recognition and judgment? Can we augment human binary analysts with the right kind of information from automated analyses, helping us to answer important questions involving data flow and create mission solutions faster?

This is what we want to find out. This time, we are not going to try to start from the data flow information and abstractions that we already have. They might be great, but we do not know which ones are best. Instead, we are going to try to understand how the human analysts are thinking about data flow, and we are going to try to help them externalize their own mental models.[2]

[2] This report was written primarily by binary analysts and cognitive psychologists. We try to make the report accessible to everyone, especially as we have spent these past two years learning each other's language, but it is likely that in many places we assume an audience that is expert in both fields. Please accept our apologies and bear with us through this report.

Summary

National security missions require understanding third-party software binaries, a key element of which is reasoning about how data flows through a program. Our research goal in the CIAO LDRD project (creating an interactive, analyst-oriented data flow representation) was to significantly reduce the human time burden for binary software reverse engineering and vulnerability analysis by allowing humans to intuitively interact with data flow in static binaries. We used human factors methods to design visual representation requirements, resulting in two primary research and development accomplishments.

First, we designed a revolutionary user-centric representation of data flow. We captured our data flow representation in a spreadsheet detailing requirements for a representation and design choices for implementing those requirements in a visual graph. We produced a handful of example static visualizations by manually applying our requirements to specific DARPA Cyber Grand Challenge (CGC) binaries. Our representation, if implemented per our developed requirements, may significantly reduce the human time burden in binary vulnerability analysis by enabling novel code navigation, information presentation supporting rapid decision-making, and information sharing between human analysts and automated binary analyses. This representation still requires implementation to achieve impact, but it may support alternate and significantly more efficient workflows for binary vulnerability analysts.

In this report, we describe specific tests and human factors methods designed to extract these representation requirements from analysts and to begin to evaluate the utility of these requirements. In addition to standard human factors methods, we used a two-stage modified sorting task to generalize our task-specific requirements across diverse analysis tasks. To overcome 1) the extended duration of analysis projects (weeks) and 2) the diversity in data flow element names and categories introduced by varying analysis goals, we first asked analysts to identify categories describing variable names from their own completed projects, and then we asked different analysts to collaboratively analyze these categories. The modified sorting task used analyst descriptions of data flow from diverse, previously analyzed binaries to produce many of our visualization requirements.

Second, we describe potential workflow modifications and other insights about vulnerability analysis workflows. Our Georgia Tech Academic Alliance collaborators began to prove out some of these workflow modifications, creating a proof-of-concept automatic generator and a proof-of-concept visualization for interactive exploration, annotation, and update of such graphs. We describe both efforts briefly in this report.

More generally, we established a new knowledge base at Sandia in applying human factors methods to binary software reverse engineering. We are growing collaborations between human factors specialists and software reverse engineers. We are beginning to explore basic workflows,

we have identified human factors methods from other domains that may be applicable, and we have identified some ways in which current human factors methods may need to be modified to apply to the variety of goal-driven reverse engineering workflows.

In this report, we present our findings within the context of the project trajectory. Where we cannot include these findings within the general story line, appendices provide this documentation.

Chapter 1

Introduction: the Problem

Society increasingly relies on software that both interacts with security-critical data and communicates with external networks (e.g., in the military, in medicine, in education, and at home). Further, software complexity, size, variety, and modification rate continue to increase. Concern about how we will assure that software does not have vulnerabilities is growing [96].

Ideally, automated tools would assess and protect binary software statically, without executing the program. Static binary analysis avoids needing access to all the supporting systems required to run the binary, missing vulnerabilities introduced during the translation from source code to binary [36], and introducing threats from actually running the code. Unfortunately, automatic static binary analyses do not scale to real-world software [97].

Currently, experts assess and protect systems by performing static binary vulnerability analysis (VA) manually with assistance from automated tools [95] and then mitigating discovered vulnerabilities. These experts use extensive domain knowledge of binary code, operating systems, hardware platforms, programming languages, and vulnerabilities; they engage in reverse engineering (RE) [101] to understand binary programs [66][43], combining their extensive knowledge and that of their colleagues with automated tool results and line-by-line analysis. Binary vulnerability analysis is cognitively demanding, requires persistent attentional resources, and lacks prescribed approaches or tools. Binary code analyst support tools must be effectively integrated into their workflows to support their decision-making processes [6].

Finding and fixing vulnerabilities requires understanding how data, passing through program functions, influences other program data and program decisions. Unfortunately, data flow is difficult to understand, particularly when working from a binary. Programmers write source code, using comments and variable and function names to explain the purpose of parts of the code and to help model the data flow and control flow. When translating from source to binary code, compilers remove these comments, they may remove all names, and they change the code to make it faster or smaller or safer — and usually less understandable. Vulnerability analysts, modeling how the computer would run a binary, have to rediscover the data flow and control flow that are actually present in the binary.

Analysts find that the current set of tools for understanding data flow is inadequate. Data flow information is often projected onto a control flow visualization, which shows the order of instructions in a program. Analysts find such visualizations confusing, even for small functions, and data flow questions tend to span several functions together. Current tools that provide such visualizations do not support *interprocedural static* views that allow analysts to view data flow through several functions at once [55][86][83][117][52][12]. Although some source (not binary) analysis tools do provide interprocedural views unhampered by projection onto control flow representations, these views were designed for automated tools and tend to be too complex for human analysts [103][112][31].

Our goal in this research was to derive the requirements for an analyst-centric interprocedural data flow visualization to assist binary reverse engineers in identifying and mitigating vulnerabilities in code. Rather than trying to produce a new tool, or to integrate or abstract across the varied representations that current data flow analysis tools produce[1], we tried to infer the human analysts' mental models and create a way for them to express their mental models explicitly. Our requirements needed to inform a visualization of information dense collections[2]: supporting anthropological interpretation that is difficult for technology to do, guided by decisions that the information dense data will be used to make, and supporting the development of situational awareness *as* data is collated.

To derive our requirements, we used a rolling discovery process with experienced binary analysts. We used several standard cognitive task analysis methods and introduced a modified sorting task to derive our requirements. We evaluated our requirements using proof of principle and analytical principle testing. Our primary contributions include

- a taxonomy of required features to support vulnerability analyst understanding of data flow in static analysis of binary code (Section 3.2.3),
- a description of a modified sorting task, a human factors method to achieve consensus about mental models used across diverse tasks (Section 2.5),
- and a preliminary implementation of a visualization of these requirements to be integrated into vulnerability analysis workflows.

Additionally, we provide an initial evaluation of a subset of our developed requirements through proof of concept and analytic evaluation [63]. Unfortunately, we were unable to implement a fully-featured interactive visualization during this research, and our evaluation is necessarily lacking. We describe our informal evaluation in Section 4, providing evidence that further evaluation, at least, is warranted.

Note, our ultimate goal is not to remove humans from the loop; rather, we want to make easier for analysts to spend their time on the creative aspects of analysis. We hope that this effort will make binary analysis more accessible to novice analysts as well.

[1] This is the approach that we have taken, unsuccessfully, in the past.

[2] This is as opposed to re-plotting, which presents existing data, untransformed, in a different form to aid decision making. A CFG is an example of re-plotting.

Chapter 2

Approach: Using Human Factors Methods to Derive Visualization Requirements

2.1 Applying Human Factors Methods

To begin to understand the different ways that vulnerability analyses are performed, and to derive some initial requirements for a data flow visualization, we used standard cognitive task analysis methods, including semi-structured interviews, Applied Cognitive Task Analysis, and cognitive walkthroughs. These activities identified both *interactive* requirements, supporting an analyst in building up and capturing knowledge about the binary under analysis, and *static* requirements, presenting facts or conclusions about data flow elements and their relationships. These activities showed that vulnerability analysts use data flow to identify 1) where specific data influences the code, 2) how data is parsed and manipulated through the code, 3) how the code controls and checks data to prevent problematic effects, and 4) unintended or obfuscated data flow paths.

To derive requirements that would support a visualization of data flow across these tasks, we then focused on gathering information about analyst mental models from artifacts of their own projects, which spanned these data flow tasks. We developed a two-stage modified sorting task to help us gather this information from across the analysts' own diverse tasks *without requiring the analysts to evaluate the same binaries or types of binaries*. [1]

This section describes our human factors methods in more detail, resulting in our list of requirements for a data flow visualization for binary reverse engineers. We try to cleverly iterate on design and evaluation to overcome major challenges with this approach: that binary analyst workflows, tools, and approaches differ by analysis goal and analyst, that data flow analysis spans several days to weeks and is intermingled with other binary analysis tasks, that understanding the most important data flow for vulnerability analysis requires integrating information gathered across long intervals of time, that these human factors techniques usually are not applied in exactly this way, and that, locally at Sandia and globally, we have a very small pool of expert individuals to test.

[1] We believe that our two-stage modified sorting task may help to achieve consensus about mental models across diverse tasks in domains that deal with complex, variable targets, such as those often encountered in discovery-based analysis. However, we have not been able to evaluate this.

You have been identified as someone with expertise in extracting information from binary code about how data values flow through the code. Our team is working on an LDRD to develop a representation and visualization tool to aid in binary code analysis. Your participation in this research is completely voluntary. Would you be willing to spend two hours providing some insight into what you do? You will be provided with a project and task for your time.

Figure 2.1. Recruitment message for identified binary analysis domain subject matter experts (SMEs).

2.1.1 Overview of Participants

The knowledge in this SAND report is built on results gleaned from 1) experts that volunteered to participate in our interviews, cognitive walkthroughs, and sorting task, and 2) the experience of our team members and other expert and novice colleagues.

Our research participants are volunteers who are adult colleagues from the Sandia National Labs Threat Intelligence Center. Human Subjects Research Protection protocols, approved by the Sandia National Laboratories Human Subjects Review Board, were used in collecting these data.

Because our goal was to determine effective representations for solving data flow problems, the research team identified individuals with considerable experience in this domain – a small pool of selected participants. These potential participants were contacted face-to-face or through email with a brief description of the purpose and requirements of participation (see recruitment message in Figure 2.1). Participants were not required to participate as a condition of their employment or work within their organization, and decisions to participate did not affect evaluations.

These *experts* are colleagues or team members who have 5 to 20 years of experience analyzing binaries and source code for national security vulnerability assessments or malware characterization. Many of these experts analyzed binaries for optimization purposes for 5 to 10 years before their security-focused analyses.

In addition to these formal research participants, our research was informed by expert team members, expert non-team member analysts, and novice analysts. We relied on knowledge from our 6 team member analysts who have worked in binary and source analysis for between 6 and 20 years, specifically performing reverse engineering and vulnerability analysis of a wide variety of systems using many techniques. We also relied on feedback from expert non-team member analysts (i.e., individuals not invited to project meetings) specializing in different analysis techniques, code representations, or mission questions. Finally, we incorporated feedback from 8 *novices*, student interns (advanced undergraduate or masters students) who have analyzed binaries for security purposes for less than a year and who have not often analyzed binaries for other purposes.

Note that we do make claims throughout this report about how analysts work in this field; we base these claims on our own experience and on that of the other 20 expert analysts who engaged with the team throughout this research.

2.1.2 Selected Data Flow Use Case

Standard cognitive task analysis methods are most effective when applied to a well-scoped, representative use case. See Appendix A for a discussion of our use case selection, including examples of goal- and relationship-oriented data flow tasks in binary reverse engineering, and Appendix H for considerations when designing tests and evaluations of software tools to support binary auditing.

To focus our cognitive task analysis methods, our initial interviews and cognitive walkthroughs focused on the attack surface characterization task [74], a task that is both representative of many analyst considerations when evaluating data flow and amenable to a two-hour cognitive walkthrough. For these studies, we defined **attack surface characterization** as *determining which aspects of program input reach locations or control actions that are security sensitive*, including determining how well the binary protects those locations. Essentially, we asked analysts to assume that they were ranking portions of the binary for further vulnerability analysis and mitigation development given limited resources. For our cognitive walkthroughs, analysts were asked to assume that all input from the command line or an input-specified file is attacker-controlled, and that all non-attacker-controlled data is security sensitive.

We used results from our studies on attack surface characterization to define a preliminary representation. We then used our two-stage modified sorting task, performed with artifacts created across a variety of analysis goals and binaries, to generalize our results from attack surface characterization. However, we did not attempt to evaluate our resulting requirements generally; further work will need to explore other use cases besides attack surface characterization to determine the impact of our initial scoping decisions.

2.1.3 Preliminary Requirements from Cognitive Task Analysis

As mentioned, our rolling discovery process used several cognitive task analysis (CTA) methods to determine what support experienced binary analysts need when performing binary vulnerability analysis. The CTA included semi-structured interviews, applied cognitive task analysis (ACTA), and cognitive walkthroughs of a data flow RE task. We used results from these methods to generate our preliminary list of requirements for a data flow visualization. This research was reviewed and approved by Sandia National Laboratories Human Subjects Review Board.

To begin to identify tasks, subtasks, important cognitive processes, and data flow elements, we conducted two rounds of semi-structured interviews with separate sets of experienced binary code analysts in individual sessions.

In the first round of semi-structured interviews, three experienced analysts identified the process steps, tools, and some of the cognitive challenges associated with binary reverse engineering in general. Subsequent interviews and cognitive walkthroughs focused on the attack surface characterization task [74], a task that is both representative of many of the considerations when evaluating data flow and amenable to a two-hour cognitive walkthrough.

In the second round of semi-structured interviews, three different experienced analysts answered questions from an applied cognitive task analysis knowledge audit [76]. The knowledge audit revealed the most important goals of attack surface characterization, cues in the binary code that indicate possible vulnerability or that contribute to program understanding, judgments being made during analysis, and tools used to support the work.

Building on results from these interviews, we designed a cognitive walkthrough task to capture information, in situ, about the cognitions, decisions and processes used by analysts during attack surface characterization.

We compiled the results of the interviews and cognitive walkthrough into a description of the process steps in attack surface characterization and a preliminary list of static data flow elements and interaction requirements for our data flow visualization.

2.1.4 Requirements Developed from Modified Sorting Task

Next, we used a modified sorting task of analyst artifacts to identify categories of data flow elements. We used results from this task to more fully develop requirements for a data flow visualization. This research was also reviewed and approved by Sandia National Laboratories Human Subjects Review Board.

In binary analysis projects, vulnerability analysts record the essential elements they use for understanding data flow; they assign meaningful names to variables and functions to record this information. We hypothesized that binary analysts might reveal general purpose data flow elements through a sorting task [92] over their own meaningful data variable and value names.

A standard sorting task provides insight into the mental models of users, and it is generally used to inform the grouping and naming of those categories in an interface [92]. In a typical sorting task, the elements (e.g., words or functions) to be sorted are known before the task is conducted. Each participant sorts the same elements into groups; consensus grouping, if revealed, reflects similarities in how the participants think about the given elements.

In our case, however, we were attempting to derive analysts' mental models from elements that were not known beforehand and that varied by analyst. We thus needed to overcome two main challenges: analysts name both data flow elements and categories of elements according to analysis goals and personal preference, making it difficult to find commonalities; and analysis projects span weeks, making it infeasible for analysts to independently analyze the same binaries.

To address these challenges, we created a two-stage modified sorting task. Our approach used analyst descriptions of data flow taken from diverse, previously analyzed binaries for the sorting task followed by a second stage of evaluation to hone the list of data flow elements. We used the results of the second stage to derive our list of initial requirements for a data flow visualization.

Introduction Today I am going to ask you to describe, at a high level of abstraction, the steps you take when you are analyzing a binary to determine the attack surface. You should not have to describe the identifying details of any attack surface characterization projects that you are working on.

1) Does your current work involve analyzing data flow for VA and RE?
2) How long have you been doing work that required this type of analysis?
3) When you think about doing data flow analysis for attack surface in your work, would you say that the problems that you work on are the same (similar code from similar sources, compilers, parent languages)? Or are they different?
4) For attack surface data flow analysis, what tools are useful?

ATTACK SURFACE Think about when you do VA and RE and you are trying to understand the attack surface of the problem.

5) a) What are the 3-6 steps that you use when working with the code to discover the attack surface?
   b) What determines the order of these subtasks?
   c) For each of the subtasks, what are the cognitive skills that you are using? For example, judgments, assessments, problem-solving, decision-making?
6) Do novices approach these problems differently than an expert like yourself?
7) For data flow analysis, what sorts of kludges and work-arounds do you use or has your team created?
8) How might new tools or decision aids help make the work both more effective and easier by eliminating needless capability gaps?

Figure 2.2. Interview protocol for our first round of semi-structured interviews, which identified the process steps, tools, and some of the cognitive challenges associated with binary reverse engineering and data flow understanding.

2.2 Cognitive Task Analysis

The first round of semi-structured interviews, cognitive task analysis interviews with three experienced analysts, identified the process steps, tools, and some of the cognitive challenges associated with binary reverse engineering in general and data flow understanding in specific. We wanted to answer the following questions: *What process describes an analyst's workflow when performing attack surface characterization? What cognitive processes are analysts using (e.g., judgments, assessments, problem solving, decision making), and where do they need help?*

2.2.1 Experimental Setup

In this first round, we interviewed three expert analysts experienced in binary reverse engineering and/or vulnerability analysis for mission problems. Figure 2.2 shows the interview protocol for these semi-structured interviews.

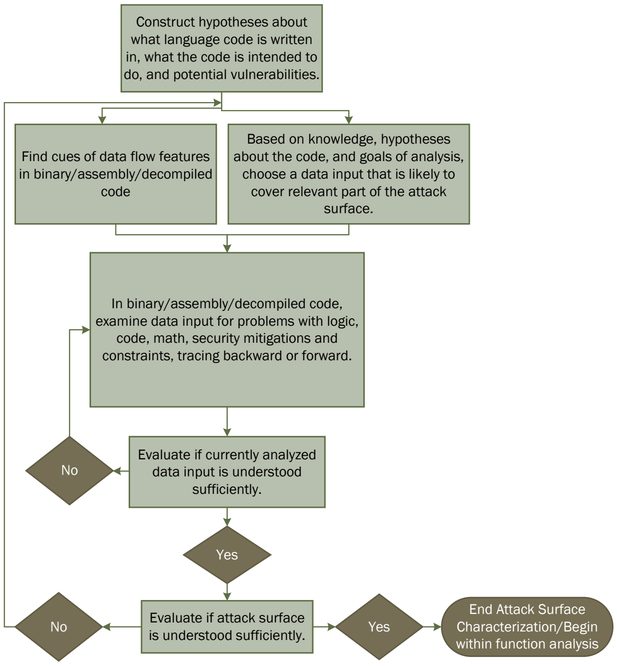

Construct hypotheses about what language code is written in, what the code is intended to do, and potential vulnerabilities.
Find cues of data flow features in binary/assembly/decompiled code
Based on knowledge, hypotheses about the code, and goals of analysis, choose a data input that is likely to cover relevant part of the attack surface.
In binary/assembly/decompiled code, examine data input for problems with logic, code, math, security mitigations and constraints, tracing backward or forward.
Evaluate if currently analyzed data input is understood sufficiently.
No
Yes
Evaluate if attack surface is understood sufficiently.
No
Yes
End Attack Surface Characterization/Begin within function analysis

Figure 2.3. Data flow analysis diagram for attack surface characterization process developed from semi-structured interviews. The bulk of the work is in the center box, tracing a data input.

2.2.2 Results

We compiled the results of these interviews into a description of the process steps in attack surface characterization, Figure 2.2.2. Analysts spend most of their time in the central box, examining data flows for potential problems. We focused on decomposing this step in later interviews.

Binary analysts use the following cognitive processes when characterizing an attack surface:

Updating mental models, working memory, episodic memory Correlating dynamic instrumentation results with the static binary code. Because we are focused on static analysis, this is less relevant for our goals.

Working memory, episodic memory, and maintaining a mental model Keeping track of what assumptions were made and where they were made earlier in the analysis.

Inductive reasoning Selecting correct knowledge to help understand the system.

Deductive reasoning Judging and refining mental models and hypotheses about the program.

Pattern recognition Identifying libraries, design patterns, data flow types, vulnerability types, mitigation types, and so on.

Planning Selecting next goals. This selection often depends on analyst preferences and experience, including knowledge of programming error patterns, familiarity with code or program type, interest in the technology, accessibility of exterior surface, and specific assessment goals and assumptions.

2.2.3 Discussion

Throughout this process, analysts create, test, refine and discard hypotheses about potential vulnerabilities, programming language, program goals, and likely sources of errors. We will need to provide categories of data manipulation that can be used as descriptive labels associated with data elements, supporting easy annotation. To help analysts answer mission questions faster, we will need to help analysts to express and manipulate their hypotheses significantly faster.

2.3 Applied Cognitive Task Analysis

In the second round of semi-structured interviews, three experienced analysts (one who had been interviewed previously) answered questions from an Applied Cognitive Task Analysis knowledge audit [76]. Our goal was to expand the central box in our initial attack surface process chart, understanding in more detail *how* analysts look for problems when tracing data flow through functions (i.e., "pulling on a thread" or focusing on a specific data flow). These 1.5-hour interviews focused on the knowledge and tools used when analyzing a binary to identify the attack surface.

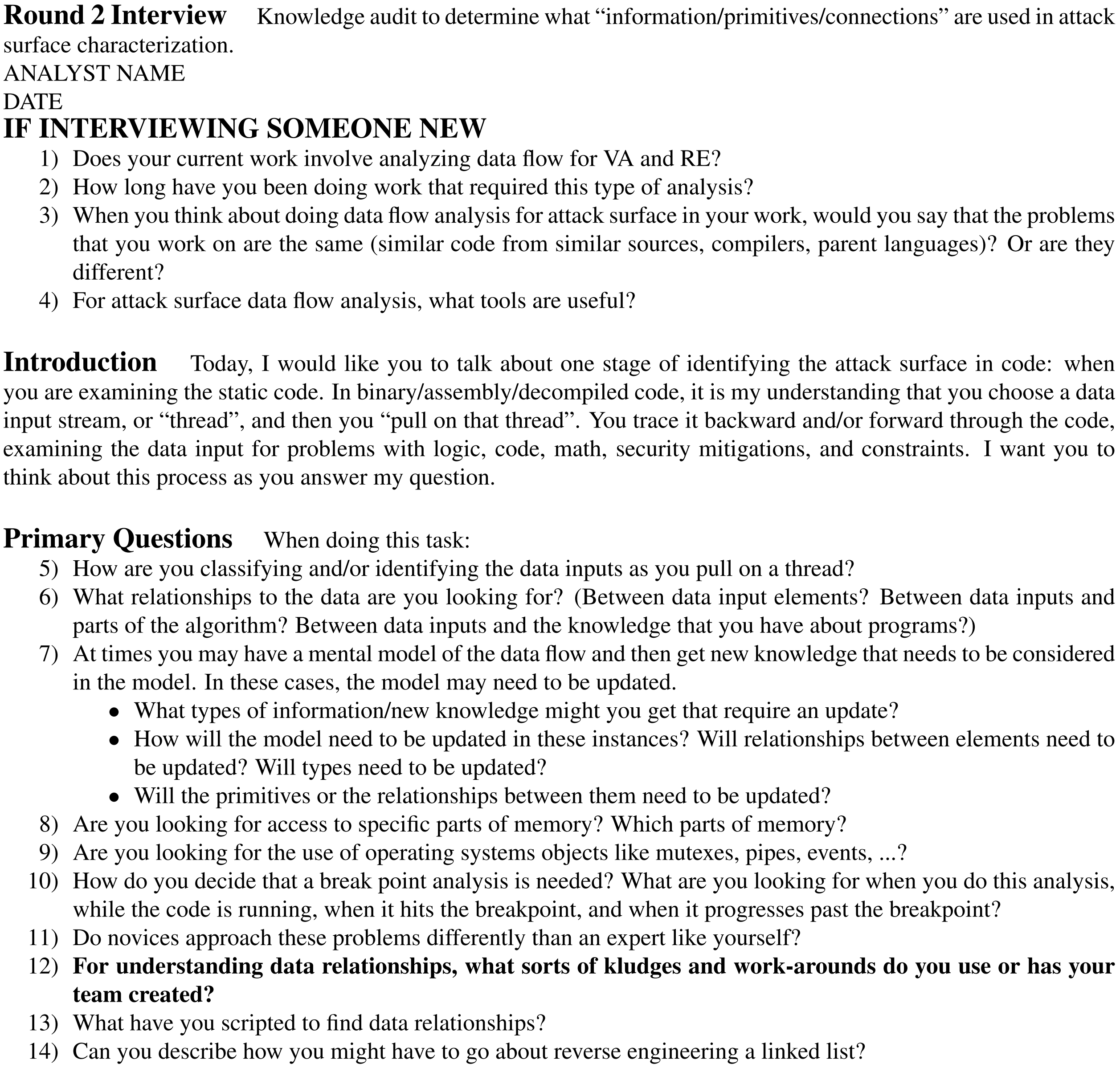
We used our **knowledge audit** to catalogue the explicit and implicit knowledge that is critical for performing attack surface characterization. We asked analysts to think about this process as we asked a set of questions to identify analyst abstractions and idioms. These questions focused on when, why, and how analysts search for, identify, classify, and abstract data input elements and relationships (Figure 2.4). However, sometimes people find it difficult to answer specific questions when thinking abstractly. To help in these cases, we used backup **knowledge elicitation** questions to provide the analyst with a foundation by having them actually retrieve episodic memories. These questions focused on places where expertise is demonstrated, e.g., when answers "pop out" or ways in which analysts "work smart".

2.3.1 Experimental Setup

In these interviews, we asked analysts to focus on the central stage of the process identified previously, "pulling a thread" or tracing a specific data flow. Figures 2.4 and 2.5 show the interview protocol for these semi-structured interviews.

2.3.2 Results

We present here our raw data from these interviews, removing binary- and analysis-specific details. We focus on analyst aids and heuristics, but we present the rest of the raw data as well in Figure 2.6.



Round 2 Interview Knowledge audit to determine what "information/primitives/connections" are used in attack surface characterization.
ANALYST NAME
DATE

IF INTERVIEWING SOMEONE NEW

1) Does your current work involve analyzing data flow for VA and RE?
2) How long have you been doing work that required this type of analysis?
3) When you think about doing data flow analysis for attack surface in your work, would you say that the problems that you work on are the same (similar code from similar sources, compilers, parent languages)? Or are they different?
4) For attack surface data flow analysis, what tools are useful?

Introduction Today, I would like you to talk about one stage of identifying the attack surface in code: when you are examining the static code. In binary/assembly/decompiled code, it is my understanding that you choose a data input stream, or "thread", and then you "pull on that thread". You trace it backward and/or forward through the code, examining the data input for problems with logic, code, math, security mitigations, and constraints. I want you to think about this process as you answer my question.

Primary Questions When doing this task:

5) How are you classifying and/or identifying the data inputs as you pull on a thread?
6) What relationships to the data are you looking for? (Between data input elements? Between data inputs and parts of the algorithm? Between data inputs and the knowledge that you have about programs?)
7) At times you may have a mental model of the data flow and then get new knowledge that needs to be considered in the model. In these cases, the model may need to be updated.
   - What types of information/new knowledge might you get that require an update?
   - How will the model need to be updated in these instances? Will relationships between elements need to be updated? Will types need to be updated?
   - Will the primitives or the relationships between them need to be updated?
8) Are you looking for access to specific parts of memory? Which parts of memory?
9) Are you looking for the use of operating systems objects like mutexes, pipes, events, ...?
10) How do you decide that a break point analysis is needed? What are you looking for when you do this analysis, while the code is running, when it hits the breakpoint, and when it progresses past the breakpoint?
11) Do novices approach these problems differently than an expert like yourself?
12) **For understanding data relationships, what sorts of kludges and work-arounds do you use or has your team created?**
13) What have you scripted to find data relationships?
14) Can you describe how you might have to go about reverse engineering a linked list?

Figure 2.4. Primary questions (knowledge audit) from interview protocol for our second round of semi-structured interviews, an Applied Cognitive Task Analysis knowledge audit.

Questions to Inventory Task-specific Expertise (BACKUP Questions)

1) Was there a time when you opened up an analysis project and knew exactly where a vulnerability was, or how to find or fix it?
2) Have you had experiences while doing data flow analysis where part of a situation just “popped” out at you – where you noticed things going on that others didn’t catch? What is an example?
3) When you analyze code to characterize the attack surface, are there ways of working smart or accomplishing more with less that you have found especially useful?
4) Can you think of a time when you were working with code to determine the attack surface and then realized that you would need to change the way you were performing in order to get the job done?
5) Can you describe an instance when you spotted a deviation from the normal process of data flow analysis or knew something was amiss?
6) Have there been times when the software pointed in one direction, but your own judgment told you to do something else? Or when you had to rely on experience to avoid being led astray by the software?

Figure 2.5. Backup questions (knowledge elicitation) from interview protocol for our second round of semi-structured interviews, an Applied Cognitive Task Analysis knowledge audit.

Hacks, Scripts, and Tools

We list the hacks, scripts, and tools that analysts described during the interviews.

- Functions and variables are named to indicate what they do and the knowledge available about them. For example,
  - Analysts might identify several memory locations (including registers) as the same, or identifying the same location as holding several distinct variables at different times.
  - Analysts use their own conventions to identify uncertainty about information conveyed in such names, e.g., prepending a “?” or “_”.
- A script pulls out debugging text strings to help identify what the function does.
- Identifying type information for a structure requires a lot of work using current tools, but this information can be automatically propagated.
- Often analysts use a wiki for documentation. Two interviewees described the same scenario for the wiki use. When investigating a single data input in a single function, the data input is used in many different ways (e.g., a case statement) and passed to different functions. The wiki is used to document each “fanning out” point and to keep track of whether each path or branch has been investigated. Unfortunately, the wiki is cumbersome and time-consuming; it could be more helpful than it is. It is usually easier to draw pictures in a notebook than in the wiki.
- A database of known data structures, objects, or library calls for an executable type can be automatically propagated through the program.
- Careful notes about choice points and decisions made allow efficient back-pedaling if necessary.

- Memory visualizations help analysts understand particular vulnerability types.
- Complicated loading processes, such as import resolution, need to be modeled correctly.

Critical Knowledge about Functions

These are indicators that further analysis of a function may be warranted. They are heuristics that the analysts use to help identify and rank potentially problematic functions.

- The function is either large or complex, or has simple errors often together [67], or is difficult to get right in the first place.
- The function lacks security or integrity checks, e.g., that data is of the expected size.
- The function performs lots of logic around the data (as opposed to lots of math).
- The function interacts with lots of data inputs (as opposed to just one).

2.3.3 Discussion

The knowledge audit revealed the most important goals of attack surface characterization, cues in the binary code that indicate possible vulnerability or that contribute to program understanding, judgments being made during analysis, and tools used to support the work.

2.4 Cognitive Walkthroughs

Building on results from these interviews, we designed a cognitive walkthrough task to capture information, in situ, about the cognitions, decisions and processes used by analysts during attack surface characterization. We observed and questioned analysts performing a crafted attack surface characterization task, focusing our data collection on the cognitions and processes used for attack surface characterization. That is, we used this run-time assessment to observe how explicit and implicit knowledge are used during this task; experts who have automated parts of their workflows tend to gloss over certain parts of their process when they are not actually performing a concrete task. In our results below, we list the cognitive challenges that were noted by analysts and observed during the cognitive walkthrough; we discuss these challenges later.

2.4.1 Experimental Setup

We selected the UNIX file utility version 5.10 [47][46] for analysis, choosing from the AFL fuzzer bug-o-rama trophy case [115].[2] We chose file version 5.10 because 1) the core processing library

[2] Selecting a vulnerability found by AFL gives us the opportunity to control further testing by, e.g., providing an initial problematic input to guide the analyst. Further, programs listed in the AFL bug-o-rama trophy case have some

- How are data inputs classified?
  - Structure specifications (semantic meaning, length, type, allocation location), though these are often incomplete, especially for large structures
  - Structure, function, and variable names
  - Size, which could be specified in variable name if uncertain or as type information when certain
  - Definition of large data structures may include only a few knowns
- What relationships to the data input elements are you looking for?
  - Logical relationships, e.g., “If this byte is this, then that byte is that or could prevent this other thing from happening.” This type of knowledge is hard to propagate. Examples include checking the size or sign of a field.
  - Loop structures repeated through code. When they look different in one implementation, this signals a possible vulnerability.
  - Relationships to privilege mode (e.g., user, kernel).
  - Is this data element checked/sanitized?
  - Data parsing, especially fixed vs. variable length (often difficult to get right)
  - How data are terminated, e.g., appropriately null-terminated
- What relationships to the data input and algorithms are you looking for?
  - Complicated code
  - Paths that are not as frequently exercised (more likely problematic)
  - Can a particular data element at a certain place in the code be passed from the periphery by an attacker?
  - What security mitigations are in place?
  - Code practices that are susceptible to errors, e.g., memcpy with math
- What relationships to the data input and the knowledge that you have are you looking for?
  - Debug text strings
  - “Creative” use of relationships
  - Linked lists can look like pointer manipulations when you focus on a single element
- During analysis, what happens to cause you to change/update your mental model of what is happening in the program?
  - Updating is required when memory space is reused by the compiler, especially when the type of the data element changes.
  - Updating is required when the number of elements in a structure is unknown; may believe there is one when there are many.
  - A model may change when an element in a structure determines the type of the next element.
  - Memory size and how memory is used affect mental models.
- How is breakpoint analysis used?
  - Might assign breakpoints to many functions to see which ones runs. Used vs. not used helps with function labeling and informs choice of focus: unused functions may not be as important.
- What things do novices have difficulty with?
  - Standard library function names may mislead, e.g. Windows’ FileCreate is often used to open a file.
  - Many kernel drivers do the same thing, but this needs to be learned.
  - Abstraction from math to code is difficult.

Figure 2.6. Other raw data (besides hacks/scripts/tools and function prioritization heuristics) from our second round of semi-structured interviews, the Applied Cognitive Task Analysis.

libmagic is vulnerable to CVE-2012-1571 [77][3]; 2) many functions in the library are involved in parsing input data from multiple sources; 3) a successful analysis requires understanding interprocedural data flow; 4) we had access to source code for both the vulnerable version 5.10 and the fixed version 5.11;[4] and 5) file is one of the smallest UNIX utility binaries listed, making it more likely that a meaningful analysis could be completed in less than two hours.

Three experienced binary analysts completed the attack surface characterization task with the file binary in their preferred binary analysis environment. The binary was compiled on a machine running Ubuntu 16.04 with llvm, creating a 32-bit binary with symbols. To focus our data collection on the cognitions and processes used in understanding data flow, we asked analysts to begin analysis at the file_buffer function in libmagic, treating the array argument and length as attacker-controlled, i.e., as the "inputs" for the exercise. We did not require analysts to discover the vulnerability; rather, we asked analysts to produce, as if for future analysis, 1) a ranked list of (internal) functions or program points where the inputs are processed and may affect the security of the system, including specific concerns at each point, and 2) any comments, notes, or diagrams that might support a formal report for a full vulnerability analysis. We asked analysts to focus on depth over breadth (i.e., following data flow) and to think aloud while performing analysis. Our human factors specialist took notes about task performance and asked for additional details to understand the thought process of the analyst, including asking for reasoning behind judgments and decisions, and asking for clarification about sources of frustration. We provide our instructions to analysts and questions for analysts in Figures 2.7 and 2.8.

Walkthroughs lasted two hours including the time to set up the analysis environment. Analysts created the list of functions and concerns, but they produced few comments and no diagrams or additional notes. Although analysts often use two to four screens, we captured only the primary screen of each analyst. These artifacts were not analyzed separately.

2.4.2 Results

We compiled the results of the interviews and walkthroughs into a more detailed description of the process steps in attack surface characterization, a list of pain points and a cognitive decision matrix, and a preliminary list of data flow elements and interaction requirements for our data flow visualization.

We capture the process steps of analysts performing attack surface characterization on file in Table 2.1. We describe pain points observed in Figure 2.9. We present decision and cognitive requirements from this activity, including some descriptions of difficulties and potential solutions, in Table 2.2.

claim to providing "real" programs for analysis rather than small designed programs provided by, e.g., the Cyber Grand Challenge challenge binaries.

[3] This known CVE in the binary could allow us to perform cognitive walkthroughs of other binary analysis tasks, e.g., determining the risk of or mitigating a known vulnerability.

[4] Having source for both versions allowed us to control the binaries analyzed, e.g., whether we provided symbols or reduced optimizations.

Today I am going to ask you to describe, at a high level of abstraction, the steps you take when you are analyzing a binary to characterize the attack surface and related data flow. This exercise is not intended to be a full vulnerability analysis. I am asking you to focus on characterizing an attack surface so that I can observe you doing a part of the work that you typically do and can understand what types of cues and relationships you use in that part of your work. I will not be evaluating your work, nor should you have to describe details about any similar projects that you are working on.

To guide this description, I will provide you with a binary to analyze. If you have analyzed this binary in the past, please let me know and I will provide you with another. I will ask you to describe aloud how you are assessing the binary, why you take certain actions in assessing the binary, what cues or information you use that lead you to take certain actions, and how you are categorizing different parts of the binary. I may interrupt with questions to help you explain and think aloud. If such interruptions drastically disrupt your work, please let me know and we can arrange for you to complete the analysis exercise first and then talk through your analysis afterward.

By "characterizing an attack surface", I mean determining what aspects of inputs reach locations that are security sensitive and how well the binary protects those locations. For this binary, please:

- Begin at the identified function/interface file_buffer; arguments to this function are the "inputs" for this exercise, and one input is named for you.
  - You may explore anywhere in the binary to gain context, but your goal is to characterize the binary as if this function were an external interface.
- Produce a ranked list of (internal) functions or program points where inputs are processed and may affect the security of the system.
- Characterize these functions or program points – and the paths that reach them – with respect to the inputs and any other data sources.
- Focus on depth over breadth in your characterization: follow data down into internal functions and chase the path(s) of interest.
- Keep notes, comments, and diagrams as though you were going to complete a full vulnerability analysis, including a formal report characterizing the attack surface (you will not actually create this report).
  - Please take any written notes in our provided notebook.
  - Please leave us with a copy of any electronic artifacts created or modified during this activity.

Figure 2.7. Instructions provided to analysts for attack surface characterization cognitive walkthrough.

Background Questions Before beginning the exercise, we ask each analyst the following questions:

1) For how long have you been doing work that required analysis of binary code?
2) For how long have you been doing work that required analysis of attack surface of a binary?
3) For how long have you been doing work that required analysis of attack surface of source code?

Questions to Ask during Analysis We interrupt each analyst as needed with the following questions:

1) What are you trying to do or learn right now?
2) What did you need to do to learn about that code?
3) What are you thinking about right now?
4) In the section of code that you are working on right now, what are you thinking about?
5) Why did you jump to this section of code?
6) Why did you decide to name this?
7) What does the name mean?
8) Why did you decide you should make a comment here?
9) What does the comment mean?
10) Where are you looking on the screen? Why?
11) In the section of code that you are working on right now, what information are you thinking about?
12) Does the data that you are currently analyzing relate to other analysis that you have already done? How?

Figure 2.8. Questions we ask analysts before and during our attack surface characterization cognitive walkthrough.

1) Must find a reference to an argument within the assembly before it can be selected. The string cannot be searched for.
2) Even when a reference is highlighted throughout a function, must scroll through the entire function to find the yellow highlighting, and it is hard to know for sure you have seen all of them.
3) Tools are designed to be used in a top-down way. Decompiler propagates some specifications down into code, but does not propagate things, like structures, up to earlier calls. This leads to a problem when a structure is identified deep in the code one day, and the next person to work on the code does not know or forgets and starts in a different part of the code where structure is used again.
4) Fighting with the decompiler: a) stack buffer that the decompiler did not know about and was hard to follow; b) in a data structure editor, different fields like size are tweaked differently; c) structures within an array; complex types.
5) **Need better ways of keeping track of how the analysis has proceeded and what has been learned.** Two analysts explicitly noted memory failures. The failures were of two types: episodic and prospective. Episodic memory failures were failures to remember that a function had been previously evaluated. Prospective memory failures were failures of remembering that only attack surface was being characterized. Analysts wanted to go further.
6) Need better way of transmitting learning to other analysts on team.

Figure 2.9. Pain points observed during attack surface characterization cognitive walkthroughs.

Table 2.1. Detailed process steps identified in cognitive walkthroughs of attack surface characterization data flow analysis task. This task asked analysts to focus on "pulling on a thread" or tracing data flows.

Process Steps	Questions Analyst is Asking about Code	Observations of Interest	Notes	Cognitive Processing Challenges
1. Constrain the domain knowledge of consideration and activate relevant knowledge	What platforms was the code written for? What does this code do? Will I look, primarily, at disassembled or decompiled code? What functions of code does the customer want assessed?	Expectations about what the code does will activate knowledge of a subset of common vulnerability types. Knowledge of platform may also constrain common vulnerability type considerations.	This code was identified as parsing code which constrained where analysts looked for vulnerabilities. How many different "types" are there? For example, one analyst suggested that their attack surface process is different for web server hosted programs and for browser-based programs.	Hypothesis Generation and Knowledge Retrieval
2. Plan the scope of the work	How frequently and where in memory is input data used? Does input data get passed with function calls? Does input data get written to other memory locations? What areas of code or data flows does the customer want assessed?	1) Highlighting strings associated with data locations is used as a way of helping to move attention to only relevant info. 2) Frequency of highlighting is a quick qualitative assessment of complexity of characterizing attack surface. 3) Co-occurrence of highlighted area with assessment of whether data is passed to another memory location (read-write) were noted. 4) Co-occurrence of highlighting with function calls were noted. 5) Co-occurrence of highlighting with memory location offset specifications were noted.	(1) This interaction could be made more efficient by a better method of finding all highlighted locations and their co-occurrence. For example, in MS Word when you search on a word or phrase, the navigation pane provides you list of clickable matches along with some context around them.	Attentional Control to Search for Relevant Parts of Code
3. Assign priority to different paths of investigation	What function or pathway will be the most likely candidate, while still being accessible or manageable? What function or pathway will provide the most program knowledge that will generalize to the remainder of the project?	In the absence of a preview of a function, this judgment is based on 1) the names of the function or text strings related to the calling function that indicate possibility of complexity, spawning a process, known vulnerabilities; and/or 2) size in number of bytes of function that is called; and/or 3) how the returned values are used.	For example, cue words: Magic, File type indicators (ASC, ELF, CDF). ASC not expected to require a lot of evaluation, most did not know what a CDF was, and once the name of the CDF was found in text strings it was expected to require a lot of parsing.	Episodic Memory for Analysis Path Choices
4. Within a function or along a pathway that uses input data, find potential problems	Does this function possibly do anything problematic with the data I am interested in?	Cues: 1) function names known to be a problematic, e.g., file_print_f; 2) function names indicating spawning another process, e.g., function names with "exec" in them; 3) functions with a lot of arguments; 4) functions that have CFG that are "tall and narrow" suggesting a wide variety of calls to libraries; 5) memory calls and comparisons like mget, memcpy, malloc, free; 6) input/output flushing; 7) forking and creating pipes "which are complicated".	1) Variation in how completely this task was completed within a function. Some found a few possible problems, some wanted to identify all at this level and then investigate deeper in. 2) Some functions (e.g., print_f) known to be a problem.	**Pattern Recognition**; Knowledge Retrieval
		Used pattern recognition of common programming idioms and other features to exclude some blocks of the function from consideration based on familiarity/boilerplate material.		
5. More detailed evaluation of functions and sub-functions	How likely are the identified concerns to lead to problems?	1) How much of the memory that has input data is being used, e.g., size of offsets; 2) How many different locations within memory of interest are being used, e.g., variation in the offsets used to access the memory; 3) memory operations, e.g., malloc evaluation; 4) exec; 5) call to function that called this one (recursion) 6) checking of bytes at offset within data memory location of interest.	Analysts indicated that with an actual project they would spend more time in tracing and verifying that the suspected problematic processing is happening.	**Working Memory:** Maintaining State; **Pattern Recognition**

Table 2.2. Decision requirements table for data flow analysts

Decision and Cognitive Requirements	**Why Difficult**	**Visualization Solution**
Locate functions that are vulnerable if “critical” values are different than the expectation.	Large number of functions and values. Not clear how much detailed knowledge of program purpose, and what individual functions do, is necessary.	Visual indicators to denote as much irrelevant/less important code as possible. For example, with visualization of the entire binary, graying out of irrelevant code.
	Following input values through code, assumptions are made about what code is doing that are sometimes incorrect and need to be remembered.	Quick, easy method of keeping track of assumptions and uncertainty about the constraints on the data values and functions while allowing for propagation through code. This is informally done with naming and use of uncertainty indicators like “_” and “?” in names.
	Following input values through code, assumptions are made about what code is doing that are sometimes incorrect and need to be remembered **and revised**.	Quick, easy method of keeping track of assumptions. A function-based/data-based bread-crumb trail generated by user for current thread where elements can be categorized to indicate assumptions made or choice points.
	Following input values through code, choices are made about what path to follow that are sometimes dead ends. Choice points need to be remembered to investigate other paths.	Quick, easy method of keeping track of choice points. (Same bread-crumb trail.)
Identify what code does within the “critical” function that could lead to a vulnerability.	Values associated with binary must be identified from earlier code.	
Locate the functions along the paths leading into “critical” functions.		
Evaluate whether values from data on these leading paths could be manipulated to influence “critical” values in “critical” function.		
Evaluate whether values from data on these leading paths are sanitized by checking or manipulating, ensuring that values are within the range that is expected by “critical” function.		

As a direct result of pain point 2 (Figure 2.9), we explored one way to help analysts quickly make judgments about data when they are assessing every instance of a symbol in the code. Inspired by Microsoft Word co-location information displayed in source, we hypothesized that we could reduce analysts' cognitive overhead by providing a search and decision tool to make navigation easier. In the cognitive walkthroughs, the analysts performed a text string search for a symbol of interest, and the tool highlighted examples of the string. However, the cognitive overhead was high: the analyst still had to search out each highlighted string, make a judgment about whether it needed further analysis, remember the judgment, remember which instances had been evaluated, and search through the code to find the next instance of the string. A tool that lists each instance of a variable along with some context information, and recorded some basic decisions about each instance, could help analysts to make priority judgments more quickly and address memory failures by having a taint-informed colocation (summary) view of variables.

Two novice analysts developed a proof-of-concept IDA Pro [55] plugin to display co-location information for instructions identified as tainted by the Ponce plugin. We did not test this proof-of-concept visualization; focusing our limited resources on developing data flow visualization requirements, we leave further development and testing to future work. Appendix F describes our IDA Pro plugin.

2.4.3 Discussion

These results informed our initial lists of elements of data flow analysis, presented in Figure 2.10. Interestingly, different elements are important at different stages of vulnerability assessments. Elements that are included in a visualization would need to be different depending on the stage of the assessment. For example, elements of algorithmic descriptions and diagrams provided during an end-of-project report would provide the most abstract data flow elements that are useful for understanding. These diagrams are usually one-off creations, but if they were standardized and saved as artifacts with previously-worked projects, they could help with gaining context for new projects. For our goal, creating a visualization to help analysts gain understanding in the first place, we need to consider *how* analysts generate these diagrams.

We also noted a cross-cutting issue for visualizations developed to help analyze data flow: uncertainty about hypotheses about elements needs to be represented. Sometimes these uncertain hypotheses about a data input or data process may be used for mental simulation of program execution. Sometimes these hypotheses have been partially verified. Sometimes these hypotheses represent knowledge-based judgments of likelihoods, e.g., that programmers typically use a given design pattern for a specific reason. In any case, visualizations will need to support representing these uncertainties.

Sources of Data

- direct data (user) input
- network
- file
- internal program data
- pointers to data

Program Representation of Data

- type / structure
- size possible
- size expected
- storage in memory (addresses in pointers, registers, buffers, stack, heap)

Data Flow Operation Categories

- copy (implies new location and variable name)
- subset
- manipulate multiple sources to create a single new data source (e.g., concatenate different sources)
- manipulate one input data to create new data
- manipulate to obfuscate (may be same as previous)
- compare two data inputs
- access data (read) (Is this operation ever done in the absence of another operation?)
- write to unused memory
- write to previously used memory (implies overwriting previous data)
  - partial overwrite
  - full overwrite
- erase from memory; complexity here is related to scope of the operation (e.g., is all of memory erased?)
- constraint
  - check length
  - check type
  - check content
- relationship to privileged operations, e.g., `exec`

Relationships between Data

- co-occurrence
  - structures
  - arrays
- memory access patterns
- structures, e.g., linked lists, doubly-linked lists, red-black trees
- values, e.g., hashed locations

Figure 2.10. Initial list of data flow analysis elements created based on results of cognitive walkthroughs.

2.4.4 Conclusions

Our interviews and cognitive walkthroughs showed that data flow analysis is a cognitively demanding task requiring extensive domain knowledge and focused attentional resources. It is performed by a small group of experts using different tools and no prescribed approach. Current tools are not usually integrated into these current, individualized workflows. Analysts often do not have the cognitive bandwidth to stop their analysis and look for or learn new tools.

These observations have guided our goals for this and future research:

How can we design binary analysis tools to minimize the time human analysts spend doing work that software is better at, and maximize human analyst access to relevant knowledge for extracting meaning from code and making good judgments, while integrating with existing workflows and providing benefits to outweigh the costs to users.

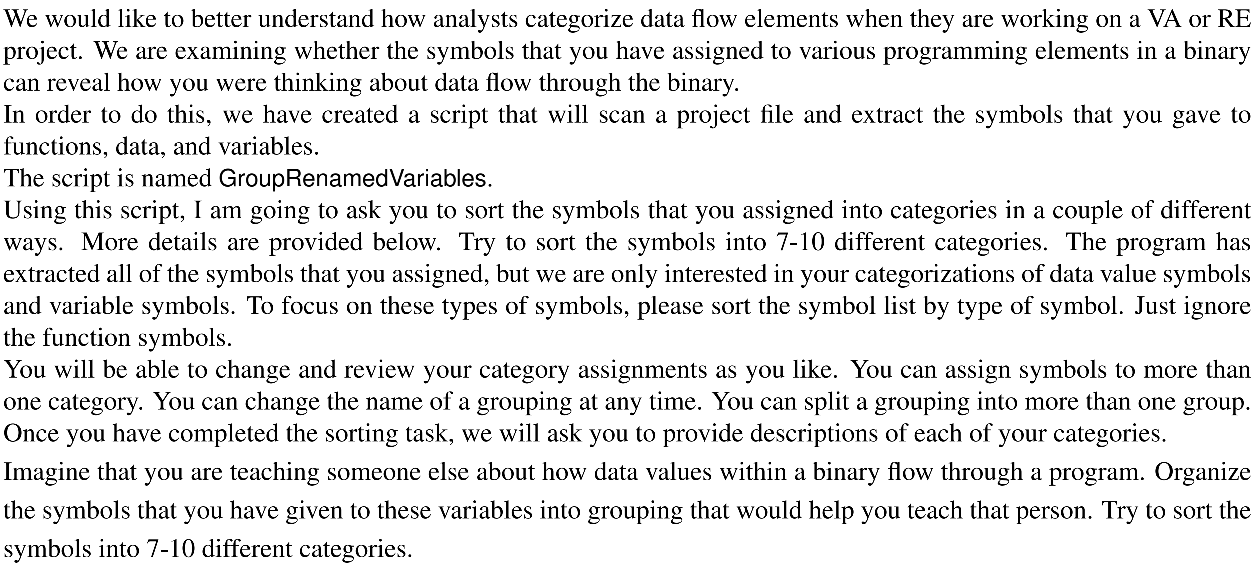
2.5 Modified Sorting Task

Next, we needed to develop the list of requirements, or a list of essential data flow elements and relationships, that generalized across diverse binary programs and analysis goals. We considered conducting additional cognitive walkthroughs[5], but we decided instead to analyze analyst artifacts via a modified sorting task, described below, for three reasons. First, our requirements were to enable a new type of visualization, not an analysis environment; walkthroughs of other data flow tasks required more understanding of and interaction with the analysis environment and would have yielded little specific data flow information. Second, we wanted to capture information critical to understanding data flow across a wider array of program types. Third, we wanted to utilize an analysis technique that would rely less on recall and explicit reporting of thought processes and, perhaps, reveal automatic processing associated with data flow analysis and understanding.

We hypothesized that binary analysts might reveal general purpose data flow elements through a sorting task [92] over their own meaningful data variable and value names. When binary analysts work, they use specialized reverse engineering tools to discover program behavior and record what they discover. These tools allow analysts to add comments and to rename code elements like functions and variables, propagating assessment-relevant names through the code base. When they encounter a previously-renamed element in other contexts, they can know what important information has already been discovered about that element.

Unfortunately, analysts name both data flow elements and categories of elements according to analysis goals and personal preference, making a typical sorting task unable to reveal mental models shared across analysts and projects. Further, analysis projects span weeks, making it infeasible for analysts to independently perform the same analysis. To address these challenges, we added

[5] We did design and dry-run an additional cognitive walkthrough focused on determining the impact of a known vulnerability by understanding a CVE in file. However, the dry-run indicated that significantly more work would be needed to design an effective cognitive walkthrough. See Appendix B for more details.



We would like to better understand how analysts categorize data flow elements when they are working on a VA or RE project. We are examining whether the symbols that you have assigned to various programming elements in a binary can reveal how you were thinking about data flow through the binary.

In order to do this, we have created a script that will scan a project file and extract the symbols that you gave to functions, data, and variables.

The script is named GroupRenamedVariables.

Using this script, I am going to ask you to sort the symbols that you assigned into categories in a couple of different ways. More details are provided below. Try to sort the symbols into 7-10 different categories. The program has extracted all of the symbols that you assigned, but we are only interested in your categorizations of data value symbols and variable symbols. To focus on these types of symbols, please sort the symbol list by type of symbol. Just ignore the function symbols.

You will be able to change and review your category assignments as you like. You can assign symbols to more than one category. You can change the name of a grouping at any time. You can split a grouping into more than one group. Once you have completed the sorting task, we will ask you to provide descriptions of each of your categories.

Imagine that you are teaching someone else about how data values within a binary flow through a program. Organize the symbols that you have given to these variables into grouping that would help you teach that person. Try to sort the symbols into 7-10 different categories.

Figure 2.11. Instructions provided to analysts for the first stage of the modified sorting task, extracting categories of label names from analyst artifacts.

a second stage to the sorting task, allowing us to derive our static requirements for our data flow visualization.

2.5.1 Experimental Setup

The first stage of the modified sorting task consisted of analysts sorting the products of one of their own past projects into categories important for understanding data flow. To help the analysts in this sorting task, we created a program that pulled analyst-assigned variable names from a code base, presenting the names and allowing analysts to request their decompiled context by clicking on a name. The program displayed the entire list of names and allowed the names to be sorted into analyst-defined categories one by one or in groups.

We asked seven analysts to select a completed project with data flow considerations for the sorting task. Figure 2.11 shows the instructions given to these participants. Projects included a variety of applications and operating system drivers. The selected programs provided anywhere from 200 to over 500 names. We asked analysts to spend up to 40 minutes going through the names they had assigned and binning them into groups based on how they would teach someone else about how data values flow in the code. As expected given the time constraints, analysts were only able to categorize between 72 and 110 names into 6 to 11 categories. To ensure that the important categories of data elements had been captured, we asked analysts to review the entire list at the end of the sorting period for missed categories; no analyst felt that categories were missing. Analysts then assigned category names to each of their groups and explained why that group was

SET 1: Data Flow Relationships

1) Data flow represents initial data configuration.
2) Data flow represents communication from system and from user(s).
3) Data values influence control flow, usually through static data elements such as a list of things that direct control (e.g., magic values).
4) Data flows in isolated parts of a program that talk to each other.
5) Data values that represent states, e.g., where the data is. These values tend to be dynamically updated.
6) Special global variables that store data that reach into other parts of system, e.g., large complex globals, mailboxes.
7) Data values that live inside functions, e.g., loops, and iterators. (This type of category was not identified by many of the projects.)

Figure 2.12. Set 1, describing data flow relationships identified by the modified sorting task.

important for understanding data flow. Our collected data consisted of these category names and their descriptions. The analyst-created sorting task category names varied across analysts; program type and analysis goal had a significant impact on the created categories. Because category names and descriptions were derived from proprietary assessments, we will not share these intermediate results.

To determine which category names described similar data flow elements and which names described unique aspects of data flow, we added a second stage: an additional level of categorization by a separate group of analysts. A panel of six experienced binary analysts (one of whom had participated in the original categorization task) and one experienced source code developer reviewed the sorting task categories and descriptions; each member of the panel categorized the analyst-created categories, and then, working together, the panel identified similarities and differences across the analyst-created categories that were important for understanding data flow in binaries. We added these important similarities and differences to our preliminary list of data flow elements, creating a list of required data flow elements to be represented in our static data flow visualization.

2.5.2 Results

Again, because category names and descriptions were derived from proprietary assessments, we will not share the intermediate first stage results.

The second stage results consisted of sets of categories represented by the category names and types from the first stage. These sets are presented in Figures 2.12, 2.13,2.14, 2.15, and 2.16. Note that these sets describe different views of data and data relationships. The panel came to consensus about these different views of how to organize the categories provided by the first stage.

The panel claimed, during the discussion, that a visualization that provides a summary of these types of data flow relationships would not be helpful. However, we note that we do not yet know

SET 2: High Level Data Flow Relationships

1) Inputs
2) Outputs
3) Does this data influence something else?
4) Is this data influenced by something else?
5) How much does this data value change?
6) Other relationships between data values (e.g., value of datum1 is the length for datum2)

Figure 2.13. Set 2, describing high level data flow relationships identified by the modified sorting task.

SET 3a: Phases of Analysis

1) Finding boundaries for where to investigate.
2) Identifying uncertainty and trying to reduce that uncertainty.
3) Asking question of what role does this have initially, and within the system?
4) Re-evaluation leads to filtering that takes you back to Phase 1.

SET 3b: Axes Many labels are combinations of these axes, e.g., low-level semantics, high-level semantics, search notes)

1) Some things are used for the analysis; they are specific to the goals of the analysis. E.g., critical, debug info, overwritable, static, boring.
2) Some things define the meaning of the structure, e.g., https structure, gif.
3) Some things are simply RE notes or low-level semantics; e.g., unknown, complex, potentially unsafe.

Figure 2.14. Sets 3a and 3b, describing phases of analyses and categories of reasons for categorization as identified by the modified sorting task.

SET 4: Axes

1) What the data semantically/syntactically is, e.g., string, gif, html.
2) How the data is structured or laid out in memory.
3) Who "owns" the data and can update it, free it, and see it.
4) Where the data comes from/goes to (flow), e.g., from users, network.
5) What the data is for.
6) How the data is used, e.g., lookup tables, whether it is transient or modified or constant.
7) Where the data lives or the scope, e.g., global, task-specific.
8) Uncertainty analysis conclusions.
9) Relationships among things.

Figure 2.15. Set 4, describing axes over which data might be characterized as identified by the modified sorting task.

SET 5

1) Special global or ambient information
2) Communication outside system
3) Communication within a program
4) State/input to control decisions
5) About the analyst's process (not data flow)
6) Control as data
7) Initial data or configuration

Figure 2.16. Set 5, describing other types of categories identified by the modified sorting task.

if there are ways of measuring these categories that would be informative in a visualization. For example, frequency information about each of these elements cross-referenced with control flow or proximity information about each element may be useful.

2.5.3 Discussion

The two-stage modified sorting task is designed to understand and help specify essential elements of user mental models. It differs from a typical sorting task in which elements to be sorted are defined beforehand. In a typical sorting task, participants each sort the same elements to reveal, through consensus grouping, similarities in mental models.

In the two-stage modified sorting task, we relied on the domain experts, i.e., our binary reverse engineers, to identify the relevant elements, i.e., the names related to data flow. In our task, each participant sorted different elements from extended, large projects analyzing different types of binaries for different analysis goals. While the task was straight-forward for the analysts, resulting category names reflected personal preference and analysis goals. Distilling the sets of category names and descriptions into visualization requirements required a second stage of abstraction, with domain experts, to identify the similarities and differences between the categories described in the first stage. The collaborative second-stage grouping revealed important sets of elements and similarities in how participants think about data flow elements.

Artifact analysis, such as this modified sorting task, can be powerful for understanding the mental models of experts in a domain: artifacts can be systematically analyzed without incurring the cost of devising controlled but realistic projects with different goals. Additional artifacts that might be explored similarly include analysts' change history for names and analysts' comments in the binary code, which summarize their discoveries.

It is difficult to assess the replicability of the results generated from this work. Several factors may make it difficult to reproduce our results. Our preliminary interviews and walkthroughs tested only a few people under each protocol and focused on a single type of data flow task, i.e., attack surface characterization. Further, the results of the modified sorting task may have been biased by the functionality of the programs selected or the range of potential vulnerabilities, and the judgments of our panel of experts may have been skewed by their work. Despite these concerns, we incorporated several strategies to increase the likelihood that our results are replicable. We used a range of approaches: interviews, walkthroughs, and the modified sorting task. We captured the essential data flow elements from a range of projects with different analysis goals.

2.6 Initial Requirements

We used the results from the modified sorting task, augmented with results from the semi-structured interviews and cognitive walkthroughs, to derive a data flow taxonomy. This taxonomy, or set of static visualization requirements, describes types of data elements to be represented, types of relationships to be represented, and types of information to be conveyed via a data flow visualization to support binary analysts. Figures 2.17 and 2.18 provide this set of initial requirements.

We did not attempt to address the outstanding question: **how can we capture the RE process in the creation of these data flow graphs?** Analysts identified a major pain point in that they lack good ways to diagram data flow patterns in binaries (as distinct from algorithmic data flow), and they lack good ways to mark whether each branch has been explored yet or not. We describe mechanisms for interacting with our data flow graphs in Section 5, but we do little to address this question other than to claim that, if implemented with annotations per our requirements, our data flow representation should help with capturing the RE process.

We next needed to evaluate the utility of our produced requirements. To do this, we assigned visual design elements to the elements in our requirements (taxonomy). We then produced a visualization of a binary and evaluated the utility of that visualization, improved the visualization, and iterated. We describe this initial instantiation and iterative refinement in Section 3.

Nodes: What do analysts want to know about the data values? Information about the data value might be conveyed by visual variation in node (fill effect, shape, outline effect, size, etc.). Color and outline color are often used, too, but color is typically a redundant coding with some other feature because of prevalent color discrimination variation in the population.

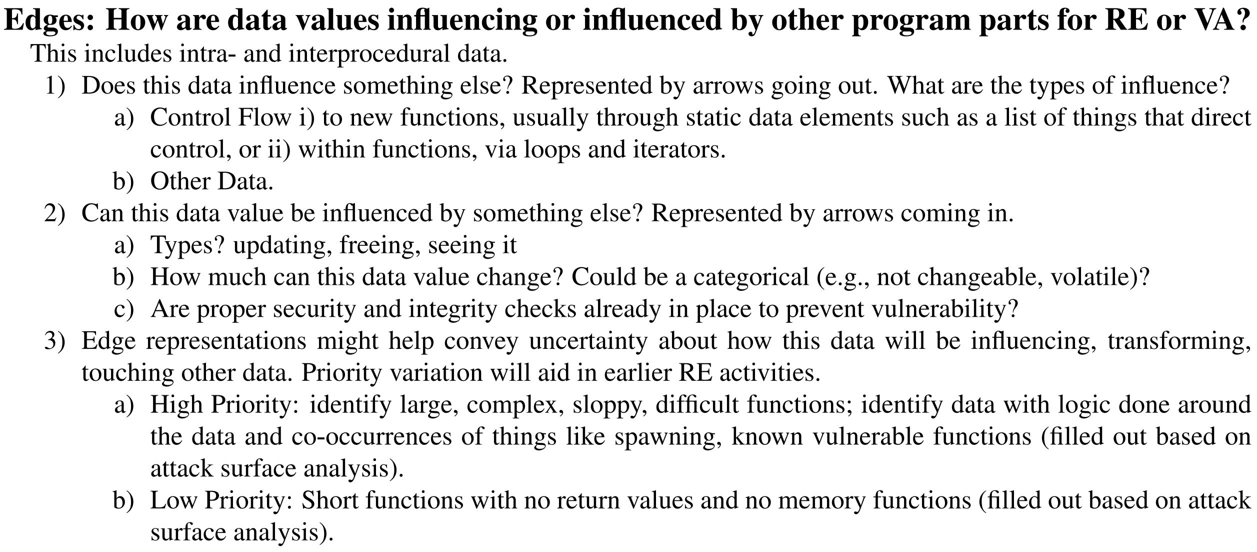
1) Where data come from
    a) Input or communication: user-generated, from system, from file
    b) Initial data configurations
    c) Internally-generated: values replicate portions of existing data through concatenation or parsing from other data
    d) Internally generated: values are the result of evaluating data sources, e.g., returning a state variable from a function call, or calculating from other data values
2) Data values that represent program states, i.e., where data is
3) Frequency (absolute number, number of functions, log normalized) of accesses – *or* numbers of reads and writes; this feature functions differently early in RE as compared to later in the process;
4) Output; this information could be conveyed based on spatial layout. Canonically it is related to when the data re-used, but we could consider other layouts. The output mechanisms are central to the goals that analysts have, and evaluating paths; therefore, analysts may want output to be spatially displayed more prominently.
5) Special global variables that store data that reach into other parts of system, e.g., large complex globals, mailboxes
6) Types: GIF, string, html, and others from our categorization task
7) Value relationships between variables

Information about groups of data values that are consistently related in the program might be conveyed by a consistent spatial grouping and alignment.

1) Data in isolated parts of a program that talk to each other
2) Given relationships, e.g., datum 1 value is the length of datum 2; there may need to be special visual depictions in nodes for information like length
3) Can type information for a structure be conveyed in a similar way? Should consistent correlations between fields in a structure be edges or a different visualization element?
4) Could templates for these consistent relationships be created and useful? Is there enough consistency and frequency for templates to be useful for reducing the cognitive load of creating this component of the visualization?

Information about data values that is boring might be conveyed by removal from the visualization or hiding within an edge, e.g., local variables, data already proven safe.

Figure 2.17. Initial requirements for information conveyed by nodes in a directed graph data flow visualization.



Edges: How are data values influencing or influenced by other program parts for RE or VA?

This includes intra- and interprocedural data.

1) Does this data influence something else? Represented by arrows going out. What are the types of influence?
    a) Control Flow i) to new functions, usually through static data elements such as a list of things that direct control, or ii) within functions, via loops and iterators.
    b) Other Data.
2) Can this data value be influenced by something else? Represented by arrows coming in.
    a) Types? updating, freeing, seeing it
    b) How much can this data value change? Could be a categorical (e.g., not changeable, volatile)?
    c) Are proper security and integrity checks already in place to prevent vulnerability?
3) Edge representations might help convey uncertainty about how this data will be influencing, transforming, touching other data. Priority variation will aid in earlier RE activities.
    a) High Priority: identify large, complex, sloppy, difficult functions; identify data with logic done around the data and co-occurrences of things like spawning, known vulnerable functions (filled out based on attack surface analysis).
    b) Low Priority: Short functions with no return values and no memory functions (filled out based on attack surface analysis).

Figure 2.18. Initial requirements for information conveyed by edges in a directed graph data flow visualization.

Chapter 3

Visualization

Because binary analysts are very comfortable working with directed graph representations, and because the data flow elements were consistent with this type of representation, we next iterated on finding visualization design elements in an elaborated directed graph representation that could convey the required information. As shown in Figures 2.17 and 2.18 previously, we assigned data elements like data values and memory locations to types of nodes; we assigned information about types of influence or relationship to edges. We assigned conveyance of other types of information to grouping, layout, or annotation, or left them to be determined. We specified visual representations for our data flow taxonomy elements, the final versions of which are specified in Tables 3.1, 3.2, 3.3, 3.4, and 3.5 at the end of this section.

These tables describe general requirements for a user-centered graph representation of data flow. Graph elements include nodes (locations and values) and directed edges (influence). Although the draft representation described here is a graph, the ultimate visual representation designed is not constrained to be a graph. The two primary goals of this representation are: 1) to show data locations and dependencies, including control dependencies or influence, possible values, and relationship constraints (i.e., within and between procedures, what is influencing the data?), and 2) to make it easy to diagram data flow patterns, including indicating whether each branch has been explored yet or not.

Again, these specifications delineate the types of information that we would like conveyed about data flow in a data flow representation for binary analysts. The binary analysts we consider are those performing reverse engineering (RE) for vulnerability analysis (VA), malware analysis, and mitigation development. This representation is intended to help binary analysts transfer knowledge about data flow to themselves (returning to the analysis task later) or to other analysts. This representation is focused on transferring knowledge from static analysis of data flow and is not currently intended to help transfer information about dynamic paths. We anticipate that an analyst will create this graph through discovery and analysis, use this graph for navigation, and take some working notes within the graph. We anticipate that the analyst will use multiple other tools (e.g., IDA Pro, dynamic evaluation) and representations (e.g., the CFG, call graph) in conjunction with this representation.

In this section, in addition to providing the final versions of these specifications, we provide example annotations and roles and interpretations in Tables 3.6 and 3.7. We also describe our iterative development and design process, during which reverse engineers frequently reviewed the adequacy of the data flow elements and the design choices made in our visualization. We do not present all of the iterations from this phase of the development process. Instead, to give a sense of our development and refinement process, we present the first evaluation in Section 3.1 and a description of one round of refinement in Section 3.2.

3.1 Initial Instantiation

We further developed our data flow requirements list while manually creating a data flow visualization for the Cyber Grand Challenge [50] binary CROMU_00034 (Diary_Parser)[1], choosing specific instantiations of visual design elements.

3.1.1 Experimental Setup

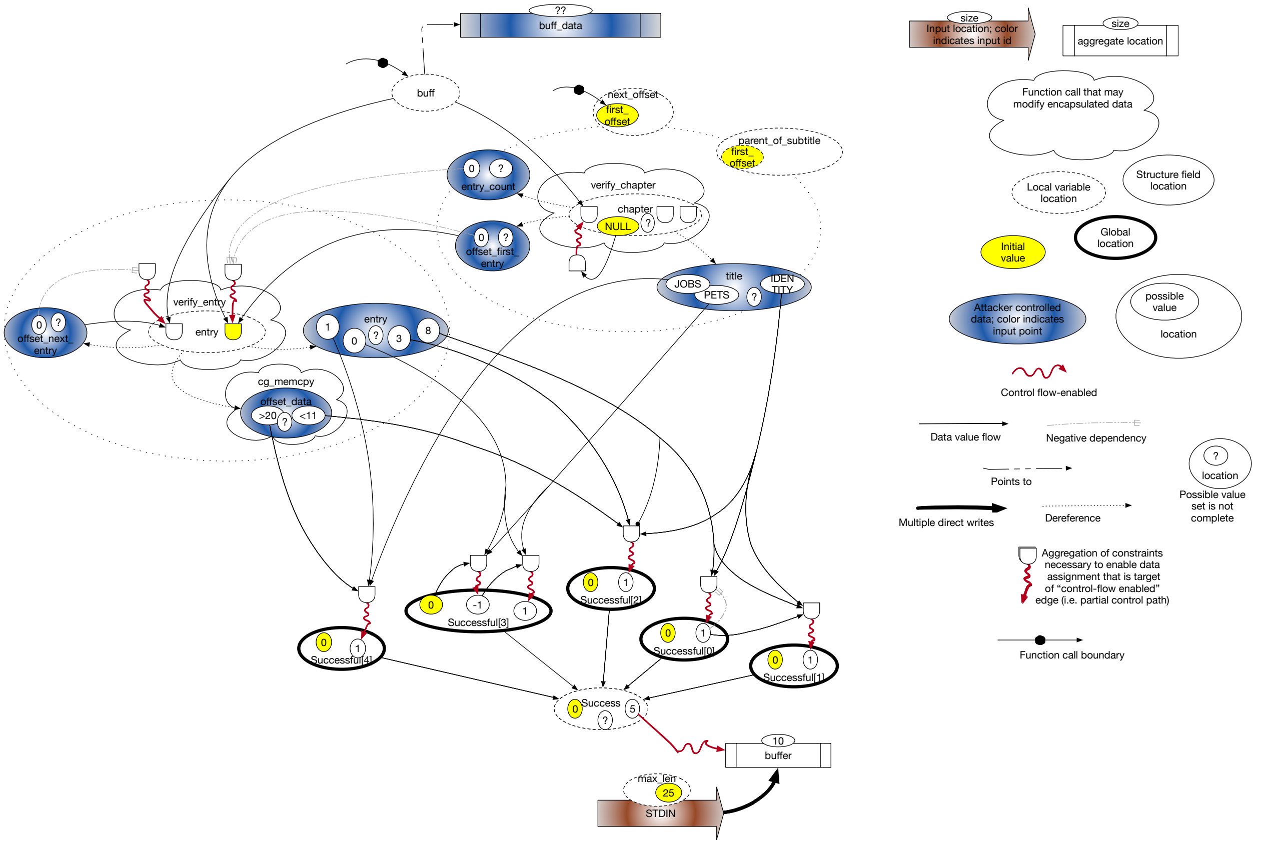
For this experiment, one binary analyst manually RE'd the *source code* for CROMU_00034. The analyst focused on understanding vulnerability 2 (described by CWE-121 [30]) and data flow related to how an attacker might trigger that vulnerability through specialized input. This vulnerability is a simple buffer overflow that is only reachable when an attacker provides a "diary" that meets five specified conditions. The relevant source code consisted of about 500 lines of code. The analyst had not investigated this source code before, nor was the analyst familiar with the specified vulnerability other than as described in the source README provided [104].

Using the taxonomy guidelines developed after the modified sorting task (Figures 2.17 and 2.18) as a guideline, the analyst created a static visualization depicting discoveries about the data flow related to understanding vulnerability 2. The analyst made solitary decisions about how to represent each feature in the graph over the course of about four hours. Figure 3.1 shows the first graph produced; all manually created graphs in this document were created using The Omni Group's OmniGraffle diagramming tool, version 7.9. Note that this graph *does not* follow many of the design choices present in our final requirements; instead, it shows our initial straw man attempt at assigning visualization elements, many of which were changed. Further, this graph does not depict the entire program, nor does it depict all data flow related to vulnerability 2.

Next, eight binary analysts and a cognitive psychologist reviewed this graph with the original analyst. Six of the other analysts were unfamiliar with the code base. The original analyst described the vulnerability and trigger mechanism using this visualization.

[1] CGC Challenge binaries for DECREE are provided by DARPA [19]; versions ported to standard operating systems have been released by TrailOfBits.[20]. We use the DARPA challenge name for binaries, but we provide the TrailOfBits name in parentheses.

Figure 3.1. First straw man attempt at assigning visualization elements to data flow elements in the context of an analysis. This graph represents incomplete data flow understanding related to triggering of vulnerability 2 of the DARPA CGC challenge CROMU_00034.



3.1.2 Results

In Figure 3.2, we describe the evaluation team's feedback and observations about this initial graph. Recall that this visualization is intended to help convey understanding of the CGC vulnerability.

As a result of this feedback, we iterated on the visualization elements and presentation of the data flow graph. The next version of the graph is shown in Figure 3.3. This second version incorporated the idea of layers: the **Memory** layer showed locations and points-to relations, the **Taint** layer added value flow edges, and the **Influence** layer showed values and influence edges. These layers, pulled directly from the full graph, are shown in Figure 3.1.2.

3.1.3 Discussion

After this exercise, we began documenting our requirements and selected visualization elements in a spreadsheet. We created two tables documenting requirements for nodes and requirements for edges. This first pair of tables did cover most of our final node and edge types as well as some annotation types. However, the tables were not self-consistent, and we changed many of our visualization elements as we iterated. Tables 3.1 and 3.2 give our final requirements and selected visualization elements for nodes. Tables 3.3 and 3.4 give our final requirements and selected visualization elements for edges. The visualization elements, however, are just those that we selected; other representations of these requirements may be significantly more effective.

Our taxonomy at this point consisted of guidelines that might have been incomplete or at the wrong level of abstraction. We needed to gain confidence that we were on the right path to developing a visualization that would be useful for analysts. Thus, development going forward needed to be iterative, taking into account subject matter expert (SME) feedback.

Further, we needed to begin to address several major issues:

Scalability Can this sort of representation be as effective and created for a larger binary? It may be more fruitful to scale based on workflow rather than the whole binary: How could this sort of visualization could be constructed within an analyst's current workflows? How can the analyst *easily* build up a representation like this? And how can we represent uncertainty in edges?

Integration How would this sort of representation be created? It takes a couple of hours for about 500 lines of source(?) code. Can elements be easily added during initial program analysis?

Selectivity The current graph is selective, and the selectivity is based on the analyst's understanding of the vulnerability. What would happen if the structure of all the code was mapped, or at least the structure for parts of code that had been previously analyzed?

- This graph appears to scale better than the corresponding static value flow (SVF) graphs showing how memory is related [103], but it is not clear how well it will scale to realistic programs. *This was never addressed.*
- Analysts liked tags on the bubbles. They could see how lots of information about data variables could be documented, e.g., interval, maximum, relationship between max length and in-depth constraints
- Nodes: clouds (functions), ovals (variables) and circles (data values)
  - Visual depictions need to be more easily distinguishable
  - Function clouds may not be necessary
  - How does one indicate the type of data values, e.g., numeric, string, html, gif?
  - Data type information, e.g., int or uint, should be specified via a toggleable layer.
  - Should whether a data value is calculated be depicted in the type of influence, in the variable, or in the data value itself?
  - How does the graph depict that a data value is not-changeable? Is it captured by the lack of influencers on the variable, the data value?
- Edges: influence
  - The goal of this graph is to answer the question "what are the shenanigans to get to this point in program?" Understanding edges and influence seems to allow analysts to answer that question.
  - The graph is incomplete, e.g., check happen on chapter that are not represented.
  - Definitions of the edges need to be about how they influence (either through control flow or in other ways). There should not be a separate entity for designating control flow influence from the arrow itself.
  - A negative dependency is currently depicted through a shadowed edge that branches from the constant edge. Other analysts did not understand this type of constraint.
  - What is the principled reason for distinguishing influence from data values from influence from a variable? Why would an analyst want to depict one and not the other? Could the important distinction be coded in the edge as well? Or as a constraint type?
  - There should be a beginning state of influence uncertainty.
  - Many types of influence that have been noted (like overwriting, freeing, checking) are not displayed.
- Consistency
  - Why are there two bubbles called entry? *One is a local variable called entry, the other is a structure field called entry.*
  - Why are there two things in the entry blob? *There are two different assignments to entry.*
  - How does the graph represent complicated or repeated data structures like the structure containing entry?. *It does not. But conventions from symbolic memory graphs [38] might help.*
- Understanding
  - Analysts expressed confusion about what *exactly* attacker-controlled means. The initial analyst stated, "The attacker gave me those bytes."
  - There was significant ambiguity about whether the word "location" means data or program. *This was resolved to mean data location only in our requirements document.*
  - There is a desire to use color to show influenced data.
- What is most important is to show data locations and dependencies, including control dependencies; possible values; constraints; and control flow. What is influencing the data?
- There are three things that this representation should communicate, and analysts might be better served by allowing for different levels or layers and views:
  - I have a program location; under what conditions do I arrive here?
  - Where do the values that influence my arrival here come from?
  - What are the values?
- Analysts wanted a data flow graph to help with different types of tasks:
  - Show me everything that is possible (a static view). *This is the focus of our requirements.*
  - Show a path that I care about, or a condition to arrive at a state.
- A path on this data flow graph is not really a trace; rather, it is a subset of the providence of a value or variable. However, this distinction can be very confusing, as most analysts think about data flow dynamically.
- Analysts should be able to easily distinguish between data values and names given to program entities.

Figure 3.2. Feedback about our first straw man data flow graph.

Figure 3.3. Second straw man attempt at assigning visualization elements to data flow elements in the context of an analysis of vulnerability 2 of the DARPA CGC challenge CROMU_00034.

Partial DRAFT Data Flow Graph Representation
of Cyber Grand Challenge (CGC) Challenge Binary CROMU_00034,
a.k.a. Trail of Bits Version Diary Parser

Generated from source
Describing vulnerability 2 (CWE-121) and data flow related to
how an attacker could reach that vulnerability through specialized input
as described here: https://github.com/trailofbits/cb-multios/tree/master/challenges/Diary_Parser

Shelley Leger, Karin Butler
Jan 3, 2018

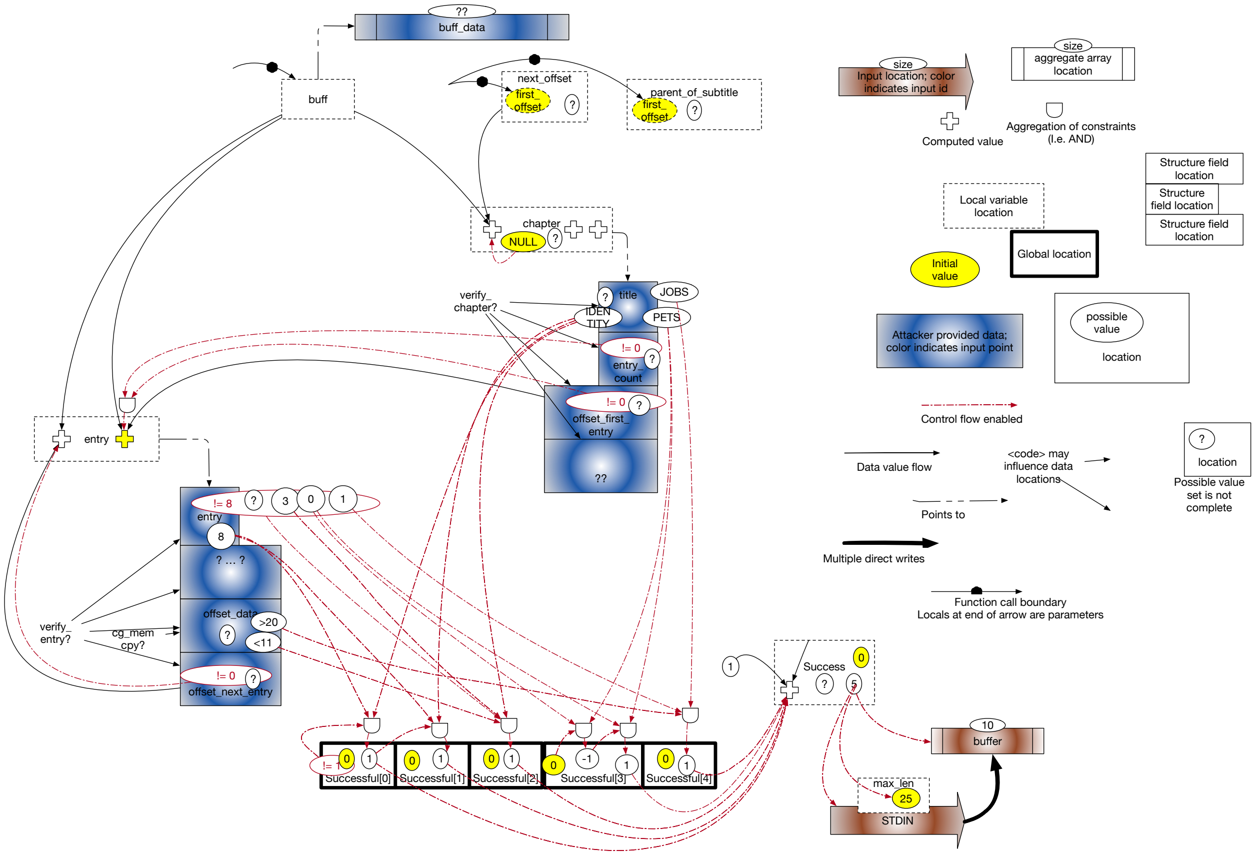
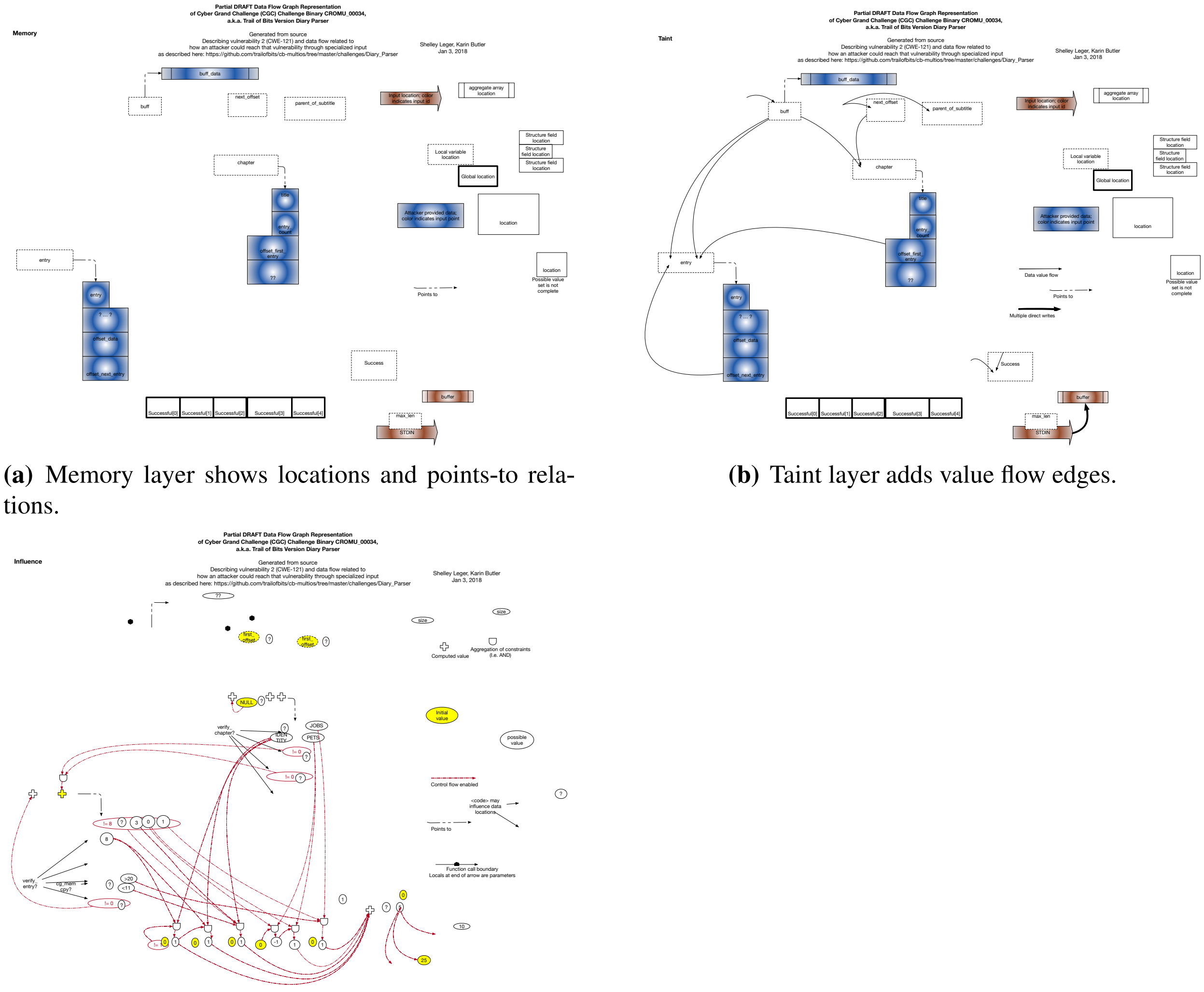


Figure 3.4. Individual layers pulled from second straw man visualization. These figures show how toggle-able layers can help an analyst to pare down the information in the graph to focus on specific types of locations and influence.


Partial DRAFT Data Flow Graph Representation
of Cyber Grand Challenge (CGC) Challenge Binary CROMU_00034,
a.k.a. Trail of Bits Version Diary Parser
Memory
Generated from source
Describing vulnerability 2 (CWE-121) and data flow related to
how an attacker could reach that vulnerability through specialized input
as described here: https://github.com/trailofbits/cb-multios/tree/master/challenges/Diary_Parser
Shelley Leger, Karin Butler
Jan 3, 2018
buff_data
buff
next_offset
parent_of_subtitle
Input location; color indicates input id
aggregate array location
Structure field location
Local variable location
Global location
chapter
title
entry count
offset_first_entry
??
Attacker provided data; color indicates input point
location
entry
?…?
offset_data
offset_next_entry
Points to
Possible value set is not complete
Success
buffer
max_len
STDIN
Successful[0]
Successful[1]
Successful[2]
Successful[3]
Successful[4]
Taint
Data value flow
Multiple direct writes
Influence
first_offset
size
Computed value
Aggregation of constraints (i.e. AND)
NULL
JOBS
PETS
verify chapter?
Initial value
possible value
Control flow enabled
<code> may influence data locations
Function call boundary
Locals at end of arrow are parameters
verify entry?
cg_mem cpy?
>20
<11
!= 0
25
10

(a) Memory layer shows locations and points-to relations.

(b) Taint layer adds value flow edges.

(c) Influence layer shows values and influence edges.

3.2 Refinement

Using an iterative process similar to an Agile approach to software development, we further developed the data flow requirements list. Experienced binary reverse engineers frequently reviewed design choices and accessibility of data flow information.

3.2.1 Experimental Setup

For this walkthrough demonstration [71], approximately 15 binary analysts, one source analysis expert, three visualization experts (including expertise in sensemaking of intelligence information), and two cognitive psychologists walked through the visualization together. Discussion was focused around the questions "How do analysts discover things? What are they discovering?" The group was asked to ignore the discovery process and instead focused on how data flows once there has been sufficient discovery.

The group focused on the CROMU_00034 graph in Figure 3.3 as a specific example of the requirements listed in our nodes and edges tables. In the visualization used, nodes represented data values and locations, and edges (or arrows) represented influence. Spatial ordering in the boxes, left to right and top to bottom, tended to map to instruction order. Data flow was represented left to right in an attempt to reduce back edges.

3.2.2 Results

Discussions from the walkthrough demonstration resulted in observations about the current state of the practice and a list of features that analysts asked for in a data flow visualization tool. Most of the discussion focused on building *on top of* our straw man static visualization; very few changes to the visualization elements were suggested other than as related to color and thickness. We present the results of these discussions in Figures 3.5 and 3.6.

3.2.3 Discussion

Given this feedback, we extended our requirements to call out interactivity requirements. Additionally, we iterated on the visualization elements and presentation of the data flow graph. Further iteration of the requirements also occurred during our evaluations (Section 4).

We represent our final requirements, including our data flow taxonomy and specific visual instantiation choices, in tables describing interactivity requirements, example annotations, node information, edge information, and example roles and semantic interpretations. Our data flow taxonomy elements and interactivity requirements are given in Tables 3.1, 3.2, 3.3, 3.4, and 3.5.

- Automated tools do provide data flow information, but they are not trustworthy and provide too much information.
- Analysts are used to viewing and interacting with control flow graphs (CFGs), zooming in and out to answer questions like “What value does j depend on?” They need to discover possible values and encode them once they know them.
- In attack surface characterization, specific values are less important, but paths are very important.
- Uncertainty is not represented in these tools; rather, it is in analysts’ heads (probabilities).
- Scaling visualization is a challenge, even for simple programs.
- Specialized views, like hiding things, could help certain parts of the analysis but hinder others. Toggle-able layers might help this.
- It is difficult to balance representation of what is important and what is too much information, particularly as we do not know yet which features are most important for the analysts to key off of.
- This appears to be a classic sensemaking problem: it includes lots of hypotheses, uncertainty, and so on.

Figure 3.5. Observations about the current state of the practice provided by analysts.

We provide example annotations and roles and interpretations in Tables 3.6 and 3.7 We summarize many of the final selected visualization elements in a key in Figure 3.7.

We present our final visual representation of CROMU_00034, using this key, in Figure 3.8. For those interested in working through a very simple example with many of the visualization features, we provide the sums example graph in Figure 3.9.

- Analysts want to hide things that they understand, or at least the details that led to that understanding, but also capture what they understood about it and *how* they knew it (evidence). They also may want to collapse the display of unimportant information, particularly as a full binary would likely be overwhelming.
    - Starting with everything collapsed changes information foraging, but this mechanism has worked well in other scenarios [82][32]. Possible alternatives include collapsing influence edges and expanding on demand, or providing a fish eye view [101].
    - Code folding is really powerful, but many tools do not support it. For example, a tool might pull out the code relevant to highlighted things and show them together.
    - Detailed questions tend to be more local, although influence questions can be global; however, very local questions are often easy enough to answer with the current tools and are "uninteresting".
    - Influence is very important (except in the case of related constraints, e.g., "with this value, this path would never happen").
- Analysts would like a data visualization tool integrated with data flow analysis tools.
    - An API might run an automated analysis and then pop up a visualization at points of interest.
    - The graph needs to help disentangle apparent flow from actual flow.
    - The tool should let the analyst get under the hood to see the data flow. Questions might include 1) Does this one thing (identified via a click, e.g.) go to places of concern? 2) Where have data structures been copied multiple times? 3) Where have structures been passed by value versus passed by reference? 4) If this element has the value 3, how would that affect the rest of the visualization? 5) Are there places where data does not actually flow even though it appears to (i.e., there is no sink other than being passed as a parameter)? 6) Do this set of values ever interact with this other set of values, or are they guaranteed to be independent? 7) Between this source and sink, what data flows from here to there? What are the paths from here to there? 8) Or, to a certain depth (e.g., "one step away", what are the relevant flows?
- A tool should help analysts with both memory and communication of the progression of the analysis:
    - keep track of hypotheses and allow analysts to go back and alter them when something is wrong.
    - represent uncertainty about the accuracy of the analysis.
    - provide a summary of high-level things and what has been touched, including encoding how long or how often an element has been analyzed.
    - represent the minimal amount of information to let an analyst know what he did.
    - help transfer information to another analyst (very hard).
    - fast-forward through past analysis or timeline to refresh memory, though it would be nice to be able to hide or delete irrelevant explorations.
    - provide support for investigation *and* bookkeeping, including annotations and bookmarks.
    - provide externalizations of the current hypothesis under test and where analysts still need to look
- Effective layouts could help with context-switching, memory, and retention.
    - Consistent spatial relations would help with memory. Specifically, an analyst would probably want the same layout a week or a month later.
    - A visualization could act as a memory aid: analysts could map their mental models onto a visualization once they are familiar with it, and then the visualization itself can help refresh memory.
    - Local changes (within a parent node, e.g.) are probably okay. Combining parent nodes may not be, as the graph would have to be rearranged in such a case. However, animating rearrangement has worked in other situations.
    - What sequencing information is necessary? Can spatial layout be used to convey instruction order?
- Analysts need to express and move between multiple levels of abstraction, possibly seeing several at once.
- Analysts often work with multiple views and want to save them.
- Initially analysts may want things to stand out based on how much they are used in the program.

Figure 3.6. Features desired by analysts in a data flow visualization tool as described by analysts in a group walkthrough of our straw man data flow visualization.

Table 3.1. Requirements for information to be conveyed through value and location nodes. * denotes types expected to be updated.

Type	Sub-type	Details	Visual Design Element Instantiation in Current Representation	Source Artifact Evidence	Notes / Examples
value		a set of values irrespective of where those values are stored	most types differentiated by display name; an empty oval may indicate uncertainty in the value, unimportance of the value, or simply an as-yet unspecified value		Currently, a value has a single definition and is thus associated with a single instruction / assignment. A value is associated with the specific location in which it is stored (one-to-one); value nodes are children of location nodes. For simplicity of presentation, the location holding the value (the parent location node) may be omitted when the location is not relevant to the analyst's analysis.
	constant	a specific constant value or named constant	oval shape containing constant or name; will have no incoming value flow edges	instruction and constant operand	Example CONSTANT VALUE: a node may have a value 8 or a named constant such as "PETS", and it should be possible to choose, on a per-node basis, which is displayed.
	computed	the value resulting from a (possibly complex) computation	plus-sign shape; incoming value flow edges represent values used in the computation	instruction set involved in computation, instruction performing the definition (assignment) of the final computed value	Details of the computation should be visible when selected or requested. The node itself is associated with the final assignment in the computation. Example COMPUTATION: z = x + 23*y would be associated with the final assignment to z. Incoming edges could be either x and the result for 23 * y, in which case the computation would be +, or incoming edges could be x, y, and 12, in which case the associated computation would be x + 23 * y. This is an example where the generation algorithm has multiple options, and transformations or simplifications could be used to construct simpler or canonicalized graphical representations.
	constraints*	a generalization of a constant, which may be a value set, a named set, an uncertain set, or the complement of such a set	oval shape displaying set or constraint name; may want concise or graphical representation of constraint imposed on value	branch instruction for constrained value; set of instructions for value sets	Representation of this data may not scale nicely. Perhaps the set/constraint should only be visible when selected or asked. Example CONSTRAINT: location y has a value that must be ¡= x, e.g., as imposed within a loop by a loop check.
	uncertain*	an unknown or uncertain* value or value set	oval shape displaying a question mark	set of possible definition instructions	A question mark indicates an unknown or unconstrained value, or it may indicate uncertainty in the value. This may also be represented by leaving the node display name blank.
location		a logical memory location (registers may be represented if specifically requested); this set does not necessarily map 1:1 onto the architecture because logically distinct uses of the same physical or virtual location are represented as separate nodes	rectangular shape; types differentiated by outline or fill	declaration or allocation sites (instruction or allocation function call site)	A location may contain (be parent of) multiple value nodes. For simplicity of presentation, when a location is associated with only one value, the value's display name may be displayed instead of the location's displayed name and the value node itself may be omitted. Unlike values, a location may be associated with multiple code locations. Registers are not represented unless they are needed to untangle complicated computations, in which case they are represented as stack (temporary) locations.
	local	register or stack; temporary location (lifetime within a specific scope or context)	dotted outline	definition	Some stack locations may not be represented, in particular for parameters or virtual registers, if not necessary to distinguish the value or location from more permanent locations.
	heap	shared, dynamically allocated location; context-independent lifetime	solid outline	allocation site	These locations do not end their lifetime (and thus theoretically lose their values) on exiting a context. Lifetime extends from some sort of allocation to some sort of deallocation.
	global	statically allocated location	node filled with gray	space allocated	Global locations are considered shared across everything.
	shared memory*	location accessible outside the program under analysis	NOT YET REPRESENTED	allocation or identification site	Shared memory is expected to be modified outside the context of the represented program.

Table 3.2. Requirements for information to be conveyed through aggregate, code, and communication nodes, and through annotations on nodes. * denotes types expected to be updated.

Type	Sub-type	Details	Visual Design Element Instantiation in Current Representation	Source Artifact Evidence	Notes / Examples
aggregate		collections of nodes	NOT YET REPRESENTED		
	array	a collection of adjacent locations of the same type	rectangle (long width to height ratio) with double lines on sides	allocation site(s)	An array may or may not represent individual indices explicitly inside of it (sub-locations laid out within the array node), and it may choose to represent only a subset of indices. An array also may or may not have individual indices or their structures represented as separate entities in the graph (e.g., represented separately through pointers).
	structure	a collection of adjacent locations of potentially different types	vertically stacked locations (rectangles) representing the fields of the structure	allocation site(s)	Restating, the fields of the structure are stacked, the bottom of one immediately adjacent (in layout) to the top of the next, and no explicit edge exists between them. To support uncertainty in the width and number of fields, a structure may contain implicit aggregates representing an unknown number of fields or an unknown size of fields. Note that this particular representation does not map to source code analysis as it does not explicitly represent the uncertainty of the final structure layout in the compiled binary.
	and constraints	a collection of constraints that all must be met	a Boolean AND symbol (shield)	conditional branches	Each code-influence edge is triggered when the constraints are met; all input edges must be satisfied for output edges to be satisfied.
	or constraints	a collection of constraints, any of which may be met	a Boolean OR symbol (pointy shield)	code or branches	Any input edge may be satisfied for output edges to be satisfied.
code		a section of code that may act on (influence) a location or value	text-only nodes (no edge or fill)	code or boundary (function, module)	Example CODE: a function, module, or equation might modify the values in specific locations, potentially in unknown ways. This is not a data location, but code is essentially a location (large group of locations) that can influence the values in another location. We represent this as a summary node containing the name of the code (e.g., the function name) with an influence edge to the location it might influence.
communication		data values come from or flow to something outside the binary	STDIN/STDOUT are large arrows; other input/output types are NOT YET REPRESENTED	evidence	We may need to encode different types or sources of input/output differently.
	input*	data values come from something outside the binary	STDIN is a large arrow with an outgoing data value flow edge	initial source	Example INPUT SOURCES: user-provided, from the system, or from a file.
	output*	data values leave the binary	STDOUT is a large arrow with incoming data value flow edges	uses of locations in output functions	Output mechanisms are central to the analysis goals and path evaluation (e.g., we may want to prove that there is not an information leak), so we may want the output to be spatially displayed more prominently. Example OUTPUT: STDOUT, file write, network send.
annotation		additional information about locations or values			
	initial configuration	initial value; the first definition that affects a location	value nodes filled with yellow	first definition	We may need to encode different sources of initial configurations at some point, but we haven't run into that yet.
	data type*	type of data stored in a location	NOT REPRESENTED	declaration / evidence	Type encodes how to interpret the data and what to expect. Example TYPE: uint, short, string, html, gif, binary file.
	size	number of bits in a unit; for arrays only, the number of bits in one index, not the size of the fully allocated array	NOT REPRESENTED	declaration / evidence	Could be represented (and in some cases is) using the width of field locations in aggregate structures. For aggregates other than arrays, the size should be the sum of the sizes of the individual locations in the aggregate.

Table 3.3. Requirements for information to be conveyed through value flow, points-to, comparison, and control influence edges and function boundary annotations. * denotes types expected to be updated.

Type	Sub-type	Details	Visual Design Element Instantiation in Current Representation	Source Artifact Evidence	Notes / Examples
value flow		the source node (value or location) affects value at the destination	solid black line from source node to destination node	list of copy or computation instructions moving the data	If the destination value is a computed value, the source data is used in the computation (possibly by copying a part of the value). If any other kind of value, this edge represents a copy of the source value into the destination location. Edges can represent a transfer *summary*; each may be associated with multiple instructions involved in a computation and/or multiple copies from intermediate locations; all of these instructions should be associated with the edge, especially as unimportant nodes are folded into surrounding edges.
function boundary	parameter	only on value flow edges, value is passed from a caller into a function	large dot on edge from the data node in the caller to the data location or specific value in the called function; no distinction between a parameter and a return value	dot itself is associated with function call site	Function subgraphs may be inlined or shared, based on the analyst's desires for each function and initial heuristics. When functions are inlined, the same function is represented by multiple subgraphs, but the subgraphs are relatable (if requested by the user), though by default changes to one subgraph will not be made to other instances of the subgraph. If the function subgraphs are shared, each parameter has a separate incoming edge from each call site.
	return value	only on value flow edges, value is passed back from a function to its caller	large dot on edge from the data node in the called function to the data location or specific value in the caller	dot itself is associated with function call site	If function subgraphs are shared (not inlined), each return value has a separate outgoing edge to each call site. There is currently no mechanism to relate parameters and return values from the same call site explicitly.
points-to		the source node points to (is the address of) the destination location	black dashed line consisting of a long dash and two short dashes	associated definitions of the source node	Example POINTS-TO: a global buffer named current has an incoming points-to edge from an implicit global location with one possible value: &current.
comparison		the destination constraint value is generated through comparison against the source node (location or value)	black dotted line with very long spaces between dots	comparison instruction(s)	The comparison value could be placed in either location being constrained by the compare. Frequently one location is used regularly after the compare; we put the comparison value in that location. Example COMPARISON: local variable w is compared to the result res returned from a function call; a comparison edge runs from res to w because w is the location used after the comparison.
control influence	positive	a control flow decision determines that the destination node is used	black dotted line; the destination of the edge is used (or activated) if the source value is used (is active)	branch instruction (binary offset)	If the destination is a value, then the value's location is assigned that value when the source constraint is activated (i.e., the source location holds the source value). If a location, then the location is allocated and used if constraints are met. Example CONTROL INFLUENCE: POSITIVE: if a location x is assigned 1 when y is 8 and 0 otherwise, then the value node 1 that is a child of location node x has an incoming positive control influence edge from the value node 8 that is a child of location node y.
	negative	negative branch of a control flow decision determines that the destination node is used	gray dotted line; the destination of the edge is used (or activated) if the source value is not used (is inactive)	branch instruction (binary offset)	As above. Example CONTROL INFLUENCE: NEGATIVE: if a location x is assigned 1 when y is 8 and 0 otherwise, then the value node 0 that is a child of location node x has an incoming negative control influence edge from the value node 8 that is a child of location node y.

Table 3.4. Requirements for length, sequencing, code influence, synchronization, colocation, and lifetime relationship information. * denotes types expected to be updated.

Type	Sub-type	Details	Visual Design Element Instantiation in Current Representation	Source Artifact Evidence	Notes / Examples
length		source node represents the length of a destination node (usually, aggregate or allocation)	when clear, source node is co-located (top center) with the destination location to which it applies; otherwise NOT YET REPRESENTED	allocation instruction(s)	Usually applies to aggregates, allocations, and shared memory.
sequencing		represents specific sequence of program execution; like an instruction level control flow graph	SHOULD NOT BE REPRESENTED	an ordered pair of instruction offsets	Sequencing information is not intended to be displayed, other than loosely as described in the element description. These undisplayed edges may be represented in the underlying graph simply to enable queries about control flow. Note that such queries are best put to other artifacts such as the disassembly or control flow graph (CFG).
code influence	allocatable	from a code node, the location can be allocated	code node has points-to edge to location		Lifetime information is generally not yet represented.
	freeable	from a code node, the location can be freed	code node has points-to edge to location		Lifetime information is generally not yet represented.
	readable	from a code node, the location can be read	code node has value flow edge from location		Visible influence (the ability of code to read a location) is not usually represented; it becomes useful when you have specific locations in the code influenced by the location (i.e., the edges are to more specific nodes).
	writable	from a code node, the location can be written	code node has value flow edge to location		Does the code have permission to write (or overwrite) the location? Is it expected to?
synchro-nization	lifetime*	two locations or values may have a value relationship that holds across the lifetime of the locations	NOT YET REPRESENTED other than implicitly through the layout of length nodes	set of relevant definitions and uses showing relationship	This captures relationships that hold across the lifetime of the locations. Example LIFETIME SYNCHRONIZATION: a buffer allocation's length and the value of the length variable should be the same. Example LIFETIME SYNCHRONIZATION: if the variable age is less than 12, the variable grade could be 4 or 5 but could not be 17 or 18.
	sometime*	two locations or values may have a value relationship that only hold at specific code sections	NOT YET REPRESENTED	set of relevant definitions and uses showing relationship	Example SOMETIME SYNCHRONIZATION: a block is guarded by the value check x == y. In that block, x and y have the same value, but outside of that block, that value synchronization may not hold.
colocation	spatial*	two locations may have a position (location in memory or architecture) relationship that holds, such as adjacent or with 16 bytes between	NOT YET REPRESENTED	set of relevant allocation sites and related instructions (e.g., the allocation function)	Colocation is represented above for arrays and structures; this additional mechanism describes other relationships. Example SPATIAL COLOCATION: structure A is always allocated 64 bytes from structure B (e.g., because of specialized memory management). Colocations generally encode potentially uncertain information such as vulnerabilities that might be introduced by the compiler.
	subset*	one location represents a subset, or slice, of another location	NOT YET REPRESENTED	set of relevant definition, calculation, and use sites	Example SUBSET COLOCATION: a node represents an array element, but the index of the element is calculated and may change. Visualization might have lines from the bottom corners of the large aggregate to the top corners of the indexed location. Example SUBSET COLOCATION: an ELF file resides in a buffer, the ELF header resides in a subset thereof, and the text section resides in a completely different subset.
	overlap*	two locations may overlap each other in memory	NOT YET REPRESENTED	set of relevant definition, calculation, and use sites	
lifetime	*	two nodes (locations or values) should be valid for the same timeframe	NOT YET REPRESENTED	set of relevant definitions and lifetime ending instructions	Example LIFETIME: a buffer and the length of that buffer have approximately the same lifetime as each other (they are tightly coupled) and should share a lifetime edge.

Table 3.5. Interactivity requirements for a data flow visualization to support binary vulnerability analysis.

Type	Details	Visual Design Element Instantiation in Current Representation	Notes / Examples
code / artifact source	link to source artifact (binary or source / decompilation)	NOT REPRESENTED	This is a direct link to ground truth (code) that this node or edge represents; it is not a graph entity. It includes: file, function, line (source/decompilation) or instruction offset (binary), token / character / operand.
element GUID	a unique node/edge (i.e., element) identifier	NOT REPRESENTED	The GUID is for programmatically interacting with elements of the graph; it is not intended to be displayed.
node display name	an analyst-assigned name, which may change over time	text in node	Example LOCATION: a node changes from SP+24 to int@sp+24 to selector to algorithm selector as the analyst gets more information. Example VALUE: a node changes from 8 to PETS to contain information about what a value means.
source	link to the locations or values that influence	when a node is selected, source nodes that influence that node are highlighted	The source could be a location or a value. Sources and destinations for edges are updatable, and edges can connect to either values or locations. Analysts should be able to add, remove, and update edges.
destination	link to the locations or values that are influenced when a node is selected, destination nodes that are influenced by that node are highlighted		
layout	locations of nodes and edges in the overall graph	initial layout may reflect input or output and sequencing (generally, left to right, then top to bottom), but the analyst ultimately controls the layout	Layout by default attempts to convey approximate sequencing of definitions and uses. Inputs flow to outputs, and flow is modeled generally left-to-right and top-to-bottom. As a secondary goal, nodes attempt to lay out children values (definitions) and outgoing edges (uses) secondarily in order of instruction occurrence (sequencing). Understandable layouts should not be sacrificed to sequencing.
annotation	analyst-provided tags, judgments, and notes	an editable text box in a separate view; when requested, all visible elements with a given annotation may be highlighted	All tags, judgments, and notes associated with an element appear as annotations. Annotations may also be represented in other ways later (e.g., though specialized glyphs). Analysts liked the idea of tags on nodes to document lots of information about data variables, e.g., intervals, maximum, relationships between, maximum length and in-depth constraints. Automatic scripts and analyses should also be able to programmatically add and propagate annotations.
filtering	hiding portions of the graph	when requested, the visualization should hide graph elements (while maintaining layout) based on GUID, layer, name, type, or analyst-provided tags or judgments	
hypothesis records	a path through the graph	an ordered list of nodes and connected edges in a separate view	Hypotheses may be saved and re-loaded, highlighted, and treated as any other graph element.
exploration	on-demand graph expansion	when a node is clicked for expansion, source or destination nodes are added to the graph; requesting unshown hypotheses, elements by annotation, or elements by type should also expand the graph	
relative frequency of use	how often the node or edge is executed as compared to the graph elements to which it is connected	relative "depth" represented by node and edge thickness, where the relation can be changed (default is depth of nesting, but other relations might be lower bounds, upper bounds, etc.)	In executing the program, how often is this node accessed as compared to the nodes around it? Currently we use depth of nesting in loops to calculate this: a statement at the highest level has a thickness of one; in a single loop, two; in a doubly-nested loop, three; and so on. Recursion is not yet considered. Other options for relative thickness could look at complexity classes, such as constant, log n, n, etc., for tight or loose lower bounds, upper bounds, or average case. Could have selectable option for which relative thickness to use.

Table 3.6. Examples of annotations to support vulnerability analysis of binaries. Analyst judgments convey information about decisions analysts are making about data or a flow; analysts might use judgment scales to help guide analysis steps. Analyses similarly can use annotations to share information.

Type	Details	Visual Design Element Instantiation in Current Representation	Notes / Examples
priority judgment	how important this element is to follow up with or understand, as compared to other elements		Priority aids in earlier reverse engineering activities, and indicators of priority act as notes to and by the analyst to guide further discovery. At least low and high priority should be supported. Other schemes for categorizing priority may be appropriate.
uncertainty judgment	how much remains to be discovered about this element	values may use “?” labels; edges may lack sources or destinations; locations may lack values and vice versa; otherwise, NOT YET REPRESENTED	Conveys uncertainty about how this data will be influencing, transforming, or touching other data. An initial influence uncertainty for each node will be updated by the analyst or analyses over time. Indicators of uncertainty act as notes to or by the analyst to guide further discovery.
safety judgment	how vulnerable this element is to problematic manipulation		Shows that proper security and integrity checks are in place to prevent a specific vulnerability involving this data or relationship. Safety may be evaluated on multiple axes (different vulnerabilities, e.g.). Could be supported with visualization functionality allowing analysts to tag certain nodes.
interest level / boring judgment	how interesting is this element	removed from visualization or hidden within an edge, when represented; this will be provided by the analyst as a tag on an element	Represented implicitly by being left out. Example BORING JUDGMENT: local variables, irrelevant data. Tied in with priority and safety. The “boring” marker allows for visualization simplifications as these elements can be hidden. Example BORING node: if a function receives parameter y and copies it directly into z without using it anymore, an analyst could mark y as boring. Suppose the one call site called the function using x. There would be one parameter edge from x to y, and one data flow edge from y to z. If y has no other edges to or from it, and the analyst marks y as boring, the visualization could update to show only a parameter edge from x to z where that edge is associated with all the associated source data from the copy from x to y, y, and the copy from y to z.
extent of influence judgment	how much the value of this element might affect the functionality of the program		For edges, how much can the destination data value change based on the source? This may be a categorical (e.g., not changeable, limited change, overwritable). For nodes, how much can the value of the node affect differences in dynamic executions?
attacker-provided or attacker-controlled	whether / how much the values at the location are controlled by an attacker		Used to quickly check source of data for current code of concern to see if it is possible for the attacker to put the program into this state. Might also apply to edges, too, e.g., when an attacker could control whether a control-enabled edge was taken.

Table 3.7. Roles might convey summary information about interpretation of specific groups of data or influence, e.g., through glyphs or other simple additions. Here, we show how some common roles might be identified in the current graph.

Role	**Way to Recognize in Current Representation**	**Notes / Examples**
base pointer	a node (location or value) has a points-to edge to an aggregate and is used in a computed value that then points to a location within the aggregate	Non-aggregates should not be indexed and therefore should not have a base pointer (as distinct from a pointer)
loop variable	a constraint value with both a more frequently used control flow enabled edge (thicker; the loop-taken branch) and a less frequently used control flow edge (thinner; the loop-not-taken branch) represents that the location is a loop variable	A loop variable is a value, not a location, as it represents whether a loop was taken or not within a binary. Loop induction variables (locations) themselves may be difficult to identify in binaries, but the jump back to the head of the loop is (usually) readily identifiable. Loop / iterator edges or node decorators were specifically removed from a prior version of the display to try to keep the visualization a little simpler (e.g., the successful[] display).
state machine	control transfer: location and a set of associated data values represent state variables, e.g., where data is, or how data should be parsed; state machines are locations with values that are each sources of control flow enabled edges; delayed timing information such as that in a handshake is NOT REPRESENTED	Tied in with priority. Example CONTROL TRANSFER ROLE: location state stores the most recently completed stage of a TCP handshake; on receiving a packet, the value in state controls which control path performs the next stage of the handshake.
mailbox	data transfer: large, complex aggregates or locations that enable data transfer between components; a data transfer location is an aggregate that has both incoming and outgoing influence edges from nodes that are in disparate sections of the binary, but external data transfer is NOT YET REPRESENTED	Tied in with priority. Example DATA TRANSFER ROLE: aggregate location mailbox is used for inter-process communication (IPC).
static data	implicitly represented as a location that has only one (initial) value, or a value where the wrapping location has been removed	This node will not be updated after being set.
dead code	graph elements that don't appear or are not connected may be dead, but we currently do not support marking elements as dead other than through annotations or requested removal	We might want to explicitly visualize this, but annotations can support this use case.
parser	locations in an aggregate have multiple explicit values that have outgoing control edges, but the locations do not have incoming value flow edges	The code associated with the control edges and values is the parser; the parser itself is not really represented, though.
length	UNKNOWN AT THIS POINT IF LENGTH IS EVIDENT	May need to manually identify and annotate length role and influence/relationship edge.

Figure 3.7. Summary key of final selected visualization elements.

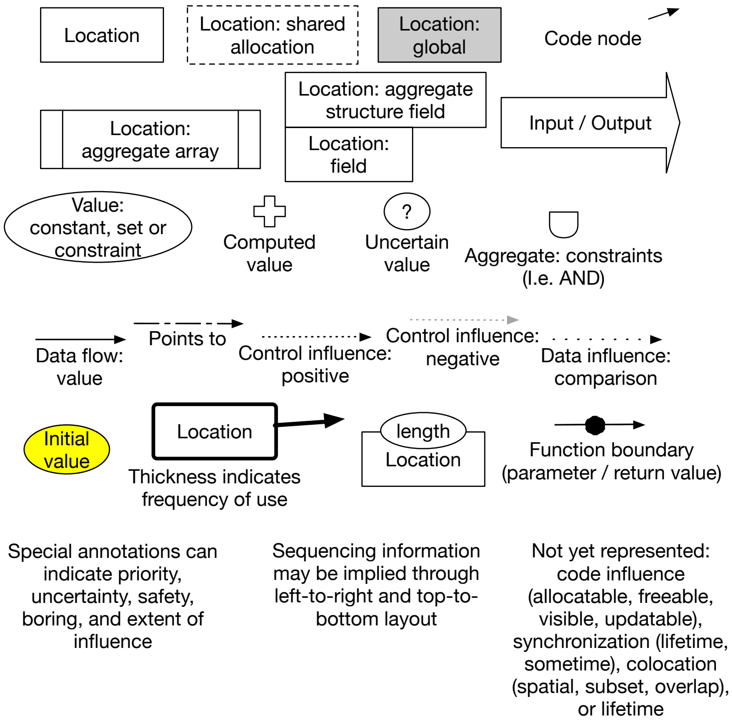

Location
Location: shared allocation
Location: global
Code node
Location: aggregate array
Location: aggregate structure field
Location: field
Input / Output
Value: constant, set or constraint
Computed value
?
Uncertain value
Aggregate: constraints (I.e. AND)
Data flow: value
Points to
Control influence: positive
Control influence: negative
Data influence: comparison
Initial value
Location
Thickness indicates frequency of use
length
Location
Function boundary (parameter / return value)
Special annotations can indicate priority, uncertainty, safety, boring, and extent of influence
Sequencing information may be implied through left-to-right and top-to-bottom layout
Not yet represented: code influence (allocatable, freeable, visible, updatable), synchronization (lifetime, sometime), colocation (spatial, subset, overlap), or lifetime

Figure 3.8. Final visualization of vulnerability 2 of the DARPA CGC challenge CROMU_00034.

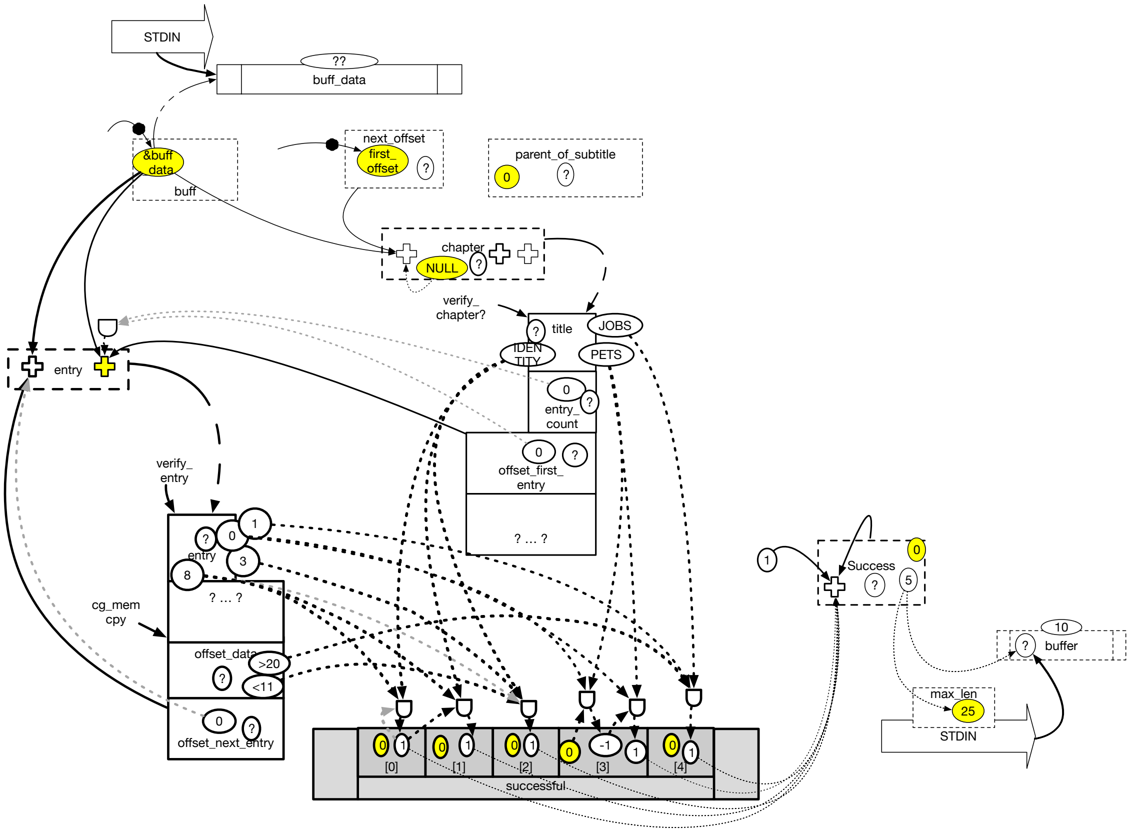
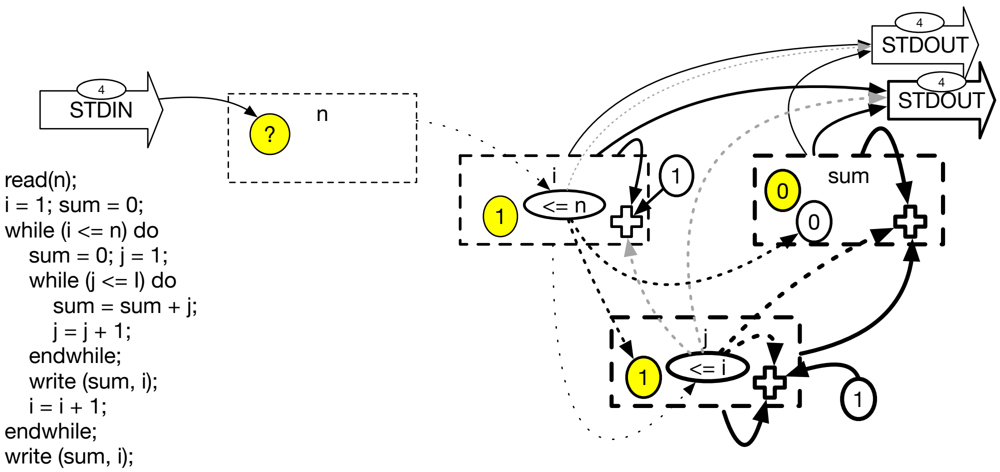


Figure 3.9. Visualization of simple sums function.



```
read(n);
i = 1; sum = 0;
while (i <= n) do
    sum = 0; j = 1;
    while (j <= I) do
        sum = sum + j;
        j = j + 1;
    endwhile;
    write (sum, i);
    i = i + 1;
endwhile;
write (sum, i);
```

Chapter 4

Evaluation

Ideally, in a formal evaluation, we would compare the accuracy and efficiency of analyst performance on relevant tasks with and without our visualization on realistically sized binaries. Unfortunately, several challenges impede a formal evaluation in this realm: it is difficult for a non-expert to judge accuracy of performance, we have not created the workflow integration components of our visualization, we would need well understood and well-crafted data flow vulnerability analysis problems, and we would need analysis problems that can be completed within a couple of hours rather than over the course of days to weeks.

Instead, we performed an informal evaluation on exceedingly simple examples as described here. Vicente (2002) recommended three ways to evaluate requirements developed through the application of human factors methods [63]: 1) a proof of principle through a demonstration that the requirements generated through the cognitive work activities can be used to create a design; 2) an analytical principle that demonstrates that the design reveals important understanding about the domain of interest, and 3) an empirical principle that uses experimental testing of the new design against an existing design or against some benchmark of task performance to demonstrate utility. We conducted proof of principle and analytical principle testing, but we decided that A/B experimental testing was premature because the visualization was not deployed within the analysis environment and only represented a subset of the information needed for a full vulnerability assessment.[1] We focused on answering the following questions: Can other visualizations be constructed with these rules? Can visualizations of different types of data flow problems be constructed and interpreted? Can SMEs interpret visualizations with minimal training? What is confusing? What is missing?

Note, our research is a work in progress. We are attempting to understand the cognitive processes of reverse engineers performing vulnerability assessment and to use these cognitive models to produce a visualization.

It is difficult to assess the replicability of the results generated from this work. Several factors may make it difficult to reproduce our results. Our preliminary interviews and walkthroughs tested only a few people under each protocol and focused on a single type of data flow task, i.e., attack surface characterization. Further, the results of the modified sorting task may have been biased by the functionality of the programs selected or the range of potential vulnerabilities, and

[1] In developing the analytical principle, we performed an A/B comparison with n=1 against an analyst using a traditional binary RE environment. We summarize the results of the binary pre-test in Appendix C.

the judgments of our panel of experts may have been skewed by their work. Because of these limitations, we may not have discovered all of the data flow primitives and relationships necessary for a complete representation.

Despite these concerns, we incorporated several strategies to increase the likelihood that our results are replicable. We used a range of approaches: interviews, walkthroughs, and the modified sorting task. We captured the essential data flow elements from a range of projects with different analysis goals. We used an iterative development and design process during which reverse engineers frequently reviewed the adequacy of the data flow elements and the design choices made in the visualization [71].

Although our proof of principle and analytic evaluations of the visualization show that analysts can represent and answer questions about data flow, additional development is needed along several lines before these visualizations could be deployed to an analysis environment. Our visualizations were created manually. Binary reverse engineers in an operational environment already maintain high cognitive loads without the added burden of creating a visualization. Manually creating the visualizations is untenable, and, although many of the data flow elements can be derived automatically, such automation is not incorporated into current workflows.

Once automation can be used to derive data flow visualization components, new insights will need to be easily injectable into the visualization during line-by-line analysis, and workflows will need to be optimized for stitching together these automated pieces with line-by-line code analysis. The interviews and cognitive walkthroughs revealed that the discovery of data flow information can unfold over the course of an analysis, e.g., identifying that a data structure is used, and then piecing together details about the parts of that structure. Thus, the data flow visualization should support the recording of unknowns and partial insights as they become known during the analysis. Additionally, interactivity requirements derived from our preliminary data gathering indicate that analysts require interactive features that support using the data flow graph to navigate through the code base, as well as features that allow sections of the graph to be collapsed when detailed information is not necessary.

4.1 Proof of Principle

The first test of the list of data flow elements was a proof of principle: could a visualization be created from the data flow primitives and their visual descriptions for a binary program, and would that visualization represent and convey the important information about the data flow vulnerabilities in the code?

4.1.1 Experimental Setup

For this test, a novice reverse engineer just out of an undergraduate computer science program was asked to create a data flow visualization for the simple `sums` example and for two Cyber Grand

Challenge (CGC) binaries CROMU_00065 (WhackJack)[2] and the C++ KPRCA_00052 (pizza_ordering_system)[3] using our list of data flow elements and visualization specifications. This test revealed several ways that the data flow primitives were not specified in enough detail to create the visualization, resulting in minor revisions to the final list of data flow primitives. The second proof of principle task identified a third CGC binary, EAGLE_0005 (CGC_Hangman_Game). This visualization was manually created for the entire binary through a collaboration between the novice and an expert. We did not require modifications to our set of elements, though we did modify our (manual) algorithm for creating the visualization to handle while true loops with a break and non-returning function calls (see Figure 4.3). This visualization represents 408 lines of relevant decompiled binary code. With existing data flow graphs, analysts would not be able to observe the entire binary at once.

4.1.2 Results

The final sums data flow graph is presented in Figure 3.9. In creating this graph, we changed the requirements to specify how to represent loop induction variables and to add a comparison edge between two locations.

During this exercise, the novice reverse engineer made judgments about how to represent certain items, e.g., which location to treat as the source for a comparison edge between two locations. Just as with functions (which one may want inlined for clarity of context or shared for clarity of code structure), this represents a decision that may vary based on location. We view these decisions as *transformations* that represent the same thing. We continue with the view that it is valid to represent things one way that could be represented differently; analysts may have preferences, preferences may change based on program point or analysis goal, and giving analysts flexibility in when and how to apply which transformations may help analysts use these graphs more effectively.

For the next two graphs, Figures 4.1 and 4.2, we present intermediate results of partial analysis of the relevant binaries (not full binaries). These graphs have not been updated to illustrate our final assignment to visual elements. Instead, they reflect graphs at two points in our iterative development process. Further, they do not incorporate all of the visual design elements completely. Finally, Figure 4.1 shows function boundaries around the data flow elements. These were used by the analyst to help map to a mental model of the code and decompilation as the graph was being created. They are stored in another layer and are toggle-able (i.e., they can be easily hidden as a group).

This exercise exposed a need for duplicating conditions and memory locations for clarity sometimes and sharing them other times (an example of a transformation applied based on analyst judgments), mechanisms for representing virtual tables and resulting function calls, and a need to hide information that is prolific when it is not useful (e.g., the this pointer). For example, the bottom of Figure 4.2 demonstrates a function that randomly calls one of two virtual functions. We can clearly

[2] WhackJack has data flow elements that are neither the most complex nor the simplest within the CGC binary set.

[3] pizza_ordering_system has some relatively complex data flow elements as compared to the rest of the CGC binaries in that dynamic dispatch is used.

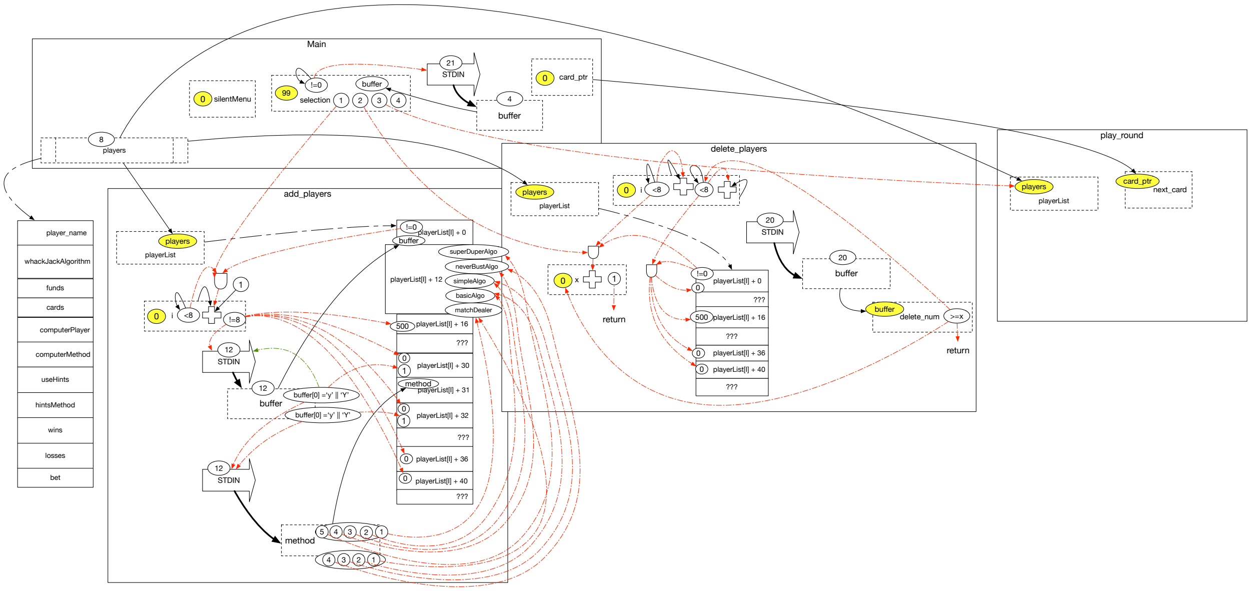


Figure 4.1. A data flow graph manually constructed from the TrailOfBits port of CGC challenge binary CROMU_00065 by a novice using our intermediate data flow requirements.

see that it calls some offset from edx in the virtual table, but we do not have a good representation of the uncertainty in the resulting location. One expert analyst responding to this visualization stated, "overall, I think it's far, far better than looking at the disassembly or the source code", but also stated, "I can't quite follow all of it... and I'm not sure whether I need more practice reading these diagrams, whether the diagram is incomplete [*it is*], or whether the representation needs a little more work."

4.1.3 Discussion

To highlight how this visualization would be useful to binary analysts performing a vulnerability assessment, Figure 4.4 shows the portion of that graph that includes the two vulnerabilities present in EAGLE_0005. In the upper left, 80 bytes from STDIN are read into the 32 byte name buffer located on the stack. This is an easily identifiable stack buffer overflow. This data is then passed, without sufficient checks on the data, into STDOUT; this is an easily identifiable format string vulnerability. These two vulnerabilities are relatively straight-forward to identify via a line-by-line analysis because they are wholly contained within a single function. However, if trying to understand how an attacker might exercise these vulnerabilities, an analyst requires interprocedural data flow understanding of nearly all of the 408 lines of code and data flow depicted in Figure 4.3. Current data flow visualizations do not enable effective visualization of an entire binary in this way. See Appendix D for comparative visualizations of the CROMU_00034 and EAGLE_0005 binaries.

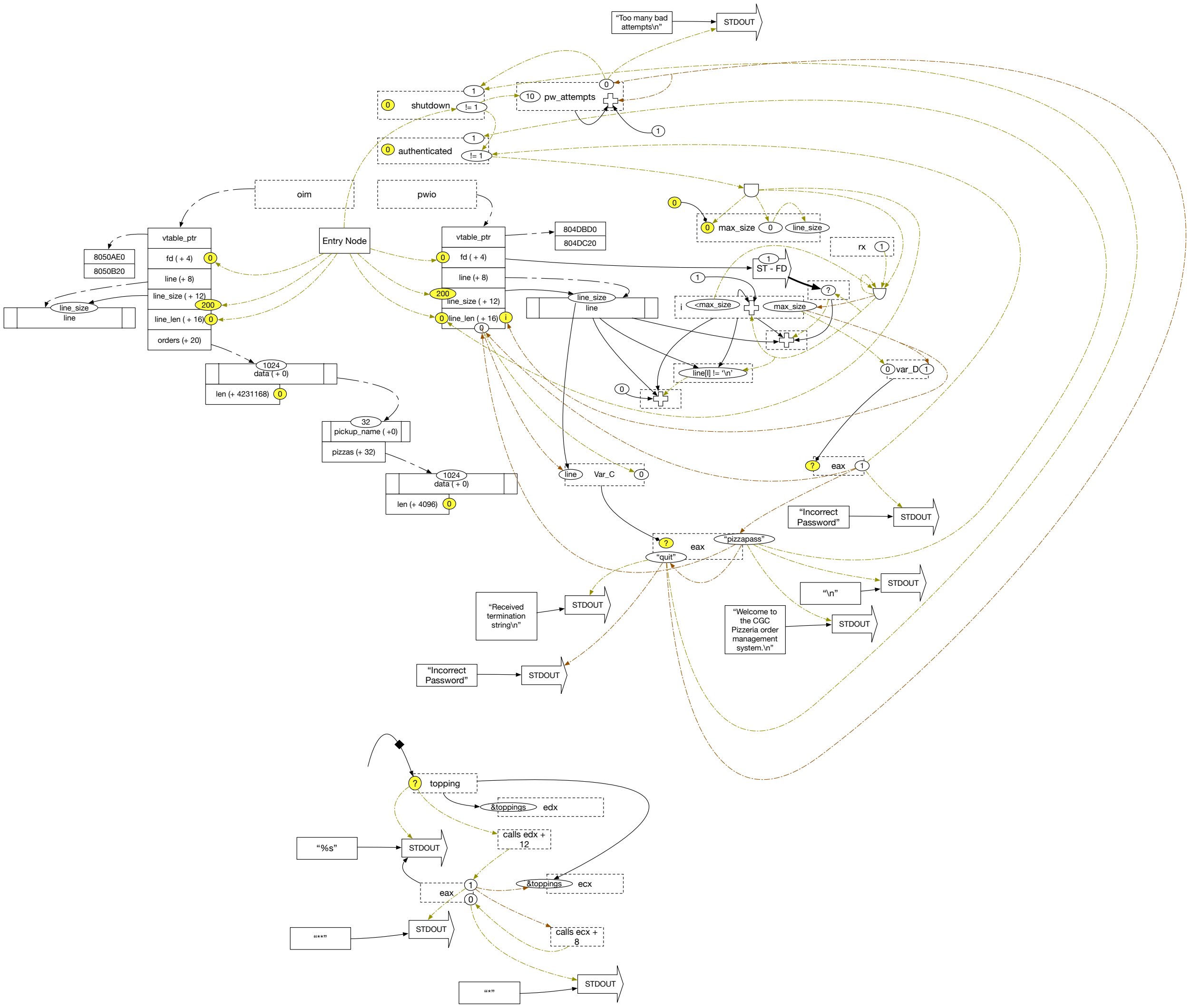


Figure 4.2. A data flow graph manually constructed from the TrailOfBits port of CGC challenge binary KPRCA_00052 by a novice using our intermediate data flow requirements. The source for this binary was written in C++.

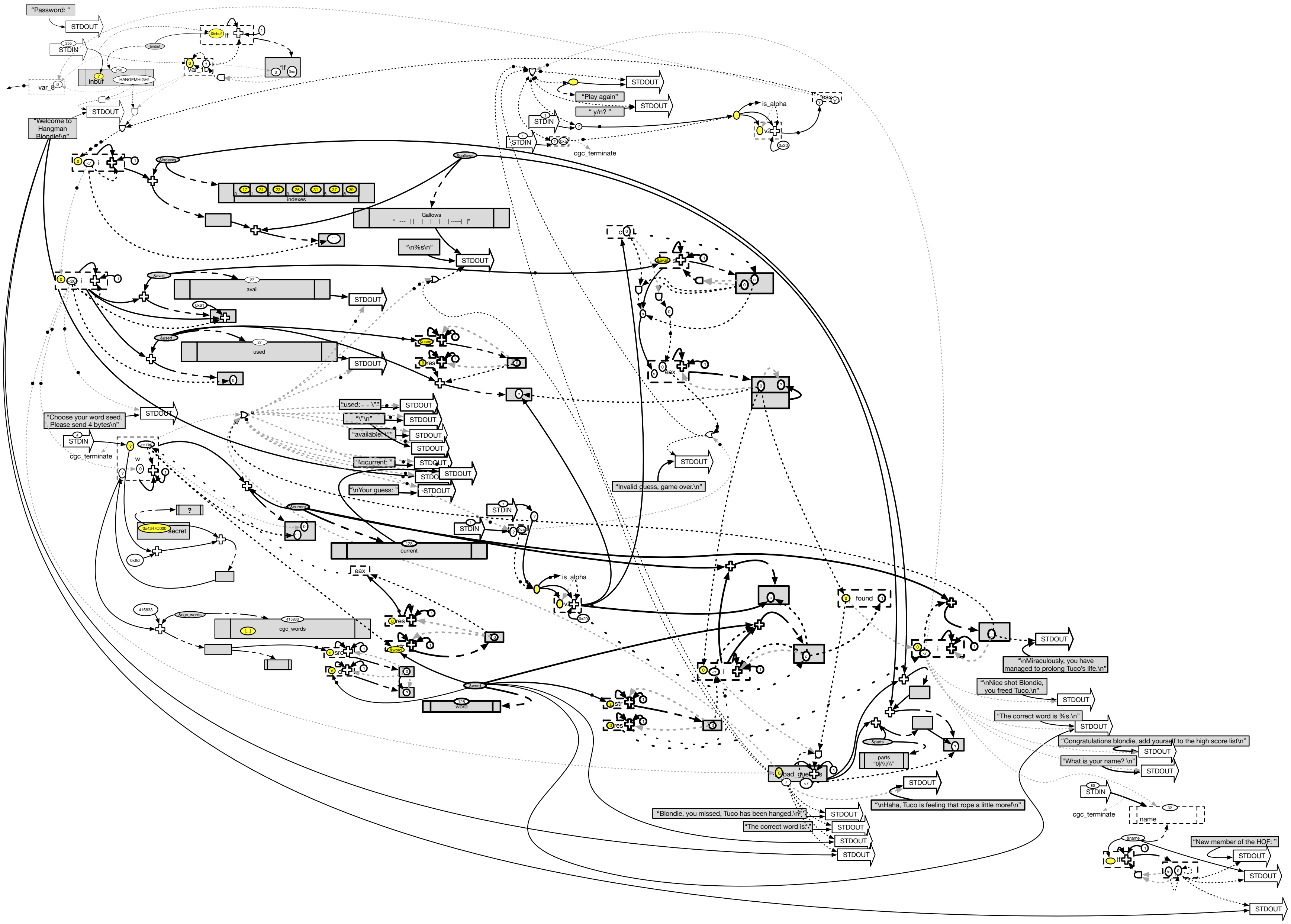

"Password: "
STDOUT
STDIN
inbuf
HANGEMHIGH!
var_8
var_1D
"Welcome to Hangman Blondie!\n"
"Play again"
" y/n? "
cgc_terminate
is_alpha
eax
indexes
Gallows
"\n%s\n"
avail
used
"Choose your word seed. Please send 4 bytes\n"
"used: "
"available: "
"\ncurrent: "
"\nYour guess: "
"Invalid guess, game over.\n"
secret
current
cgc_words
word
found
bad_guesses
parts
"\nMiraculously, you have managed to prolong Tuco's life.\n"
"\nNice shot Blondie, you freed Tuco.\n"
"The correct word is %s.\n"
"Congratulations blondie, add yourself to the high score list\n"
"What is your name? \n"
"\nHaha, Tuco is feeling that rope a little more!\n"
"Blondie, you missed, Tuco has been hanged.\n"
"The correct word is: "
name
"New member of the HOF: "

Figure 4.3. A data flow graph manually constructed using our data flow requirements and final assignment to visual design elements. Generated from the TrailOfBits port of CGC challenge binary EAGLE_0005, this graph encapsulates all instructions from the binary except those from libraries.

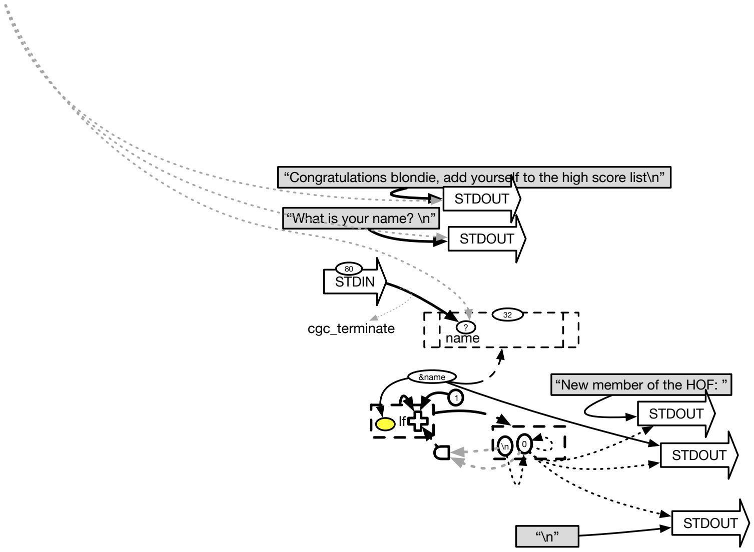


Figure 4.4. The portion of the EAGLE_0005 data flow graph showing the two vulnerabilities known to be exhibited by that binary: a stack buffer overflow vulnerability, and a format string vulnerability.

The EAGLE_0005 example demonstrates how such a visualization might be useful theoretically; we next wanted to gain some confidence that the visualization did, in fact, allow an analyst to answer data flow questions.

4.2 Analytical Principle

The second type of testing followed the analytical principle. For these tests, a list of questions about data flow and important considerations in reverse engineering and vulnerability assessment were derived from the initial project discussions and cognitive task analysis products.

4.2.1 Experimental Setup

An experienced reverse engineer who was not involved in the previous activities was given a 15-minute primer on the graph for CROMU_00034, and then he was asked to answer the questions presented in Figure 4.5 using only the data flow visualization for the EAGLE_0005 binary.

1) Where does an attacker control data input throughout the program?
2) What constraints on the password read into inbuf would allow the body of this program to be reached?
3) Looking at the processing of the input buffer inbuf in the function main (upper left of data flow graph):
    a) What is the initial value of the pointer?
    b) When is the pointer incremented?
    c) Are values being read or written as the array is walked?
    d) What values are being looked for as the array is walked?
    e) What values are written?
    f) When are those values written?
4) Looking at the global avail array initially in the function cgc_reset (left side, middle height, left-most uses in the data flow graph):
    a) Are values being read or written as the array is walked?
    b) What values are written?
    c) When are those values written?
    d) When else is avail read?
    e) When else is avail written?
5) Looking at the global used array initially in the function cgc_reset (left side, middle height, left-most uses in the data flow graph):
    a) Are values being read or written as the array is walked?
    b) What values are written?
    c) When are those values written?
    d) When else is used read?
    e) When else is used written?
6) Looking at the global word array (left-ish side, towards the bottom, left-most uses in the data flow graph):
    a) Where is the array being walked?
    b) For each use, are values being read or written as the array is walked?
    c) For each read, what values are being looked for?
    d) For each write, what values are being written and where do they come from?
    e) How were those source values located?
        i) Follow the flow backwards as far as possible.
        ii) What causes these values to change?
        iii) Are there potential problems with this flow?
7) Looking at the c variable from cgc_doTurn (middle of everything in the data flow graph):
    a) What are inputs to the value?
    b) What constraints are on the value?
    c) To what locations is c written? When?
    d) To what locations is c compared? When?
8) Looking at the name array in cgc_getHighScore (far right bottom in the data flow graph):
    a) What are inputs to the values?
    b) What are constraints on the data input to this variable?
    c) Are there potential problems with this flow?
    d) Similarly, following name through STDOUT?
9) Can you find a place where an array is walked through by incrementing a pointer?
10) Can you find a place where an array is walked through by incrementing a counter and adding that to the base pointer?
11) Can you find where the global array gallows is written?
12) Can you find where the global array current is written?
13) How many guesses do you have before you lose the game?
14) What things cause the game to restart?

Figure 4.5. Evaluation questions for analytical principle testing of EAGLE_0005.

4.2.2 Results

The analyst interviewed preferred language like "When is used being written to?", or "When is data in avail' being updated?", or "Is data in avail being overwritten?" Additionally, the analyst noted, in reference to arrows for pointers, that the directions of the arrows were sometimes confusing because an arrow can suggest a value being read... but a value being read does not represent data flow.

In spite of this, the analyst was able to answer 11 of the 14 data flow questions correctly within 40 minutes. The questions that could not be completely answered in the allotted time involved interpreting pointers and their edges (see questions 4, 6 and 7 of Figure 4.5) and suggest a possible area for improvement of the visualization.

4.2.3 Discussion

Overall, this result gives us some confidence that visualizations produced via our static requirements are useful for answering data flow questions.

However, we need to understand what levels of detailed program understanding are necessary during assessment. The expert analyst who evaluated this data flow visualization noted that answering the questions about data flow without first having a general program understanding "felt weird". As RE and VA incorporate more automated tools that allow analysts to draw conclusions about sections of code without requiring the analyst to understand the line-by-line functioning of code, the understanding of minimal levels of detail will be critical. This understanding will help to effectively incorporate new tools into workflows and to effectively document the results of team assessments completed over time.

Chapter 5

Workflow Integration

As mentioned in Section 4, additional development is needed along several lines before these visualizations could be deployed to an analysis environment. We used human studies to help identify fundamental concepts that could help to organize, scale, or filter data flow graphs to make them comprehensible. To enable integration of our visualization requirements, which described *what* information to display, we need two primary components: a generation mechanism, and a presentation mechanism. We pursued an end-to-end solution to this data flow understanding problem through collaboration with experts in program analysis and, separately, visualization from Georgia Tech's College of Computing.

Our visualizations were created manually, taking up to several hours to generate the simple graphs in this document. The full EAGLE_0005 took even longer. Binary reverse engineers in an operational environment already maintain high cognitive loads without the added burden of creating a visualization; manually creating the visualizations is untenable. We will need a semi-automated creation mechanism to help analysts generate these graphs if they are to prove useful.

In Section 5.1, we discuss initial automated extraction of the basic elements in our graph from source code, creating a json description of a data flow graph. Further automated analysis could extract additional data flow abstractions and relationships. This automated extraction is not be complete; it presupposes strong solutions to the open data flow problems of interprocedural analysis, shape analysis, and type flow analysis. However, this extraction provides a strong base on which to build, and it provides many primitive elements from which human analysts can draw their own conclusions.

Once automation can be used to derive data flow visualization components, new insights will need to be easily injectable into the visualization during line-by-line analysis, and workflows will need to be optimized for stitching together these automated pieces with line-by-line code analysis. We currently only provide a static graph, but our interactivity requirements indicate that analysts require interactive features that support using the data flow graph to navigate through the code base, to collapse portions of the graph, and to support recording unknowns and partial insights through the analysis. To meet these interactivity requirements, specialized layout and interaction capabilities are necessary.

In Section 5.2, we discuss a proof-of-concept visualization to support interactive presentation of data flow graphs described in a json file. The json description files are independent of creation mechanism; some of ours were created manually, and others were created through the dg extension.

5.1 Automatic Generation

Our data flow graph is quite similar to a system dependence graph (SDG) [59], an interprocedural program dependence graph (PDG), which displays data dependencies and control dependencies between instructions. This covers much of the non-judgment information needed to construct our data flow graph.

In our graph, however, we use a handful of transformations, layout constraints, and abstractions to help an analyst explore the graph. For example, assignment nodes in the PDG that assign to the same location are value nodes in our graph, but these value nodes are children of the same parent location node. In this case, the layout constraints enforced by the parent node allow an analyst to easily differentiate between a specific value (definition) being used in an expression and an uncertain value (one of multiple definitions) being used; they are similar to PHI nodes. Similarly, control dependencies may be flattened more than in a PDG (they explicitly connect a trigger value to the value assignment that happens when that trigger occurs), and many control dependencies are elided.

The primary difficulty in constructing PDGs is the need for accurate aliasing information. Similarly, much of the work performed to construct our data flow graph automatically (without annotations) is to infer accurate data dependence information. We build off of dg [22], which builds on top of the LLVM compiler infrastructure [70] and is a component of the symbiotic analysis platform to check common safety properties [23]. Using dg, we begin with constructing our data flow graph automatically from source. We use a points-to analysis [98], a reaching definitions analysis [26], a def-use analysis [45], and a control dependence analysis [45] to compute data and control dependence graphs. We modified dg to produce **interprocedural** dependence graphs and to generate json output in the appropriate format. Appendix E gives an example json specification for the simple sums code. We do not perform all of the simplifying transformations described by our requirements yet. For example, intermediate registers that simply aggregate partial data in a computation should not be represented explicitly in our graph, but we do not yet perform this simplification. Because this code is preliminary, we do not provide pseudo-code here.

As we attempt to generate these graphs from binaries, we will have to address the fact that aliasing information and aggregates will be more difficult to compute. That is, there will be more uncertainty in the initial constructed graph prior to analyst interaction. While we briefly looked into generating such graphs by first lifting into McSema [33], which, given a commercial IDA Pro [55] license, lifts into LLVM IR and thus could directly use the LLVM analysis, we leave generating these graphs automatically from binaries to future work.

5.2 Interactive Presentation

Again, once automation can be used to derive data flow visualization components, we need to support interactive exploration, annotation, and update of these graphs. Here, we explore a proof-of-concept visualization to support interactive presentation of data flow graphs described in a json file.[1]

As with our other efforts, we asked expert analysts to provide feedback and suggestions during our iterative development. Feedback was generally positive; analysts frequently asked when they could have such a visualization integrated with their current tools. Unfortunately, as mentioned, we have not yet integrated this visualization into current workflows.

5.2.1 Proof-of-Concept Interactive Visualization

The primary goal in this instantiation has been to find a layout that can change as new graph elements are added to the display, but remains close to the original to alleviate cognitive load.

However, analysts from other domains are often drawing a visualization to encode their understanding – not looking at a visualization to gain insight. For example, Analyst Notebook [79] is essentially a drawing tool. Binary analysts often use their manual visualizations similarly; thus, the interactive visualization needed to give analysts a lot of control over the visualization elements and layout.

The first implementation used Dagre [81] for a somewhat force-directed layout and D3 [14] for the rendering. Unfortunately, some constraints imposed by Dagre, including an expectation of ranking that ordered nodes from the top down, did not interact well with our requirements. However, the pan and zoom abilities were helpful. In this implementation, the layout took up a lot of screen space even with a simple function. The layout did not look like it would scale well past a few sources and sinks.

The second iteration used webcola [40] and a tree-like view of the data flow.

The current iteration uses cytoscape [89]. The current default layout includes the following panes (shown in Figure 5.2):

- On the left, semi-transparent panes allow parts of the graph to be seen behind them, effectively increasing graph real-estate. These panes include
  - a function list of all the functions discovered in the json file
  - a hypothesis or *flowlist* pane, which allows an analyst to start, add or store, or clear a *hypothesis*, a selected list of nodes and their connecting edges, and
  - a details pane, showing the number of times that attribute has been viewed as a progress bar and including two tabs,
    - an attributes pane, showing the associated json attributes of a graph element and
    - a notes pane, where analysts can add and search textual notes about an element.
- In the middle, the graph pane takes most of the available screen space.

[1] We tested with json files created manually or through the dg extension.

- On the right, the code view supports selecting between source files and a json view.

This version does not currently implement the visual elements encoding types of nodes and edges; rather, it focuses on meeting the layout, update, and interactivity requirements.

Figures 5.2.1 and 5.2 show screenshots of this proof-of-concept, including an analyst interacting with some capabilities to support code exploration and analysis.

Several features in the proof-of-concept interactive visualization support bottom-up view construction:

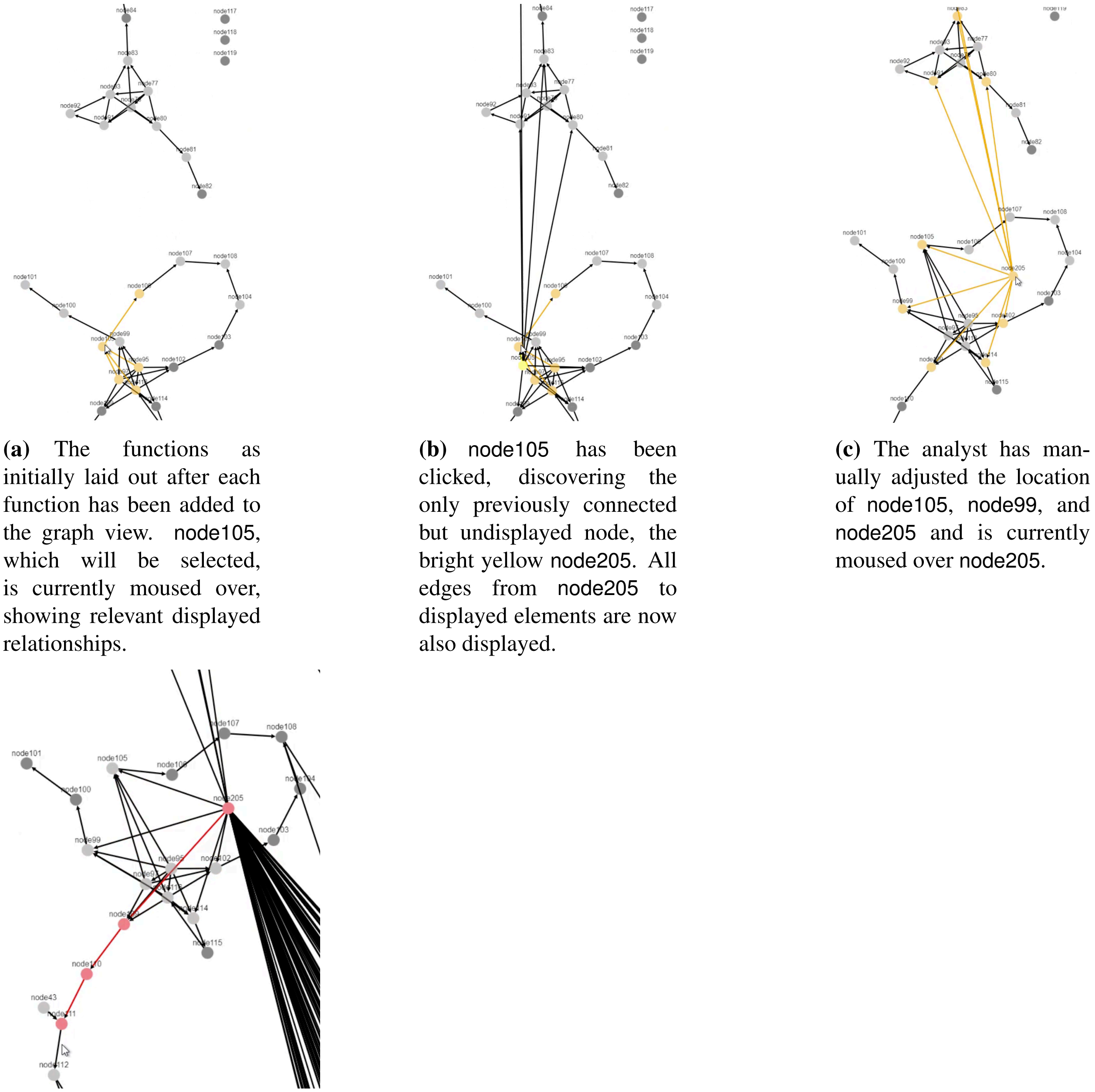
- On selecting a function from the function list, the view is populated with all other data flow nodes associated with that function, including location parents that may not be defined in the function (Figure 5.2(a)). The view supports visualizing multiple functions at a time.
- An analyst can click on a sequence of nodes and save the subgraph to a "hypothesis" list. Currently, nodes in a hypothesis are colored differently (in red). Mousing over or clicking a hypothesis highlights the subgraph in the current view, restores it to the view if hidden, or populates a secondary detail view with a small layout just of that path (Figure 5.2(d)).
- Given a selected path, the code pane shows raw code relevant to elements in the path (Figure 5.2). The visualization currently jumps to the relevant code line, if present, but a future iteration might highlight a selection, e.g., presenting a word tree. The code pane provides dynamic file loading and highlighting for most common languages.
- When a node is clicked, any connected nodes not already in the current view are added (i.e., the node is expanded; Figure 5.2(b)). In future work, nodes should also indicate if they have incoming or outgoing hidden links that outside the visible functions.
- Mousing over a node highlights all inbound and outbound links (Figure 5.2(a)).

Several features support analysis:

- Node locations can be controlled by the analyst through dragging (Figure 5.2(c)).
- Nodes and links can be annotated arbitrarily (Figure 5.2). The visualization supports easy location and highlighting of elements annotated with a suggested hashtag format, e.g. #priority:high, #uncertain, or #vulnerability. Currently the visualization changes the size of nodes annotated with #priority:high.
- Analysis histories for each subgraph are saved, and a meta-visualization of the graph analysis process can be shown. Subgraphs remember number of touches.
- Graph and analysis saving maintains positions across functions.

Finally, several future features have been requested by analysts or suggested by our analysis:

Figure 5.1. These figures show portions of the graph view as an analyst explores and interacts with two functions, cgc_clearBoard and cgc_reset, that do not directly share any nodes.

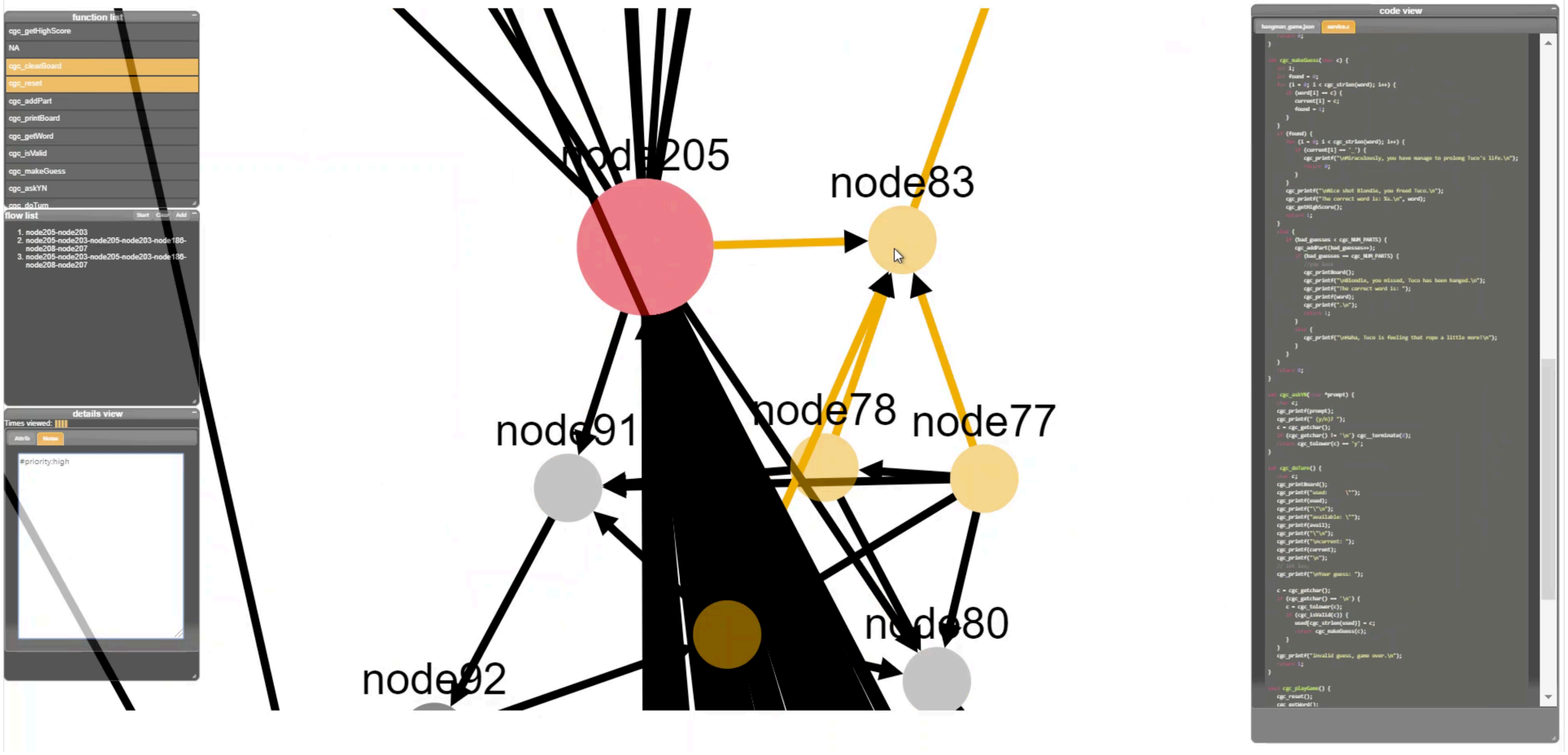


(a) The functions as initially laid out after each function has been added to the graph view. node105, which will be selected, is currently moused over, showing relevant displayed relationships.

(b) node105 has been clicked, discovering the only previously connected but undisplayed node, the bright yellow node205. All edges from node205 to displayed elements are now also displayed.

(c) The analyst has manually adjusted the location of node105, node99, and node205 and is currently moused over node205.

(d) The analyst has expanded node205 (discovering many outgoing links) and is in the process of defining a hypothesis from node205 to node109 to node110 to node111.



1. node205-node203
2. node205-node203-node205-node203-node186-node208-node207
3. node205-node203-node205-node203-node186-node208-node207

#priority:high

```
    return 0;
}

int cgc_makeGuess(char c) {
  int i;
  int found = 0;
  for (i = 0; i < cgc_strlen(word); i++) {
    if (word[i] == c) {
      current[i] = c;
      found = 1;
    }
  }
  if (found) {
    for (i = 0; i < cgc_strlen(word); i++) {
      if (current[i] == '_') {
        cgc_printf("\nMiraculously, you have manage to prolong Tuco's life.\n");
        return 0;
      }
    }
    cgc_printf("\nNice shot Blondie, you freed Tuco.\n");
    cgc_printf("The correct word is: %s.\n", word);
    cgc_getHighScore();
    return 1;
  }
  else {
    if (bad_guesses < cgc_NUM_PARTS) {
      cgc_addPart(bad_guesses++);
      if (bad_guesses == cgc_NUM_PARTS) {
        //you lose
        cgc_printBoard();
        cgc_printf("\nBlondie, you missed, Tuco has been hanged.\n");
        cgc_printf("The correct word is: ");
        cgc_printf(word);
        cgc_printf(".\n");
        return 1;
      }
      else {
        cgc_printf("\nHaha, Tuco is feeling that rope a little more!\n");
      }
    }
  }
  return 0;
}

int cgc_askYN(char *prompt) {
  char c;
  cgc_printf(prompt);
  cgc_printf(" (y/n)? ");
  c = cgc_getchar();
  if (cgc_getchar() != '\n') cgc__terminate(0);
  return cgc_tolower(c) == 'y';
}

int cgc_doTurn() {
  char c;
  cgc_printBoard();
  cgc_printf("used:      \"");
  cgc_printf(used);
  cgc_printf("\"\n");
  cgc_printf("available: \"");
  cgc_printf(avail);
  cgc_printf("\"\n");
  cgc_printf("\ncurrent: ");
  cgc_printf(current);
  cgc_printf("\n");
  // int len;
  cgc_printf("\nYour guess: ");

  c = cgc_getchar();
  if (cgc_getchar() == '\n') {
    c = cgc_tolower(c);
    if (cgc_isValid(c)) {
      used[cgc_strlen(used)] = c;
      return cgc_makeGuess(c);
    }
  }
  cgc_printf("Invalid guess, game over.\n");
  return 1;
}

void cgc_playGame() {
  cgc_reset();
  cgc_getWord();
```

Figure 5.2. Full screenshot of visualization environment, including all default panes and views and zoomed view of graph generated automatically from the TrailOfBits port of CGC challenge binary EAGLE_0005. In this screenshot, after all steps in Figure 5.2.1, an analyst has added the note #priority:high to node205, has saved three hypotheses that start at node205 (two are identical), and has moused over node83 to see related flows.

- Collapsing subgraphs that are contained within a hypothesis would allow analysts to control summarization within the main view.

- Analysts want the ability to move parents to new locations.

- A timeline of interaction (e.g., branching, review and restore, or breadcrumbs) would enable knowledge transfer.

- Analysts requested an ability to pass a hypothesis to an automated analysis engine, e.g., to look for more details about a structure, to hide irrelevant parts of a structure, or even to ask questions about values or constraints related to the hypothesis.

- As expected, analysts asked to be able to interact with a reverse engineering environment such as IDA Pro [55], and potentially to map graph elements back to both the binary in the RE environment and the source.

- Toggle-able layers based on node or link type were requested.

- Similarly, analysts should have the option to force node or edge information to be visible all the time, visible on hover, or never visible. These options could be tested in later instantiations: Can people extract meaning when details are not shown? Do they know where to look to find information rich areas? How much detail can be hidden?

- Interactivity to combine nodes or change clustering would allow analysts more control over their representations, which would be more like their current manual drawings, but this kind of interactivity would have to be built on top of the visualization engine. Currently the easiest (and expected) way to achieve this kind of interaction is to redefine the graph in json and allow the engine to create a new layout. This may lose the mapping to past layouts, though.

5.2.2 Interesting Layout Ideas for Data Flow Understanding

Several layouts may give interesting insights into data flow relationships and may be worth exploring:

Word Tree The word tree [111] shows how summary and global relationship information might be displayed. This layout allows forward and backwards traversal, puts code on the side, and supports easily hiding the relationship snippets.

Hive Plots and Pivot Graphs Layouts like hive plots [68] and pivot graphs might be used as an alternative view to help show when single nodes are particularly important sources or sinks and where the related data flows connect.

Scatter Plots Scatter plots from a force directed layout graph makes the location of the node meaningful. While such layouts can be ugly, they can also be interestingly informative.

5.3 Analyst Usage during Analysis

Recall, we expect that analysts will create this graph through discovery and analysis, use this graph for navigation, and take some working notes within the graph. The analyst will need to be able to use multiple other tools (e.g., IDA Pro, dynamic evaluation) and representations (e.g., the CFG, call graph) in conjunction with this graph. For effective use, this visualization will need to be integrated with other analyst tools. Ideally, navigation within one tool should be able to inform or control the views of other tools. This is a large engineering effort that remains to be addressed.

Chapter 6

Future Work

The application of human factors to binary reverse engineering and vulnerability analysis is growing. The breadth of potential research and impact is huge. We describe here a few of our ideas for future work at different scopes: locally to improve and evaluate our data flow visualization, and globally to explore binary analysis workflows.

6.1 Completing our Data Flow Visualization

Several thrusts remain to complete and complement our data flow visualization exploration. Further efforts should:

- Explore ways to encode additional information from "requirements" spreadsheet.
- Design solid tests and evaluate the utility of this representation for experts ...
- and compare to the utility of this representation for novices.
- Evaluate the visualization against other mission questions, including within the context of real analyses.
- Evaluate the visualization against our stated hard problems, including dynamic dispatch, pointer manipulations, and verifying and maintaining relationships between variables and values.
- Update automatic generation of json from LLVM to include other information from rule set.
- Implement automatic generation from a binary tool (e.g., IDA Pro) or front-end (e.g., McSema [33]).
- Update interactive visualization to include selective display based on annotations.
- **Tie analysis frameworks and visualization together in an interactive loop.**
- Expand this representation or, more likely, design a representation that helps analysts create a high-level understanding of data flow, i.e., an algorithmic description such as previously

designed hierarchical graphs designed after an analysis. Primitives or categories here would be goals, and tools would need to support creating this type of visualization over the course of the assessment, including categorizing uncertainty as it changes over the assessment and remembering critical details of analyst judgments.

- Extend a simple dynamic memory representation like kapsi [41], which color codes access types and displays them in an XY coordinate graph of memory accesses across time. We could layer on some useful bottom-up context information (i.e., linking parts of the graph to parts of the code) to help analysts quickly identify or confirm hypothesis-driven understanding of algorithms. Such a visualization could capture complex algorithmic behaviors (e.g., hashing, encryption), and provide the ability to compare dynamic memory information.

- Expand the IDA Pro navigation pane to include an ability to quickly categorize and tag code snippets (like a swipe right / swipe left interface). One question is: what judgment would be useful to tag data elements with? In the attack surface characterization cognitive walkthroughs, analysts were typically making judgments about the likely category of the data element to prioritize their future work.

6.2 Explorations of Binary Analysis Workflows

We believe that thinking of binary analysis as a classic sensemaking problem, and attempting to apply solutions to sensemaking problems in other domains, may prove fruitful. Similarly, exploring solutions and guidelines from the situational awareness (SA) literature may prove fruitful. Appendix G calls out specific SA design principles that are particularly relevant for binary analysis through symbolic execution.

We are particularly interested in research to meet the following goals:

- Understand how different workflows should be specialized for different questions.

- Efficiently communicate essential knowledge to yourself and others, with support for examining bases for conclusions and hypotheses.

- Support effective decision making, including helping analysts to answer the following questions:
  - When is the evidence for a particular conclusion/hypothesis strong enough to move on?
  - What portions of the analysis should I prioritize?
  - What tools might help me understand this question? And how do I use them?

- Observe analysts working in high-performing pairs or teams.
  - When do analysts elect to pair RE?
  - What does information hand-off look like in pair RE?

  - How is this cognitively advantageous? Do the analysts bring different sets of knowledge and skills to bear, whether general knowledge, specific vulnerability knowledge, code source knowledge, data flow knowledge, knowledge of programming paradigms, or tools to understand code?
  - How do the analysts interact? Do they attend to the same parts of code simultaneously? Pose questions for one another? Help to keep each other on track with the current goal?
  - How do tools and discussions reveal answers to questions and provide evidence for hypotheses? Does this make project hand off easier?

- Categorize design patterns of vulnerabilities and program features in binaries, similarly to the categorization of source code vulnerabilities described in [35].

- Create templates for data "audit logs", providing a format that allows an analyst to summarize valuable information and have that information automatically propagated through the relevant data flows in the binary.

- Allow analysts to explicitly represent and control uncertainty about hypotheses.

- Gather data from the RE environment to help determine what analysts are doing, where they need help, and where automated analyses could help to pare down the code to relevant sections.

- Explore applications of eye tracking to understanding expert and novice binary analysts (see Appendix I).

Previous human factors explorations of program understanding have identified cognitive design elements that are needed to support the construction of mental models. Storey and colleagues identified two broad classes of design elements important for helping software analysts maintaining code to build their mental model: those that support comprehension, and those that reduce the cognitive overhead of the analyst [101]. Examples of elements that support comprehension include tools and features that support the construction of multiple mental models, and tools and features that provide abstraction mechanisms. Such elements support both bottom-up (line-by-line analysis) and top-down (knowledge-driven) comprehension as well as the integration of the two. Examples of design elements that reduce the cognitive overhead of the analysts include support for navigation through the code, decision making, and documentation of findings. Although these insights came from studying software maintainers, they are relevant for binary reverse engineers as well. These insights will continue to be important as new tools are developed, automatic analyses are advanced, and reverse engineering workflows evolve.

We frequently observed the need to reduce the cognitive overhead of binary analysts. It was described as a pain point for analysts, e.g., forgetting decisions that were made about how to proceed with an assessment (such as which path to follow), and it was observed for all the analysts in the 1.5 hour cognitive walkthroughs (e.g., analysts described small memory lapses for where they had already looked in the code or for what they were looking). Analysts could recover from these memory lapses quite quickly, but their occurrence suggests that more support of analyst cognitive processing is required.

Besides further developing our colocations visualization (Appendix F), another opportunity for reducing the cognitive overhead of the analyst is to provide tools that can help them to record the details of their analysis. For example, something like a knowledge transfer diagram [116] could be particularly beneficial. These visualizations can help to externalize an analyst's understanding of both the program and the assessment. A record of this understanding can help maintain the current goal of the analysis, establishing the mental context that is required for analysis when returning to a project, or communicating the current state of understanding to other analysts or customers. Applied cognitive task analysis, activity analysis, time analysis, and work domain analysis could support development of these externalizations.

One of the most striking observations about binary analyst workflows is the mismatch in functional allocation. A central tenet of human factors for systems design is to assign functions of the system to the component, human or machine, that is best suited for performing that function. Binary code is the language that machines understand, and thus that function within the reverse engineering workflow should be assigned to the machine. Instead, it is the human that is trying to accurately focus and direct their attention to line-by-line processing while keeping multiple bits of information in memory to be available for future processing... in addition to remembering decisions and details related to the workflow itself. If mental simulation of program function cannot be avoided, highly automated systems for capturing the states of the system and the states of decision making across time could ease the cognitive burden of these analysts.

Changes to binary analyst workflows may increase human analyst throughput by orders of magnitude. However, significant domain mapping is required before appropriate workflow supports and changes can be effectively designed and implemented, particularly as analysis goals appear to significantly affect the course of an analysis.

Chapter 7

Related Work

Traditional static data flow analyses use unwieldy mathematical representations for computation [65]. Most visualizations of these analyses overlay data flow or other information onto a control abstraction, the CFG [55][83][117], the call graph [86][52], a file view [87] or a condensed view of the textual layout of the code [8][42].The former two sets of visualizations do not provide fine-grained interprocedural views; the latter set does not support interactive updates from the analyst (e.g., correcting the disassembly). Several past visualizations helped analysts filter, organize, and abstract overwhelming control flow graphs [78][102], delocalized data flow relationships [72][15], historical animated views [7] and hierarchical interactive views [80], and even hypothesis-driven understanding [84][69], but many of those visualization mechanisms do not appear to be implemented in the common reverse engineering platforms of today [55][3][12].

Visualizations of program dependence graphs (PDG) [112], annotated system dependence graphs (interprocedural PDGs) [31] and static value flow graphs [103] provide a reasonably intuitive view of many important data flow relationships, primarily among locations. However, these are statically computed graphs that are not designed to be updated, they are cognitively overwhelming, and they tend not to highlight important values. One visualization of a dynamic data flow graph uses physical address / execution time pairs with values [61], making most location and points-to relationships and some values easier to understand than in other representations. However, these dynamic representations cover *one* potential set of relationships associated with a single trace, and thus they do not generalize well to static analyses. Another recent work provides insight into values [58], but these visualizations support source code understanding around variables rather than locations. Such work complements our proposed requirements by exposing more information about value sets.

Decompilers such as HexRays [55] and Dream [114] provide the most intuitive advanced data flow representations today, encoding data flow information in automatically selected variable names. The Dream++ extension [113] even selects names to reduce cognitive load on analysts parsing the decompiled code. However, these text-based visualizations still use a control flow-based layout, encoding control flow depth using whitespace indentation just as in code development. They also display *all* the code rather than providing code folding [54], and analysts inject knowledge at a different layer of representation than that displayed (i.e., on the disassembly).

In contrast, we generated requirements that: engage human pattern recognition skills by representing information as a graph, support analysts updating or correcting the underlying information in the graph, and provide ways to filter, organize, and abstract these graphs.

Analysts interacting with instantiations of our requirements will be engaging in model building through visual analytics [4]. When implemented within a graph, such instantiations will need to support effective visual analytics through specialized interaction mechanisms and layouts such as those in [82][69][32][64][88][111][68]. However, these mechanisms are targeted to control flow abstractions or completely different domains. To support effective visual analytics, the data flow graph interaction and layout mechanisms will need to focus on providing easy correlation to the many other views that analysts use (including control flow views), reducing disorientation caused by changes in this and other views, and providing easy mechanisms to alter and annotate the graph.

Our work is heavily influenced by two individuals who have thought deeply about supporting visual analytics effectively: Storey [101] and Victor [110]. Storey provides a taxonomy of 14 cognitive design elements to support mental model construction during reverse engineering of source code for code maintenance (focusing on program understanding), and she points out the extensive background knowledge required by reverse engineers (including programming, languages, compilers, bugs and vulnerabilities, safe coding practices, and structures of different classes of programs such as state machines or parsers); we summarize some of her insights in Section 7.1. Victor argues for immediate feedback, particularly from tools supporting individuals who are engaging in a creative process (such as source code development, or, in our case, reverse engineering) [110]; easy movement between multiple levels of abstraction [109]; and natural interactive control mechanisms [108]. However, our work is focused in the more limited domain of answering data flow questions about a binary.

Groups considering the human as a part of the binary or vulnerability analysis system are growing in number. For example, the angr group is exploring ways to offload analysis tasks to non-expert [95]. The DARPA CHESS program is building research to support humans and computers working together to reason about the security of software artifacts [51]. Research groups such as [75] are exploring ways to allow users who are not experts in analysis algorithms to better control the analysis. Much (though not all) of this work is focused on building analytic systems to support more targeted allocation of work; in contrast, we focus on the externalization of human analysts' mental models.

While we use the Cyber Grand Challenge (CGC) binary challenge set for our tests [50][97], other data sets such as the Juliet test set [13] and LAVA test set [34] are also promising for simple test source code and binaries. While some studies move directly to realistically sized binaries, we currently focus on small binaries because we are manually generating most of our data flow graphs.

7.1 Insights from Storey's Studies

Storey studies reverse engineering for code maintenance. In [101], she calls out that reverse engineers require extensive background knowledge, including programming, languages, compilers, vulnerabilities, safe coding practices, and structures of different classes of programs (e.g., state machines, parsers). Citing other work, she describes how analysts build and maintain three types of mental models as they understand code:

- a *domain model*, describing the program's functionality, which analysts may begin the RE task with,
- a *program model*, specifying the control flow abstractions (and, likely, global data flow abstractions),
- and a *situation model*, describing data flow and functional abstractions around a particular situation, path, or set of paths. The situation models typically are not developed until after a partial program model has been found.

Storey also provides a taxonomy of cognitive design elements to support mental model construction during program understanding, describing 14 design elements in a figure. We restate the seven of those elements (in three groups) related to improving program comprehension, highlighting those that are particularly relevant for our work on data flow visualization

- Enhance bottom-up comprehension
  - Identify software objects and the relations between them
  - Reduce the disorientation that results from having to frequently switch between files and parts of code
  - **Build abstractions (e.g., through chunking) from lower-level units; tools should be available to allow the maintainer to create their own abstractions and label and document them to reflect their meaning**
- **Enhance top-down comprehension**
  - **Support goal-directed, hypothesis driven comprehension**
  - **Provide overviews of the system architecture at various levels of abstraction**
- **Integrate bottom-up and top-down comprehension**
  - **Support the construction of multiple mental models, particularly the situation and domain models**
  - **Cross-reference mental models**

Briefly we restate the seven elements (again, in three groups) focused on reducing the maintainer's cognitive overhead:

- Make it easier to find what you are looking for; i.e., facilitate navigation by providing directional navigation and supporting arbitrary navigation.
- Make it easier to know or recall where you are in the code and the analysis; i.e., provide orientation cues by indicating current focus, displaying the path that led to the current focus, and indicating options for further exploration.
- Reduce disorientation by reducing effort of user-interface adjustment and providing effective presentation styles.

Chapter 8

Conclusion

To reduce the human time burden for vulnerability analysts performing binary analysis requiring data flow understanding, we used human factors methods in a rolling discovery process to derive user-centric visual representation requirements. We derived requirements for interprocedural data flow visualizations that can be used to quickly understand data flow elements and their relationships and influences.

We encountered three main challenges: analysis projects span weeks, analysis goals significantly affect approaches and required knowledge, and analyst tools, techniques, conventions, and prioritization are based on personal preference. To address these challenges, we initially focused our human factors methods on an attack surface characterization task. We generalized our results using a two-stage modified sorting task, creating requirements for a data flow visualization.

We implemented our requirements partially in manually generated static visualizations of small binaries with a few vulnerabilities; the binaries were drawn from the CGC challenge binary set. We attempted to informally evaluate these manually generated graphs, finding that analysts were able to use the data flow visualizations to answer many critical questions about data flow. We also implemented these requirements partially in automatically generated interactive visualizations. These generation and visualization mechanisms have yet to be integrated into workflows for evaluation.

Our observations and results indicate that 1) this data flow visualization has the potential to enable novel code navigation, information presentation, and information sharing, and 2) it is an excellent time to pursue research applying human factors methods to binary analysis workflows. We are beginning to explore basic workflows, we have identified human factors findings from other domains that may be applicable, and we have identified some ways in which current human factors methods may need to be modified to apply to the variety of goal-driven reverse engineering workflows.

We are most excited by the level of engagement and positive reinforcement we received from our colleague analysts, both at Sandia and at other institutions. We look forward to continuing to apply human factors methods to binary software reverse engineering and vulnerability analysis, and we especially look forward to discovering workflow improvements that can slow our descent into the depths of analysis backlogs.

References

[1] Rafal Ablamowicz and Bertfried Fauser. Clifford: a maple 11 package for clifford algebra computations, version 11. http://math.tntech.edu/rafal/cliff11/index.html, 2007.

[2] Frances E Allen. Control flow analysis. In ACM Sigplan Notices, volume 5, pages 1–19. ACM, 1970.

[3] Sergi Àlvarez. The radare2 book. https://radare.gitbooks.io/radare2book/content/, 2009.

[4] Natalia Andrienko, Tim Lammarsch, Gennady Andrienko, Georg Fuchs, Daniel Keim, Silvia Miksch, and Andrea Rind. Viewing visual analytics as model building. In Computer Graphics Forum. Wiley Online Library, 2018.

[5] Christoph Aschwanden and Martha Crosby. Code scanning patterns in program comprehension. In Proceedings of the 39th hawaii international conference on system sciences, 2006.

[6] Lisanne Bainbridge. Ironies of automation. Automatica, 19:775–779, 1983.

[7] Marla J. Baker and Stephen G. Eick. Visualizing software systems. In Proceedings of the 16th International Conference on Software Engineering, ICSE '94, pages 59–67, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[8] Thomas Ball and Stephen G Eick. Visualizing program slices. In Visual Languages, 1994. Proceedings., IEEE Symposium on, pages 288–295. IEEE, 1994.

[9] Thomas Ball and Stephen G. Eick. Software visualization in the large. Computer, 29(4):33–43, April 1996.

[10] Roman Bednarik. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. International Journal of Human-Computer Studies, 70(2):143–155, 2012.

[11] Tanya Beelders and Jean-Pierre du Plessis. The influence of syntax highlighting on scanning and reading behaviour for source code. In Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, page 5. ACM, 2016.

[12] Vector 35 binary ninja product description page. https://binary.ninja.

[13] Tim Boland and Paul E Black. Juliet 1. 1 c/c++ and java test suite. Computer, 45(10):88–90, 2012.

[14] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. IEEE Transactions on Visualization & Computer Graphics, (12):2301–2309, 2011.

[15] K. Brade, M. Guzdial, M. Steckel, and E. Soloway. Whorf: a visualization tool for software maintenance. In Proceedings IEEE Workshop on Visual Languages, pages 148–154, Sept 1992.

[16] John Brooke et al. Sus-a quick and dirty usability scale. Usability evaluation in industry, 189(194):4–7, 1996.

[17] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In Program Comprehension (ICPC), 2015 IEEE 23rd International Conference on, pages 255–265. IEEE, 2015.

[18] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research, pages 1–9. ACM, 2011.

[19] Darpa cgc challenges source repository. `https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges`, 2016.

[20] Darpa cgc challenges ported to standard os. `https://github.com/trailofbits/cb-multios`, 2016.

[21] Marek Chalupa. Dg: Llvm dependencegraph and llvm-slicer. `https://github.com/mchalupa/dg`.

[22] Marek Chalupa. Slicing of llvm bitcode. Master's thesis, Masaryk University, Brno, Czech Republic, 2016.

[23] Marek Chalupa, Martina Vitovská, Martin Jonáš, Jiri Slaby, and Jan Strejček. Symbiotic 4: Beyond reachability. In Axel Legay and Tiziana Margaria, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 385–389, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[24] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. Acm Sigplan Notices, 46(3):265–278, 2011.

[26] Jean-François Collard and Jens Knoop. A comparative study of reaching-definitions analyses. 1998.

[27] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: interface aware fuzzing for kernel drivers. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2123–2138. ACM, 2017.

[28] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In 14th Workshop of the Psychology of Programming Interest Group, pages 58–73, 2002.

[29] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In International Conference on Software Engineering and Formal Methods, pages 233–247. Springer, 2012.

[30] Cwe-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html.

[31] F. Deng, N. DiGiuseppe, and J. A. Jones. Constellation visualization: Augmenting program dependence with dynamic information. In 2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pages 1–8, Sept 2011.

[32] Sabin Devkota and Katherine Isaacs. Cfgexplorer: Designing a visual control flow analytics system around basic program analysis operations. Computer Graphics Forum, 37:453–464, 06 2018.

[33] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In ReCon 2014 Conference, Montreal, Canada, 2014.

[34] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In Security and Privacy (SP), 2016 IEEE Symposium on, pages 110–121. IEEE, 2016.

[35] Mark Dowd, John McDonald, and Justin Schuh. The art of software security assessment: Identifying and preventing software vulnerabilities. Pearson Education, 2006.

[36] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In 2015 IEEE Security and Privacy Workshops, pages 73–87, May 2015.

[37] Andrew T Duchowski. A breadth-first survey of eye-tracking applications. Behavior Research Methods, Instruments, & Computers, 34(4):455–470, 2002.

[38] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In Francesco Logozzo and Manuel Fähndrich, editors, Static Analysis, pages 215–237, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[39] Halvar Dullien, Thomas / Flake. Machine learning, offence, and the future of automation, 2017.

[40] Tim Dwyer. cola.js: Constraint-based layout in the browser. `https://ialab.it.monash.edu/webcola/`.

[41] Otto Ebeling. Visualizing memory accesses of an executable. `https://bling.kapsi.fi/blog/x86-memory-access-visualization.html`, 2013.

[42] SC Eick, Joseph L Steffen, and Eric E Sumner. Seesoft-a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering, 18(11):957–968, 1992.

[43] Eldad Eliam and Elliot J Chikofsky. Reversing: secrets of reversing engineering, 2007.

[44] Mica Endsley, Betty Bolte, and Debra Jones. Designing for Situation Awareness: An Approach to User-Centered Design. 2014.

[45] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 9(3):319–349, 1987.

[46] Maintained source for file utility. `https://github.com/file/file`.

[47] Original source packages for file utility. `ftp://ftp.astron.com/pub/file/`, 2012.

[48] Kraig Finstad. The usability metric for user experience. Interacting with Computers, 22(5):323–327, 2010.

[49] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In Security and Privacy (SP), 2016 IEEE Symposium on, pages 377–396. IEEE, 2016.

[50] Dustin Fraze. Cyber grand challenge (cgc). `https://www.darpa.mil/program/cyber-grand-challeng`, 2016.

[51] Dustin Fraze. Computers and humans exploring software security (chess). `https://www.darpa.mil/program/computers-and-humans-exploring-software-security`, 2018.

[52] Zoe Hardisty. Radia github page. `https://github.com/zoebear/Radia`.

[53] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In Advances in psychology, volume 52, pages 139–183. Elsevier, 1988.

[54] T. Dean Hendrix, James H. Cross, II, Larry A. Barowski, and Karl S. Mathias. Visual support for incremental abstraction and refinement in ada 95. Ada Lett., XVIII(6):142–147, November 1998.

[55] SA Hex-Rays. Ida pro disassembler. `https://www.hex-rays.com/products/ida/`, 2008.

[56] SA Hex-Rays. Hex-rays decompiler, 2013.

[57] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 54–61. ACM, 2001.

[58] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. Augmenting code with in situ visualizations to aid program understanding. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, page 532. ACM, 2018.

[59] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs, volume 23. ACM, 1988.

[60] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In USENIX Security Symposium, pages 177–192, 2015.

[61] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In 24th USENIX Security Symposium (USENIX Security 15), pages 177–192, Washington, D.C., 2015. USENIX Association.

[62] A.G. Illera and F. Oca. Introducing ponce: one-click symbolic execution. `http://research.trust.salesforce.com/Introducing-Ponce-One-click-symbolic-execution/`.

[63] Kim J Vicente. Ecological interface design: Progress and challenges. 44:62–78, 02 2002.

[64] Bernhard Jenny, Daniel M Stephen, Ian Muehlenhaus, Brooke E Marston, Ritesh Sharma, Eugene Zhang, and Helen Jenny. Force-directed layout of origin-destination flow maps. International Journal of Geographical Information Science, 31(8):1521–1540, 2017.

[65] Gary A. Kildall. A unified approach to global program optimization. In Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[66] Fatih Kilic, Hannes Laner, and Claudia Eckert. Interactive function identification decreasing the effort of reverse engineering. In Revised Selected Papers of the 11th International Conference on Information Security and Cryptology - Volume 9589, Inscrypt 2015, pages 468–487, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[67] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In 29th International Conference on Software Engineering (ICSE'07), pages 489–498, May 2007.

[68] Martin Krzywinski, Inanc Birol, Steven JM Jones, and Marco A Marra. Hive plots—rational approach to visualizing networks. Briefings in bioinformatics, 13(5):627–644, 2011.

[69] Thomas D LaToza and Brad A Myers. Visualizing call graphs. In Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, pages 117–124. IEEE, 2011.

[70] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, page 75. IEEE Computer Society, 2004.

[71] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. Evaluation strategies for hci toolkit research. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, pages 36:1–36:17, New York, NY, USA, 2018. ACM.

[72] P. E. Livadas and S. D. Alden. A toolset for program understanding. In [1993] IEEE Second Workshop on Program Comprehension, pages 110–118, July 1993.

[73] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

[74] Pratyusa K. Manadhata, Kymie M. C. Tan, Roy A. Maxion, and Jeannette M. Wing. An approach to measuring a system's attack surface. School of Computer Science Technical Report CMU-CS-08-146, Carnegie Mellon University, Pittsburgh, PA, August 2007.

[75] Ravi Mangal, Xin Zhang, Aditya V. Nori, and Mayur Naik. A user-guided approach to program analysis. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pages 462–473, New York, NY, USA, 2015. ACM.

[76] Laura Militello and Robert Hutton. Applied cognitive task analysis (acta): A practitioner's toolkit for understanding cognitive task demands. 41:1618–41, 12 1998.

[77] Cve 2012-1571. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1571`, 2012.

[78] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In Proceedings of the 10th International Conference on Software Engineering, ICSE '88, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[79] Analyst Notebook. i2 analyst notebook i2 ltd.(2007), 2008.

[80] A. Orso, J. A. Jones, M. J. Harrold, and J. Stasko. Gammatella: visualization of program-execution data for deployed software. In Proceedings. 26th International Conference on Software Engineering, pages 699–700, May 2004.

[81] Chris Pettitt. Dagre: Directed graph layout for javascript. `https://github.com/dagrejs/dagre`.

[82] Dean W. Pucsek. Visualization and analysis of assembly code in an integrated comprehension environment. Master's thesis, University of Victoria, 2008.

[83] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In 2009 6th International Workshop on Visualization for Cyber Security, pages 27–32, Oct 2009.

[84] V. Rajlich, J. Doran, and R. T. S. Gudla. Layered explanations of software: a methodology for program comprehension. In Proceedings 1994 IEEE 3rd Workshop on Program Comprehension- WPC '94, pages 46–52, Nov 1994.

[85] Keith Rayner. Eye movements in reading and information processing: 20 years of research. Psychological bulletin, 124(3):372, 1998.

[86] Jörg Rech and Waldemar Schäfer. Visual support of software engineers during development and maintenance. volume 32, pages 1–3. ACM, 2007.

[87] Nishaanth H Reddy, Junghun Kim, Vijay Krishna Palepu, and James A Jones. Spider sense: Software-engineering, networked, system evaluation. In Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on, pages 205–209. IEEE, 2015.

[88] John Sarracino, Odaris Barrios-Arciga, Jasmine Zhu, Noah Marcus, Sorin Lerner, and Ben Wiedermann. User-guided synthesis of interactive diagrams. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pages 195–207. ACM, 2017.

[89] Paul T. Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. Genome research, 13 11:2498–504, 2003.

[90] Bonita Sharif, Michael Falcone, and Jonathan I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In Proceedings of the Symposium on Eye Tracking Research and Applications, ETRA '12, pages 381–384, New York, NY, USA, 2012. ACM.

[91] Bonita Sharif and Huzefa Kagdi. On the use of eye tracking in software traceability. In Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, pages 67–70. ACM, 2011.

[92] Katie Sherwin. Card sorting: uncover users' mental models for better information architecture. https://www.nngroup.com/articles/card-sorting-definition/.

[93] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[94] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[95] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. CoRR, abs/1708.02749, 2017.

[96] James Somers. The coming software apocalypse. https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/, September 2017.

[97] Jia Song and Jim Alves-Foss. The darpa cyber grand challenge: A competitor's perspective. IEEE Security & Privacy, 13(6):72–76, 2015.

[98] Bjarne Steensgaard. Points-to analysis in almost linear time. In Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 32–41. ACM, 1996.

[99] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph: A program representation for optimal program transformations. In European Symposium on Programming, pages 389–405. Springer, 1990.

[100] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[101] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. J. Syst. Softw., 44(3):171–185, January 1999.

[102] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In Proceedings of International Conference on Software Maintenance, pages 275–284, Oct 1995.

[103] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In Proceedings of the 25th International Conference on Compiler Construction, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[104] Cromu_00034 - diary parser. https://github.com/trailofbits/cb-multios/tree/master/challenges/Diary_Parser.

[105] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In Proceedings of the Symposium on Eye Tracking Research and Applications, pages 231–234. ACM, 2014.

[106] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In Proceedings of the 2006 symposium on Eye tracking research & applications, pages 133–140. ACM, 2006.

[107] David Van Horn and Matthew Might. Abstracting abstract machines. In Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.

[108] Bret Victor. A brief rant on the future of interaction design. http://worrydream.com, 2011.

[109] Bret Victor. The ladder of abstraction. `http://worrydream.com`, 2011.

[110] Bret Victor. Learnable programming. `http://worrydream.com`, 2012.

[111] Martin Wattenberg and Fernanda B Viégas. The word tree, an interactive visual concordance. IEEE Transactions on Visualization & Computer Graphics, (6):1221–1228, 2008.

[112] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Visualization of program dependence graphs. In Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction, CC'08/ETAPS'08, pages 193–196, Berlin, Heidelberg, 2008. Springer-Verlag.

[113] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In 2016 IEEE Symposium on Security and Privacy (SP), pages 158–177, May 2016.

[114] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In NDSS, 2015.

[115] Michal Zalewski. American fuzzy lop: a security-oriented fuzzer. `http://lcamtuf.coredump.cx/afl/(visitedon06/21/2017)`, 2010.

[116] J. Zhao, M. Glueck, P. Isenberg, F. Chevalier, and A. Khan. Supporting handoff in asynchronous collaborative sensemaking using knowledge-transfer graphs. IEEE Transactions on Visualization and Computer Graphics, 24(1):340–350, Jan 2018.

[117] Zynamics binnavi product description page. `https://www.zynamics.com/binnavi.html`.

Appendix A

Selecting a Data Flow Task to Study

To make effective use of human factors methods such as cognitive task analysis and cognitive walkthroughs, we needed to select a data flow task that can be completed in one to two hours. Unfortunately, currently defined static binary analysis tasks often take days to weeks to complete within the context of answering a mission question across weeks to months.

We also wanted this task to be fairly representative, allowing us to encounter analyst mental models that span a large portion of relevant data flow tasks. Again unfortunately, standard approaches to choosing a relevant task are not quite applicable in this domain yet. The higher-level mission question appears to affect data flow tasks rather dramatically. Further, analysts often describe tasks by their approaches, which differ across analysts. Many analysts are just beginning to explore disciplined ways to identify sub-goals and processes shared across mission questions.

Finally, we wanted to extract information from our tests with as little time commitment from our volunteer analysts as possible. Our test subjects were limited to a pool of about forty analysts whose job is to answer mission questions. Each test and interview we performed took them away from their missions in improving national security. We wanted to minimize the time we stole from each subject and refrain from burdening the same analysts multiple times.

To help us identify a reasonable data flow task to study, we explored two ways to define data flow tasks: by high-level goal, and by low-level relationship. A high-level goal-oriented task resonates with analysts, easily maps to approach-agnostic problems (allowing analysts to pursue their typical approaches), and creates well-defined and comparable output. A low-level relationship-oriented task constrains the questions about data flow to a specific set. Each high-level task involves several low-level tasks, and each low-level task may be used in several high-level tasks.

A.1 Goal-oriented Data Flow Task Definition

We define high-level data flow tasks by goals that might comprise part of a security assessment if a binary. Examples of high-level goal-oriented data flow tasks include:

Characterize an attack surface Where are areas of security concern within the system? Where does the attacker have influence on the boundary of the system? How might attacker-controlled input influence security sensitive code? What areas should be prioritized for further vulnerability analysis?

Reverse engineer an undocumented structure What data constraints hold over each field? How do the field values influence each other? What relationships exist between the fields, statically or dynamically? What is the semantic meaning of each (relevant) field?

Reverse engineer an undocumented protocol Where is data produced or consumed on the program boundary? How are messages parsed and created? What are the phases of communication? What induces a transition to a new phase?

Characterize potential impact of a vulnerability Given a vulnerability, is there an attacker-controlled input that could exercise that vulnerability? What paths are vulnerable? Along those paths, how does system code influence or add constraints on data values and relationships? How much control could an attacker gain? Could that vulnerability be used to affect security sensitive code?

Mitigate a dataflow exploit Given a dataflow exploit, how does the data in the exploit control the system code? What assumptions within the software allow this exploit to function? What code or system changes would prevent it?

A.2 Task-oriented Data Flow Task Definition

We define low-level data flow tasks by specific relationships between data that an analyst wants to discover. Examples of low-level relationship-oriented data flow tasks include:

Slice forward Given a source, where are the sinks?

Slice backward Given a sink, where are the sources?

Resolve dynamic dispatch Given a call site, where can control go? What data influences that decision and how?

Infer dynamic value correlation Given some data, is other data implicitly synchronized? For a simple example, a buffer length should represent the number of bytes actually allocated in memory for the corresponding buffer.

Trace an input Given an initial state and an input, how does the state change? Where does control go and why?

Propagate constraints (along a set of paths) Given a set of paths, how are data values restricted? What restricts them?

Infer aggregates Given some data, what other data is co-located and should be abstracted with that data?

Infer points-to information Given a pointer, where does it point?

Infer heap object shape Given a pointer, what kind of data structures does it point to?

Infer type flow Given some typed data at a program point, is other data restricted in type?

A.3 Data Flow Task Selection

For our first interviews and cognitive walkthroughs, we decided to focus on attack surface characterization. Attack surface characterization is well-understood; analysts know where and how to address such a goal. The problem itself and desired outputs are easily defined, and the task can be time-limited. Analysts can focus on forward slicing, minimal constraint propagation, and prioritization judgments, although other low-level tasks are also brought to bear.

In contrast, reverse engineering structures and protocols tends to be extremely time intensive. Time-limited analyses produce partial structure or protocol definitions, and such partial output is difficult to compare across analysts given relatively small study sizes. Data flow exploitation is relatively new in the academic literature, difficult to reason about, and not yet well-understood.

We attempted to define a second set of cognitive walkthroughs (Section B) to help us generalize the results of our attack surface characterization studies (Section 2.4). For this phase, we decided to focus on characterizing the impact of a vulnerability. In a carefully defined task, analysts can focus on constraint propagation, backwards slicing, and mental tracing. Unfortunately, our dry-run indicated that a full test would require significantly more care in task definition. Vulnerability impact characterization requires either a significant time investment or an effective mechanism to transfer knowledge from earlier phases of binary exploration. Lacking these, we did not pursue this method for generalizing our results further.

We did not attempt to incorporate many of the other low-level tasks mentioned for similar reasons. Tasks resolving dynamic dispatch, inferring dynamic value correlation, and discovering aggregates tend to be either trivial or extremely complex. Finding appropriately complex (non-trivial, time-limited) examples of such tasks would require significant time, and we chose to ignore these tasks. Pointer analysis, shape analysis, and type flow analysis are areas of active research in program analysis; they are generally computed via fixed-point analyses that use global knowledge about the binary. We specifically chose to exclude such tasks from our research.

Appendix B

Designing a Cognitive Walkthrough: Analyzing the Impact of a Vulnerability

We briefly explored creating cognitive walkthroughs for a different task, assessing the impact of a vulnerability, as one possible way to generalize our data flow elements derived for the attack surface characterization task. As with other use cases, we had to overcome several challenges:

- Analyses unfold over days.
- Binaries are originally written in a variety of programming languages.
- Analysts use a variety of analysis environments.
- Analysts ingrate multiple sources of information.

We designed and performed a pre-test with one analyst. However, our pre-test indicated that significantly more work would be needed to design an effective cognitive walkthrough. Specifically, to fit within our one- to two-hour timeframe for each session, we would need to provide a few hours beforehand to allow the analysts to familiarize themselves with the code, or we would need to figure out how to transfer knowledge about the binary effectively from earlier phases of binary exploration. Lacking these, we did not pursue this method for generalizing our results further.

B.1 Experimental Setup

We selected the UNIX file utility version 5.10 [47][46] for analysis, as previously. This version of file's core processing library libmagic is vulnerable to CVE-2012-1571 [77]. We had access to source code for both the vulnerable version 5.10 and the fixed version 5.11. We built version 5.10 as a 32-bit executable with symbols.

We asked analysts to begin analysis again at the file_buffer function in libmagic, treating the array argument and length as attacker-controlled, i.e., as the "inputs" for the exercise. This is a legitimate simplifying assumption in a real analysis; most of the processing before this point involves command line parsing and reading the input file into a buffer in memory. Our human factors specialist took notes about task performance and asked for additional details to understand the thought process of the analyst, including asking for reasoning behind judgments and decisions, and asking for clarification about sources of frustration. Walkthroughs lasted two hours including

Today I am going to ask you to describe the steps you take when you are analyzing a binary to understand how and whether the code has sufficient protections to prevent exploitation. This exercise is not intended to be a full vulnerability analysis. I am asking you to focus on understanding how input data are processed and transformed through a limited portion of the binary. I want to understand how you make sense of the data inputs for this assessment: 1) What goals do you have when you are understanding the code? 2) When do you change goals and why? 3) What knowledge and information from the binary are you using to make these decisions at each stage? I will not be evaluating your work, nor should you have to describe details about any similar projects that you are working on.

To guide this description, I will provide you with a binary to analyze and specify what portion of the code to assess. I will ask you to describe aloud how you are assessing the binary, why you take certain actions in assessing the binary, what cues or information you use that lead you to take certain actions, and how you are categorizing different parts of the binary. I may interrupt with questions to help you explain and think aloud. If such interruptions drastically disrupt your work, please let me know and we can arrange for you to complete the analysis exercise first and then talk through your analysis afterward.

By "assessing the protections", I mean determining what aspects of inputs reach locations that are security sensitive and how well the binary protects those locations. For this binary, please:

- Assess the utility file binary for the impact of vulnerability CVE-2012-1571. We provide you with a binary of version 5.10. This vulnerability is patched in version 5.11. You may access the source code for both versions if desired.
- Evaluate the path between the input data at function/interface file_buffer. You may explore anywhere in the binary to gain context, but your goal is to assess the protections in the binary for preventing exploitation.
- Produce an algorithmic description of the input and how it flows to the security sensitive area, conveying the critical protections, data transformations, constraints and mitigations that are in place and may protect the security of the system.
- Characterize these input data security elements (e.g., functions or program points) – and the paths that reach them – with respect to the inputs and any other data sources
- Keep notes, comments, and diagrams as though you were going to complete a full vulnerability analysis.
    - Please take any written notes in our provided notebook
    - Please leave us with a copy of any electronic artifacts created or modified during this activity

Figure B.1. Instructions provided to analysts for vulnerability impact cognitive walkthrough.

the time to set up the analysis environment.

We provide our instructions to analysts and questions for analysts in Figures B.1 and B.2. The questions are almost identical to those in the attack surface characterization cognitive walkthroughs

B.2 Observations

To guide the analysts to the correct portion of code, we needed to provide an initial input document for file to parse. For this pre-test, we provided a CDF file that ended up being the wrong type. The analyst discovered the error quickly and created a "composite document file v2" Microsoft .doc file, but this simple file did not drive the execution to the appropriate section of code. While such proof-of-vulnerability inputs may be provided with CVEs, a problematic input was not available in this case. We would need to construct one to support analysts within the timeframe desired.

Background Questions

Before beginning the exercise, we ask each analyst the following questions:

1) For how long have you been doing work that required analysis of binary code?
2) What types of security-related binary analysis do you do? And for how long have you done each?

Questions to Ask during Analysis

We interrupt each analyst as needed with the following questions:

1) What are you trying to do or learn right now?
2) What did you need to do to learn about that code?
3) What are you thinking about right now?
4) In the section of code that you are working on right now, what are you thinking about?
5) Why did you jump to this section of code?
6) Why did you decide to name this?
7) What does the name mean?
8) Why did you decide you should make a comment here?
9) What does the comment mean?
10) Where are you looking on the screen? Why?
11) In the section of code that you are working on right now, what information are you thinking about?
12) Does the variable or data that you are currently analyzing relate to other analysis that you have already done? How?

Figure B.2. Questions we ask analysts before and during our vulnerability impact cognitive walkthrough.

We provided the description of the vulnerability from the CVE database, but this description was not sufficient. The analyst had to remind himself of the vulnerability, even though he had been exposed to it within the past two weeks. Other analysts taking the test without the same background would take even longer to familiarize themselves with the binary. We would need to construct a good description of the vulnerability to save analyst time.

Finally, we would need to provide a work environment appropriate to each analyst. Our task allowed analysts to look at both the vulnerable and the patched version of file, so analysts would need their preferred source auditing platforms. Our task allowed analysts to trace the binary, so analysts would need their preferred dynamic tracing environment set up to work with our provided binary.

Because our pre-test analyst was unable to construct and evaluate a problematic test case in the timeframe allowed, our pre-test indicated that, without further development, analysts would be focusing on the same data flow tasks used for attack surface characterization. They would not have the time to focus instead on the later tasks, backwards tracing and constraint propagation. Rather than focusing on eliminating or minimizing the obstacles encountered in this pre-test, we shifted gears and focused on generalizing our data flow primitives via analysis of analyst artifacts produced for past projects2.5.

Appendix C

Designing an A/B Comparison

We briefly explored an A/B comparison for experienced binary reverse engineers between our static data flow visualization and analysts working with a stripped version of the binary. We describe this A/B comparison pre-test here for completeness, but we decided that A/B experimental testing was premature because the visualization was not deployed within the analysis environment and only represented a subset of the information needed for a full vulnerability assessment.

C.1 Experimental Setup

We selected the TrailOfBits provided port of the EAGLE_0005 binary, named CGC_Hangman_Game, compiled in 32-bit and stripped of symbols. We felt a stripped binary was most appropriate because 1) many third-party binaries and malware are analyzed without symbols, and 2) our static data flow visualization did not provide most symbol names. However, since our static visualization did provide some symbol names as reference points, we added those specific names back into the stripped binary before the analysis.

We asked two experienced binary analysts to analyze the binary using their standard RE environment and to answer the same questions as the analyst using the visualization (Section 4.2), though we only kept detailed notes about the second. These questions were annotated with specific binary offsets to enable the analysts to locate the features of interest quickly. While only the primary questions 1-8 changed, we provide the entire question set here in Figure C.1 for reference.

C.2 Observations

The second analyst spent 55 minutes analyzing the binary. Questions 1 and 6 took the most time to answer, using 8 and 15 minutes, respectively. In both cases, the analyst was exploring new parts of the binary and commented that the exploration should have proceeded faster, especially with more familiarity with the code. The analyst missed partial information in answering questions 3, 7, and 14, and failed to document discoveries completely in answering questions 5 and 7. In these cases, each question asked for a list of information and did not provide the analyst with any information as to the correct number of items in each list; the analyst provided answers but did not provide

1) Where does an attacker control data input throughout the program?
2) What constraints on the password read into inbuf would allow the body of this program to be reached?
3) Looking at the processing of the input buffer inbuf in the function main:
    a) What is the initial value of the pointer?
    b) When is the pointer incremented?
    c) Are values being read or written as the array is walked?
    d) What values are being looked for as the array is walked?
    e) What values are written?
    f) When are those values written?
4) Looking at the global avail array accessed at 0x0804c6f7:
    a) Are values being read or written as the array is walked?
    b) What values are written?
    c) When are those values written?
    d) When else is avail read?
    e) When else is avail written?
5) Looking at the global used array accessed at 0x0804c701:
    a) Are values being read or written as the array is walked?
    b) What values are written?
    c) When are those values written?
    d) When else is used read?
    e) When else is used written?
6) Looking at the global word array:
    a) Where is the array being walked?
    b) For each use, are values being read or written as the array is walked?
    c) For each read, what values are being looked for?
    d) For each write, what values are being written and where do they come from?
    e) How were those source values located?
        i) Follow the flow backwards as far as possible.
        ii) What causes these values to change?
        iii) Are there potential problems with this flow?
7) Looking at the variable written at 0x08004cd25:
    a) What are inputs to the value?
    b) What constraints are on the value?
    c) To what locations is c written? When?
    d) To what locations is c compared? When?
8) Looking at the name array accessed by the function at 0x0804c8e9:
    a) What are inputs to the values?
    b) What are constraints on the data input to this variable?
    c) Are there potential problems with this flow?
    d) Similarly, following name through STDOUT?
9) Can you find a place where an array is walked through by incrementing a pointer?
10) Can you find a place where an array is walked through by incrementing a counter and adding that to the base pointer?
11) Can you find where the global array gallows is written?
12) Can you find where the global array current is written?
13) How many guesses do you have before you lose the game?
14) What things cause the game to restart?

Figure C.1. Evaluation questions for binary analyst pre-test of EAGLE_0005.

all possible items. The analyst did not provide any incorrect answers, nor did the analyst fail to answer any questions.

These results indicate that the visualization alone may be comparable to the standard binary analysis process. Without integration into a workflow, however, we do not feel that a full A/B comparison will give us informative results about the strengths and weaknesses of this visual approach.

Appendix D

Comparative Visualizations of Small Cyber Grand Challenge (CGC) Binaries

We provide a few current visualizations of the CGC CROMU_00034 and EAGLE_0005 binaries for comparison.

First, we provide compare visualizations of CROMU_00034 from Figure 3.8. Figure D.1 shows a call graph created by a tool similar to the Cartographer module [82], allowing interactive discovery of call nodes, aggregation of nodes, and a summary of all analyst annotations found within the function. Here, library functions are hidden and only the application functions are shown. Figure D shows the instruction level graph for only those functions and the interprocedural CFG for all functions in the binary, both as displayed by the Gephi graph visualization tool, version 0.9.2. While analysts would not use this tool to display such graphs (additional information within each node is relevant and specialized layouts help identify interesting code), these images serve to show the overwhelming number of nodes in even this extremely simplistic example.

Finally in Figure D, to give an example of the data provided by a system dependence graph (SDG, an interprocedural PDG), we compare a visualization of the PDG for the 33 lines of the main function from EAGLE_0005 from Figure 4.3 to our graph of the full binary (the highlighted portions are those that are relevant to the main function). This PDG was produced by Frama-C

Figure D.1. Call graph representation of analyst discovery of vulnerability 2 in the DARPA CGC challenge CROMU_00034.

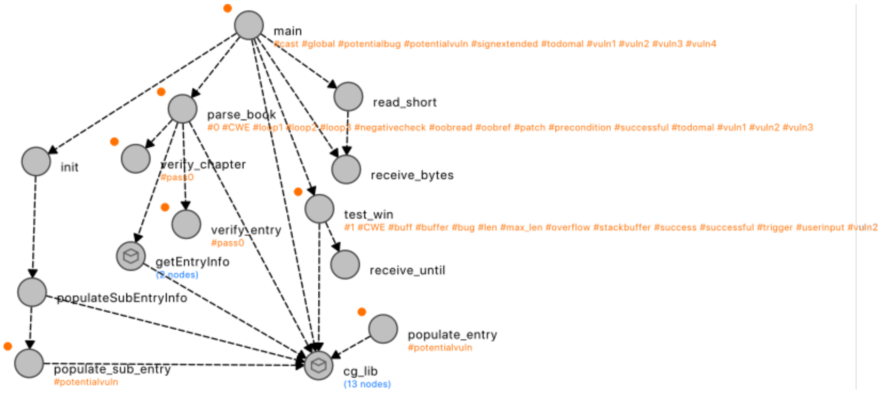
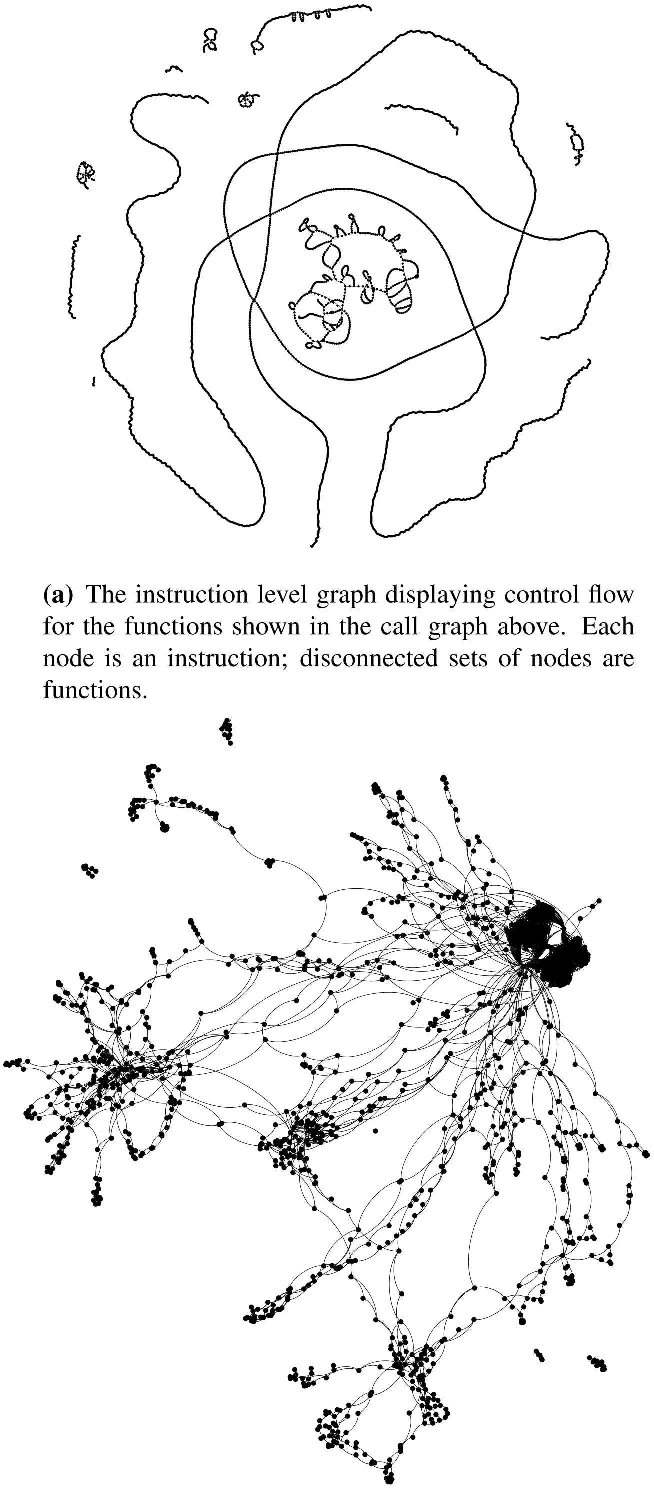

main
#cast #global #potentialbug #potentialvuln #signextended #todomal #vuln1 #vuln2 #vuln3 #vuln4
read_short
parse_book
#0 #CWE #loop1 #loop2 #loop3 #negativecheck #oobread #oobref #patch #precondition #successful #todomal #vuln1 #vuln2 #vuln3
init
verify_chapter
#pass0
receive_bytes
test_win
#1 #CWE #buff #buffer #bug #len #max_len #overflow #stackbuffer #success #successful #trigger #userinput #vuln2
verify_entry
#pass0
getEntryInfo
(2 nodes)
receive_until
populateSubEntryInfo
populate_entry
#potentialvuln
populate_sub_entry
#potentialvuln
cg_lib
(13 nodes)

Figure D.2. These figures show control flow abstractions of DARPA CGC challenge CROMU_00034 as displayed by Gephi v.0.9.2.

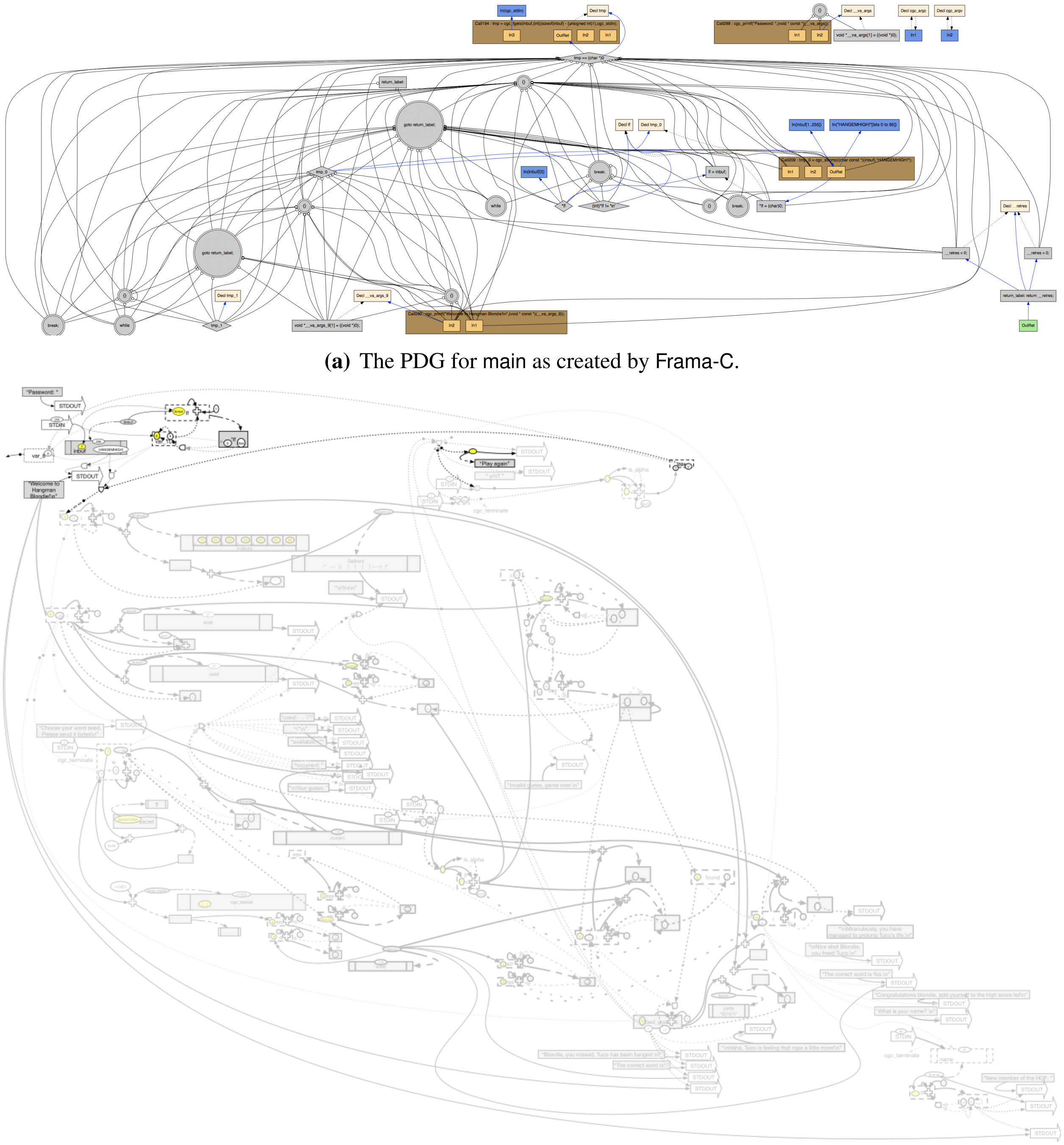


(a) The instruction level graph displaying control flow for the functions shown in the call graph above. Each node is an instruction; disconnected sets of nodes are functions.

(b) The interprocedural control flow graph (CFG) for all functions, including library functions.

[29], though we were unable to produce the full SDG quickly. (For a reason we have not tracked down, the abstract interpretation hit bottom early in the next called function, precluding further PDG generation or full SDG generation.)

Figure D.3. These figures compare a program dependence graph (PDG) for the main function of EAGLE_0005 to our graph of the full binary.


In(cgc_stdin)
Decl tmp
Call194 : tmp = cgc_fgets(inbuf,(int)(sizeof(inbuf) - (unsigned int)1),cgc_stdin);
In3
OutRet
In2
In1
{}
Decl __va_args
Call288 : cgc_printf("Password: ",(void * const *)(__va_args));
void *__va_args[1] = {(void *)0};
Decl cgc_argc
Decl cgc_argv
tmp == (char *)0
return_label:
goto return_label;
Decl lf
Decl tmp_0
In(inbuf[1..255])
In("HANGEMHIGH!"[bits 0 to 95])
Call209 : tmp_0 = cgc_strcmp((char const *)(inbuf),"HANGEMHIGH!");
tmp_0
In(inbuf[0])
break;
lf = inbuf;
while
*lf
(int)*lf != '\n'
*lf = (char)0;
Decl __retres
__retres = 0;
return_label: return __retres;
Decl tmp_1
Decl __va_args_9
tmp_1
void *__va_args_9[1] = {(void *)0};
Call290 : cgc_printf("Welcome to Hangman Blondie!\n",(void * const *)(__va_args_9));
"Password: "
STDOUT
STDIN
inbuf
HANGEMHIGH!
var_8
var_10
lf
"Welcome to Hangman Blondie!\n"
"Play again"
" y/n? "
is_alpha
cgc_terminate
indexes
Gallows
"\n%s\n"
avail
used
"Choose your word seed. Please send 4 bytes\n"
"Invalid guess, game over.\n"
correct
cgc_words
found
bad_guess
parts
"\nMiraculously, you have managed to prolong Tuco's life.\n"
"\nNice shot Blondie, you freed Tuco.\n"
"The correct word is %s.\n"
"Congratulations blondie, add yourself to the high score list!\n"
"What is your name? \n"
"\nHaha, Tuco is feeling that rope a little more!\n"
"Blondie, you missed, Tuco has been hanged.\n"
"The correct word is: "
name
"New member of the HOF: "

(a) The PDG for main as created by Frama-C.

(b) Our data flow graph of the full EAGLE_0005 binary with the components relevant to main highlighted.

Appendix E

Example json Specification of sums Data Flow Graph.

```
{
  "nodes": [
    {
      "analyst-judgments": {
        "influence": "high",
        "uncertainty": "zero",
        "safety": "low",
        "prioritization": "low"
      },
      "entity-type": "node",
      "children": [
        "node207"
      ],
      "id": "node194",
      "artifact-location": {
        "source-file": {
          "line": 1,
          "char-offset": 5
        }
      },
      "properties": {
        "name": "n",
        "source-of-data": "input",
        "data-type": "int",
        "frequency-of-use": {
          "constant": "1"
        },
        "type": {
          "primary": "location",
          "secondary": "local"
        },
        "size": 4
      }
    },
    {
      "properties": {
        "type": {
          "primary": "value",
          "secondary": "unknown"
        },
        "name": "?"
      },
      "id": "node207",
      "entity-type": "node"
    },
    {
      "analyst-judgments": {
        "influence": "high",
        "uncertainty": "zero",
        "safety": "low",
        "prioritization": "low"
      },
      "entity-type": "node",
      "children": [
        "node196",
        "node197",
        "node198"
      ],
      "id": "node195",
      "artifact-location": {
        "source-file": {
          "line": 2,
          "char-offset": 17
        }
      },
      "properties": {
        "name": "i",
        "source-of-data": "internally-generated-evaluation",
        "data-type": "int",
        "frequency-of-use": {
          "upper-bound": "n",
          "count": "n",
          "lower-bound": "1"
        },
        "type": {
          "primary": "location",
          "secondary": "local"
        },
        "size": 4
      }
    },
    {
      "properties": {
        "initial-value": true,
        "type": {
          "primary": "value",
          "secondary": "constant"
        },
        "name": "1"
      },
      "artifact-location": {
        "source-file": {
          "line": 2,
          "char-offset": 4
        }
      },
      "id": "node196",
      "entity-type": "node"
    },
    {
      "properties": {
        "role": "loop",
        "type": {
          "primary": "value-flow",
          "secondary": "set-constraint"
        },
        "name": "<=n"
      },
      "artifact-location": {
        "source-file": {
          "line": 4,
          "char-offset": 6
        }
      },
      "id": "node197",
      "entity-type": "node"
    },
    {
      "analyst-judgments": {
        "influence": "low",
        "uncertainty": "zero",
        "safety": "low",
        "prioritization": "low"
      },
      "properties": {
        "source-of-data": "internally-generated-evaluation",
        "type": {
          "primary": "value",
          "secondary": "computed"
        },
        "frequency-of-use": {
          "upper-bound": "n",
          "count": "n",
          "lower-bound": "1"
        }
      },
      "artifact-location": {
        "source-file": {
          "line": 11,
          "char-offset": 3
        }
      },
      "id": "node198",
      "entity-type": "node"
    },
    {
      "analyst-judgments": {
        "influence": "low",
        "uncertainty": "zero",
        "safety": "low",
        "prioritization": "low"
      },
      "entity-type": "node",
      "children": [
        "node200",
        "node201",
        "node202"
      ],
      "id": "node199",
      "artifact-location": {
        "source-file": {
          "line": 6,
          "char-offset": 1
        }
      },
      "properties": {
        "data-type": "int",
        "type": {
          "primary": "location",
          "secondary": "local"
        },
        "name": "j",
        "frequency-of-use": {
          "upper-bound": "n log n",
          "count": "n log n",
          "lower-bound": "1"
        },
        "size": 4
      }
    },
    {
      "properties": {
        "type": {
          "primary": "value",
          "secondary": "constant"
        },
        "name": "1",
        "initial-value": true
      },
      "artifact-location": {
        "source-file": {
          "line": 6,
        "char-offset": 5
      }
    },
    "id": "node200",
    "entity-type": "node"
  },
  {
    "properties": {
      "role": "loop",
      "type": {
        "primary": "value-flow",
        "secondary": "set-constraint"
      },
      "name": "<=i"
    },
    "artifact-location": {
      "source-file": {
        "line": 7,
        "char-offset": 7
      }
    },
    "id": "node201",
    "entity-type": "node"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "properties": {
      "source-of-data": "internally-
            generated-evaluation",
      "type": {
        "primary": "value",
        "secondary": "computed"
      },
      "frequency-of-use": {
        "upper-bound": "n log n",
        "count": "n log n",
        "lower-bound": "1"
      }
    },
    "artifact-location": {
      "source-file": {
        "line": 9,
        "char-offset": 3
      }
    },
    "id": "node202",
    "entity-type": "node"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "node",
    "children": [
      "node204",
      "node205"
    ],
    "id": "node203",
    "artifact-location": {
      "source-file": {
        "line": 3,
        "char-offset": 1
      }
    },
    "properties": {
      "name": "sum",
      "source-of-data": "internally-
            generated-evaluation",
      "data-type": "int",
      "frequency-of-use": {
        "upper-bound": "n log n",
        "count": "n log n",
        "lower-bound": "1"
      },
      "type": {
        "primary": "location",
        "secondary": "local"
      },
      "size": 4
    }
  },
  {
    "properties": {
      "type": {
        "primary": "value",
        "secondary": "constant"
      },
      "name": "0",
      "initial-value": true
    },
    "artifact-location": {
      "source-file": {
        "line": 3,
        "char-offset": 4
      }
    },
    "id": "node204",
    "entity-type": "node"
  },
  {
    "properties": {
      "source-of-data": "internally-
            generated-evaluation",
      "type": {
        "primary": "value",
        "secondary": "computed"
      },
      "frequency-of-use": {
        "upper-bound": "n log n",
        "count": "n log n",
        "lower-bound": "1"
      }
    },
    "artifact-location": {
      "source-file": {
        "line": 8,
        "char-offset": 5
      }
    },
    "id": "node205",
    "entity-type": "node"
  },
  {
    "properties": {
      "type": {
        "primary": "value",
        "secondary": "constant"
      },
      "name": "1"
    },
    "id": "node206",
    "entity-type": "node"
  },
  {
    "properties": {
      "type": {
        "primary": "code"
      },
      "name": "read"
    },
    "id": "node208",
    "entity-type": "node"
  },
  {
    "properties": {
      "type": {
        "primary": "write"
      },
      "name": "write"
    },
    "id": "node209",
    "entity-type": "node"
  }
],
"links": [
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge195",
    "source": "node197",
    "artifact-location": {
      "source-file": {
        "line": 4,
        "char-offset": 2
      }
    },
    "properties": {
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node200"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge196",
    "source": "node197",
    "artifact-location": {
      "source-file": {
        "line": 4,
        "char-offset": 2
      }
    },
    "properties": {
      "layers": [
        "influence",
        "sequencing"
      ],
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node204"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge197",
    "source": "node197",
    "artifact-location": {
      "source-file": {
        "line": 4,
        "char-offset": 2
      }
    },
    "properties": {
      "layers": [
        "influence",
        "sequencing"
      ],
      "role": "loop",
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node197"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge198",
    "source": "node201",
    "artifact-location": {
      "source-file": {
        "line": 7,
        "char-offset": 2
      }
    },
    "properties": {
      "layers": [
        "influence",
        "sequencing"
      ],
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node205"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge199",
    "source": "node201",
    "artifact-location": {
      "source-file": {
        "line": 7,
        "char-offset": 3
      }
    },
    "properties": {
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node202"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge200",
    "source": "node201",
    "artifact-location": {
      "source-file": {
        "line": 7,
        "char-offset": 3
      }
    },
    "properties": {
      "layers": [
        "influence",
        "sequencing"
      ],
      "role": "loop",
      "type": {
        "primary": "control-flow-
              enabled"
      }
    },
    "target": "node201"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge201",
    "source": "node201",
    "artifact-location": {
      "source-file": {
        "line": 7,
        "char-offset": 3
      }
    },
    "properties": {
      "type": {
        "primary": "control-flow-
              disabled"
      }
    },
    "target": "node195"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge202",
    "source": "node195",
    "artifact-location": {
      "source-file": {
        "line": 11,
        "char-offset": 2
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node198"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge203",
    "source": "node199",
    "artifact-location": {
      "source-file": {
        "line": 8,
        "char-offset": 3
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node205"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge204",
    "source": "node203",
    "artifact-location": {
      "source-file": {
        "line": 8,
        "char-offset": 3
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node205"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge205",
    "source": "node199",
    "artifact-location": {
      "source-file": {
        "line": 9,
        "char-offset": 3
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node202"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge206",
    "source": "node206",
    "artifact-location": {
      "source-file": {
        "line": 9,
        "char-offset": 4
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node198"
  },
  {
    "analyst-judgments": {
      "influence": "low",
      "uncertainty": "zero",
      "safety": "low",
      "prioritization": "low"
    },
    "entity-type": "edge",
    "id": "edge207",
    "source": "node206",
    "artifact-location": {
      "source-file": {
        "line": 11,
        "char-offset": 4
      }
    },
    "properties": {
      "type": {
        "primary": "value-flow",
        "secondary": "data"
      }
    },
    "target": "node202"
  },
  {
    "source": "node208",
    "properties": {
      "type": {
        "primary": "influence",
        "secondary": "code"
      }
    },
    "target": "node194",
    "id": "edge208",
    "entity-type": "edge"
  },
  {
    "source": "node196",
    "properties": {
      "layers": [
        "sequencing"
      ],
      "type": {
        "primary": "control-flow-
              sequencing"
      }
    },
    "target": "node204",
    "id": "edge209",
    "entity-type": "edge"
  },
  {
    "source": "node204",
    "properties": {
      "layers": [
        "sequencing"
      ],
      "type": {
        "primary": "control-flow-
              sequencing"
      }
    },
    "target": "node197",
    "id": "edge210",
    "entity-type": "edge"
  },
  {
    "source": "node197",
    "properties": {
      "layers": [
        "sequencing"
      ],
      "type": {
        "primary": "control-flow-
              sequencing"
      }
    },
    "target": "node204",
    "id": "edge211",
    "entity-type": "edge"
  },
  {
    "source": "node204",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node200",
      "id": "edge212",
      "entity-type": "edge"
    },
    {
      "source": "node200",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node201",
      "id": "edge213",
      "entity-type": "edge"
    },
    {
      "source": "node201",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node205",
      "id": "edge214",
      "entity-type": "edge"
    },
    {
      "source": "node205",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node202",
      "id": "edge215",
      "entity-type": "edge"
    },
    {
      "source": "node201",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node209",
      "id": "edge216",
      "entity-type": "edge"
    },
    {
      "source": "node209",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node198",
      "id": "edge217",
      "entity-type": "edge"
    },
    {
      "source": "node198",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node197",
      "id": "edge218",
      "entity-type": "edge"
    },
    {
      "source": "node197",
      "properties": {
        "layers": [
          "sequencing"
        ],
        "type": {
          "primary": "control-flow-
                sequencing"
        }
      },
      "target": "node209",
      "id": "edge219",
      "entity-type": "edge"
    }
  ]
}
```

Appendix F

IDA Pro Plugin: Ponce Plus Colocations View for Rapid Judgments Based on Taint

The purpose of this IDA Pro [55] plugin is to provide a visual representation for tainted lines and their colocations that is similar to the Microsoft Word colocations view of search terms (see Figure F.2(a) for an example). This plugin should allow users to quickly view decompilation of tainted lines to determine whether to inspect particular line or function. Tainted lines are extracted from Ponce [62] with a Python script, displayed using our List View plug-in, and can be viewed as a graph. This tool can be used to quickly display information about any lines of interest.

F.1 Usage

1) Install Ponce [62], `https://github.com/illera88/Ponce`.

2) Copy and paste our script TaintedInstanceListView.py into your IDA Pro plugins directory. Section F.2 provides the basic function names and comments for this script.[1]

3) Run the Taint Analysis engine from Ponce to generate a list of tainted instruction on the binary. This analysis is dynamic and requires the binary to be run in IDA's debugger.

   a) Set a breakpoint somewhere near your desired data.

   b) Enable the taint engine in Ponce, using options to show debug information in the output windows, enable the optimization to "only generate on tainted/symbolic", add comments with controlled operands, rename tainted function names, and paint executed instructions.

   c) Run the binary in IDA's debugger (configuring if necessary) and wait for the execution to hit the breakpoint.

   d) Select the data you wish to taint in the hex view, right-click on the data, and select Taint. Then continue the program execution to trace the taint.

4) Once the taint analysis is complete, run the plugin Alt-F8 to parse the disassembly text and extract the tainted instructions, addresses, and context. Figure F.2(c) shows the results of the

[1] Contact the primary authors for access to the full script.

Ponce analysis while extracting information, and Figure F.2(b) shows our view of tainted variables and colocations.

5) Interact with the text results via IDA:
   - Double-clicking on memory locations and offsets sends you to the line in any IDA view.
   - Hover over the line of interest and press 'D' to delete a line.
   - Hover over the line of interest and press 'A' to append a line to the end of the output.
   - Hover over the line of interest and press 'R' to refresh output.
   - Hover over the line of interest and press 'E' to edit a line or add a comment.

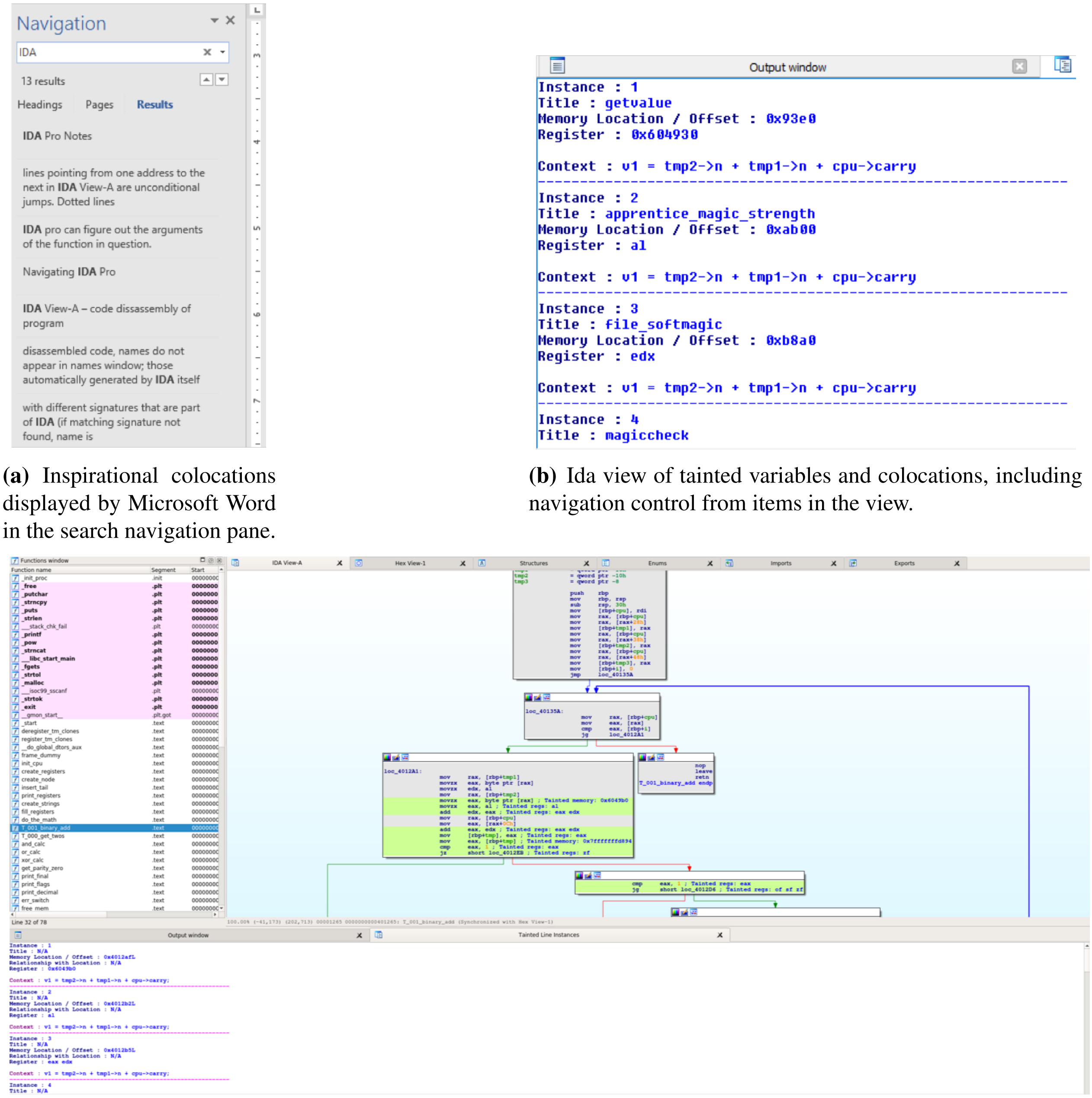
6) Edit the results in a .txt file:
   - The script automatically generates a standardized text file.
   - Set the location of this text file at the top of the Python plug-in script with the variable created_txtfile. Note: As long as you follow the standardized format text file and set the read_from_this_txtfile variable, you can display any information in a list view in IDA.
   - You can edit this text file in any editor. If you want to view an edited output file in the plug-in, you must a) Set the read_from_ponce_flag to False. b) Set the read_from_this_txtfile variable to the path of the edited text file. c) Re-run the plug-in in IDA using Alt-F8/

7) Interact with a graph of the taint relationships:
   - A .gv file is automatically created when you run the script.
   - The location of this .gv file can be specified with the script variable gvfile.
   - Using graphviz (`http://www.graphviz.org/Download.php`), you can run gvedit to open and view the automatically generated gv file.
   - You can edit the graph in gvedit by looking at the graphviz documentation (`http://www.graphviz.or`
   - Alternatively, you can alter the created_textfile directly. For example, if you want to create a link between A and B in your graph, change the Relationship with Location field to have the Memory Location / Offset value of B. Rerun the IDA plug-in, ensuring that your read_from_this_txtfile is the same as your created_textfile. The .gv file will be updated, and you can view the results as previously.

F.2 TaintedInstanceListView.py

This is the outline of the python script to extract (using Ponce) and display taint results with code colocations but without the entire body of code. We include primarily function names and debug strings; we elide the actual code.

Figure F.1. Screen shots of the inspiration for displaying colocations and the IDA Pro plugin.



IDA

13 results

IDA Pro Notes

lines pointing from one address to the next in IDA View-A are unconditional jumps. Dotted lines

IDA pro can figure out the arguments of the function in question.

Navigating IDA Pro

IDA View-A – code dissassembly of program

disassembled code, names do not appear in names window; those automatically generated by IDA itself

with different signatures that are part of IDA (if matching signature not found, name is

```
Instance : 1
Title : getvalue
Memory Location / Offset : 0x93e0
Register : 0x604930

Context : v1 = tmp2->n + tmp1->n + cpu->carry
------------------------------------------------------------------
Instance : 2
Title : apprentice_magic_strength
Memory Location / Offset : 0xab00
Register : al

Context : v1 = tmp2->n + tmp1->n + cpu->carry
------------------------------------------------------------------
Instance : 3
Title : file_softmagic
Memory Location / Offset : 0xb8a0
Register : edx

Context : v1 = tmp2->n + tmp1->n + cpu->carry
------------------------------------------------------------------
Instance : 4
Title : magiccheck
```

(a) Inspirational colocations displayed by Microsoft Word in the search navigation pane.

(b) Ida view of tainted variables and colocations, including navigation control from items in the view.

Function name	Segment	Start
_init_proc	.init	0000000C
_free	**.plt**	**0000000**
_putchar	**.plt**	**0000000**
_strncpy	**.plt**	**0000000**
_puts	**.plt**	**0000000**
_strlen	**.plt**	**0000000**
___stack_chk_fail	.plt	0000000C
_printf	**.plt**	**0000000**
_pow	**.plt**	**0000000**
_strncat	**.plt**	**0000000**
___libc_start_main	**.plt**	**0000000**
_fgets	**.plt**	**0000000**
_strtol	**.plt**	**0000000**
_malloc	**.plt**	**0000000**
___isoc99_sscanf	.plt	0000000C
_strtok	**.plt**	**0000000**
_exit	**.plt**	**0000000**
__gmon_start__	.plt.got	0000000C
_start	.text	0000000C
deregister_tm_clones	.text	0000000C
register_tm_clones	.text	0000000C
__do_global_dtors_aux	.text	0000000C
frame_dummy	.text	0000000C
init_cpu	.text	0000000C
create_registers	.text	0000000C
create_node	.text	0000000C
insert_tail	.text	0000000C
print_registers	.text	0000000C
create_strings	.text	0000000C
fill_registers	.text	0000000C
do_the_math	.text	0000000C
T_001_binary_add	.text	0000000C
T_000_get_twos	.text	0000000C
and_calc	.text	0000000C
or_calc	.text	0000000C
xor_calc	.text	0000000C
get_parity_zero	.text	0000000C
print_final	.text	0000000C
print_flags	.text	0000000C
print_decimal	.text	0000000C
err_switch	.text	0000000C
free_mem	.text	0000000C

Line 32 of 78

```
tmp2            = qword ptr -10h
tmp3            = qword ptr -8

                push    rbp
                mov     rbp, rsp
                sub     rsp, 30h
                mov     [rbp+cpu], rdi
                mov     rax, [rbp+cpu]
                mov     rax, [rax+28h]
                mov     [rbp+tmp1], rax
                mov     rax, [rbp+cpu]
                mov     rax, [rax+38h]
                mov     [rbp+tmp2], rax
                mov     rax, [rbp+cpu]
                mov     rax, [rax+48h]
                mov     [rbp+tmp3], rax
                mov     [rbp+i], 0
                jmp     loc_40135A

loc_40135A:
                mov     rax, [rbp+cpu]
                mov     eax, [rax]
                cmp     eax, [rbp+i]
                jg      loc_4012A1

loc_4012A1:
                mov     rax, [rbp+tmp1]
                movzx   eax, byte ptr [rax]
                movzx   edx, al
                mov     rax, [rbp+tmp2]
                movzx   eax, byte ptr [rax] ; Tainted memory: 0x6049b0
                movzx   eax, al ; Tainted regs: al
                add     edx, eax ; Tainted regs: eax edx
                mov     rax, [rbp+cpu]
                mov     eax, [rax+0Ch]
                add     eax, edx ; Tainted regs: eax edx
                mov     [rbp+tmp], eax ; Tainted regs: eax
                mov     eax, [rbp+tmp] ; Tainted memory: 0x7fffffffd894
                cmp     eax, 1 ; Tainted regs: eax
                jz      short loc_4012EB ; Tainted regs: zf

                nop
                leave
                retn
T_001_binary_add endp

                cmp     eax, 1 ; Tainted regs: eax
                jg      short loc_4012D6 ; Tainted regs: of sf zf
```

100.00% (-41,173) (202,713) 00001265 0000000000401265: T_001_binary_add (Synchronized with Hex View-1)

```
Instance : 1
Title : N/A
Memory Location / Offset : 0x4012afL
Relationship with Location : N/A
Register : 0x6049b0

Context : v1 = tmp2->n + tmp1->n + cpu->carry;
------------------------------------------------------------------
Instance : 2
Title : N/A
Memory Location / Offset : 0x4012b2L
Relationship with Location : N/A
Register : al

Context : v1 = tmp2->n + tmp1->n + cpu->carry;
------------------------------------------------------------------
Instance : 3
Title : N/A
Memory Location / Offset : 0x4012b5L
Relationship with Location : N/A
Register : eax edx

Context : v1 = tmp2->n + tmp1->n + cpu->carry;
------------------------------------------------------------------
Instance : 4
Title : N/A
```

(c) Ponce taint analysis results and tainted instructions, addresses, and decompilation context.

```
import sys, struct
from collections import namedtuple
import idc
import idaapi
import re

# If you are reading from Ponce, set this flag to ’True’
# If you are reading from a previously automatically generated text file,
# set this flag to ’False’ and set your read_from_this_txtfile to point to
# the location of the file.
read_from_ponce_flag = True

# Important! If you are reading your tainted line instance information from
# a text file, please edit the path below to point to the right file.
read_from_this_txtfile = ”ordered_tl_list.txt”

# Every time the plug-in is run, this plug-in creates a standardized text file
# from the Ponce output.
# This file can be edited in a text editor and
# and re-run in IDA using Alt-F8.
created_txtfile = ”TaintedLineInstances.txt”

# Edit the path below to point to the directory where you want to save
# the automatically generated .gv file which can be run in GVEdit.exe.
gvfile = ”inputToGVEdit.gv”

# Runs the plug-in in IDA Pro
class myplugin_t(idaapi.plugin_t):
    def init(self):
    def run(self, arg):
    def term(self):

def PLUGIN_ENTRY():
    return myplugin_t()

# Holds information about each tainted line instance in a named tuple
taint_inst = namedtuple(’taint_inst’, [’inst_no’, ’title’, ’mem_loc’, ’next’, ’reg’, ’context’])
inst_list = []
total_inst_no = 0

# Initialize IDAPython GUI in custviewer format
# Some of this code was sourced from https://github.com/zachriggle/idapython/blob/master/examples/ex_custview.py
        ...

class simplecustviewer_t(object):
        ”””The base class for implementing simple custom viewers”””
        def __init__(self):
        def __del__(self):
                ”””Destructor. It also frees the associated C++ object”””
        @staticmethod
        def __make_sl_arg(line, fgcolor=None, bgcolor=None):
        def Create(self, title):
                ”””Creates the custom view. This should be the first method called after instantiation”””
        def Close(self):
                ”””Destroys the view. One has to call Create() afterwards. Show() can be called and it will call Create()
                    internally.”””
        def Show(self):
                ”””Shows an already created view. If the view was closed, then it will call Create() for you”””
        def Refresh(self):

        def RefreshCurrent(self):
                ”””Refreshes the current line only”””
        def Count(self):
                ”””Returns the number of lines in the view”””
        def AddLine(self, line, fgcolor=None, bgcolor=None):
                ”””Adds a colored line to the view”””
        def InsertLine(self, lineno, line, fgcolor=None, bgcolor=None):
                ”””Inserts a line in the given position”””
        def EditLine(self, lineno, line, fgcolor=None, bgcolor=None):
                ”””Edits an existing line.”””
        def DelLine(self, lineno):
                ”””Deletes an existing line”””
        def GetLine(self, lineno):
                ”””Returns a given line”””
        def GetCurrentWord(self, mouse = 0):
                ”””Returns the current word”””
        def GetCurrentLine(self, mouse = 0, notags = 0):
                ”””Returns the current line.”””
        def GetPos(self, mouse = 0):
                ”””Returns the current cursor or mouse position.”””
        def GetLineNo(self, mouse = 0):
                ”””Calls GetPos() and returns the current line number or -1 on failure”””
        def Jump(self, lineno, x=0, y=0):

        def AddPopupMenu(self, title, hotkey=””):
                ”””Adds a popup menu item”””
        def GetTCustomControl(self):
                ”””Return the TCustomControl underlying this view.”””

# The action handler
class say_something_handler_t(idaapi.action_handler_t):
```

```
class mycv_t(simplecustviewer_t):
        def Create(self, sn=None):
                # Form the title ”Tainted Line Instances”
                # Create the customviewer
                # For each instance, print the standard information:
                        #Instance : <inst_no>
                        #Title : <instance.title>
                        #Memory Location / Offset : <instance.mem_loc>
                        #Relationship with Location : <instance.next>
                        #Register : <instance.reg> \n
                        #Context : <instance.context>
                        #”————————————————————————————————————————————————————————

        def OnClick(self, shift):
                ”””User clicked in the view”””
        def OnDblClick(self, shift):
                ”””User dbl−clicked in the view; jump to the address of the current word”””
        def OnClose(self):
                ”””The view is closing. Use this event to cleanup.”””
        def OnKeydown(self, vkey, shift):
                ”””User pressed a key; respond, and ask the user for confirmation if changing a line
                @param vkey: Virtual key code”””
                # Press ’Esc’ to exit
                # Press ’D’ to delete line at cursor
                # Press ’R’ to refresh line at cursor
                # Press ’A’ to append line to end of output file
                # Press ’I’ to insert a line at cursor
                # Press ’E’ to edit a line or add a comment

# Some of this file was sourced from https://github.com/EiNSTeiN−/idapython/blob/master/examples/vds_xrefs.py
def get_decompiled_line(cfunc, ea):

# This function creates a standardized tainted line instance text file
# Title: N/A\n
# Memory Location / Offset : <instance.mem_loc> \n
# Relationship with Location : N/A\n
# Register : <instance.reg> \n
# Context : <instance.context> \n
# ——————————————————————————————————————————————————————\n
def create_std_file():

# This function grabs tainted line instance information through the Ponce plug−in.
def obtain_info_using_ponce():
        # Begin EA
        # Iterate through each function in the program
                # Iterate through each ”chunk”, and look for a tainted instruction

# This function uses regular expressions to parse through a standardized
# tainted line instance text file.
def obtain_info_from_txt_file():
        # Read from file
        # Parse each record

def create_graph():
        # Write gvfile digraph information
                #”””digraph G {\nrankdir = ”LR”\nnode [shape=record, fontname=Courier, style=”filled”, gradientangle=90];\n\n
                        #node <instance.inst_no> [label = ” <inst_no> Instance: <instance.inst_no> |<mem_loc> Memory Location /
                              Offset: <instance.mem_loc> |<next>Relationship with Location: <instance.next> |<reg>Affected
                              Register: <instance.reg >”];\n\n
                                        #”node<i+1>”:mem_loc −> ”node<j+1>”:mem_loc;\n

def show_win():
        # Obtain tainted line instance information using Ponce
        # or Obtain tainted line instance information from txt file

        # Create standardized tainted line instance text file that user can edit as they wish to customize their IDA Pro display.

        # Create a .gv file to read into GVEdit.exe

        # Register actions, including action name, action text, action handler, and empty action icon
```

Appendix G

Binary Analysis, Symbolic Execution, and Situational Awareness

The concept of situational awareness was developed by Mica Endsley and others to capture the requirements of mental models and how individuals construct them while engaging in real-time decision making [44]. Much of the work that motivated the concept and development of design principles for situational awareness (SA) was conducted with war-fighters and medical personnel relying on knowledge, memory and interpretation of sensor data to make decisions.

Situational Awareness has three levels:

1) perception of the elements in the environment within a volume of time and space,
2) comprehension of their meaning, and
3) projection of their status in the near future.

The activities that were studied in the development of situational awareness theory are similar to binary code analysis in a few important ways that make the design principles for situational awareness relevant. Specifically, binary code analysts are also:

- working in environments that put great demands on attention and working memory.
- constructing mental models of the "current state of the system" (although for binary analysts these are models of self-generated understanding, not driven by changes in the environment).
- projecting the status of the "current state of the system" that is being held in memory in order to devise hypothesis tests for whether the current understanding is accurate.

Table 15.1, "Summary of SA-Oriented Design Principles" from [44] provides a list of SA Design Principles organized by category. Many of those design principles are relevant for binary analysis, excepting the *Alarm Design Principles*. Here, we copy out the design principles that are most relevant for symbolic execution engine specializations for a specific application, e.g., specializing angr [94] to detecting known environmental triggers [49] or backdoors [93].

- No. 1 *General* Organize information around goals
- No. 2 *General* Present Level 2 information directly – support comprehension
- No. 3 *General* Provide assistance for Level 3 SA projections
- No. 9 *Certainty Design Principles* Explicitly identify missing information
- No. 10 *Certainty Design Principles* Support sensor reliability assessment
- No. 11 *Certainty Design Principles* Use data salience in support of certainty
- No. 12 *Certainty Design Principles* Represent information timeliness
- No. 13 *Certainty Design Principles* Support assessment of confidence in composite data
- No. 14 *Certainty Design Principles* Support uncertainty management activities
- No. 15 *Complexity Design Principles* Just say no to feature creep – back the trend
- No. 16 *Complexity Design Principles* Manage rampant featurism through prioritization and flexibility
- No. 17 *Complexity Design Principles* Insure logical consistency across modes and features
- No. 18 *Complexity Design Principles* Minimize logic branches
- No. 19 *Complexity Design Principles* Map system functions to the goals and mental models of users
- No. 20 *Complexity Design Principles* Provide system transparency and observability
- No. 36 *Automation Design Principles* Provide SA support rather than decisions

Appendix H

Testing and Evaluation of Software Tools for Binary Auditing

The process of developing software that solves problems faced by binary software auditors and that can be integrated into their existing workflows requires both domain expertise and understanding of how to measure the success of the integration of software into an existing human-computer system. Effective testing and evaluation requires expertise in cognitive task analysis, human performance measurement of software systems and binary code analysis.

Testing the usability of a software package requires designing test scenarios that capture the demands faced by both the user and the system in an operational environment, while scoping those demands to a time frame that is reasonable for evaluation. These scenario requirements can be determined through a cognitive task analysis that uses semi-structured interviews of domain experts to identify essential processing steps in task performance, the information needs at each of those steps, and the constraints imposed by the work setting (e.g., time, development environment).

Software can be evaluated through self-report measures such as the 10-item System Usability Scale (SUS) [16] or 4-item Usability Metric for User Experience (UMUX) scale [48]. These measures can only evaluate the learnability and usability of a software system; they do not provide diagnostic information regarding how the software might be improved. Other measurements might focus on individual human constraints that software is not sufficiently acknowledging and supporting. For example, because binary code auditing requires significant cognitive loads during much of task performance, software that adds significant additional cognitive burden is likely to be difficult to integrate into the existing workflows. Measures of cognitive workload, such as NASA-TLX [53], can be used to assess whether a new software tool is likely to introduce an additional cognitive burden that might deem the software unusable.

In situations where the usability of a system is worse than desired, we use techniques taken from experimental psychology to assess and diagnose opportunities for improvement. Experimental protocols have experts perform representative tasks while using their various tools. The eye movements, mouse movements, and thoughts (through think-aloud protocols) of the experts are recorded simultaneously to assess attentional deployment, cognitive failures, and cognitive efficiency. These task performance artifacts are then segmented and categorized to provide a model of what fluent task performance looks like, as well as metrics of how the task performance may be disrupted through memory errors, execution errors, and user frustrations. Task performance disruptions can be linked to specific features of the software system, thus providing a means of diagnosing how to improve the software tool.

Appendix I

Literature Review of Eye Tracking for Understanding Code Review

In February of 2017, performed a cursory literature review of academic work related to eye tracking for understanding code review. We present the results of that literature review for the interested reader.

Reading and understanding code is complex, beyond that of typical reading. However, currently little is known about the cognitive processes involved in code comprehension, and, importantly, what separates experts from novices. There has been a recent push in the computer science literature to better understand how experts understand code in order to leverage these insights to improve computer science education and reduce the time it takes for analysts to become experts. One method uses eye-tracking, in which infrared cameras record a reader's gaze position on a screen with high temporal resolution (anywhere from 60Hz up to 1000Hz—that is, recording the gaze position once every 16 milliseconds to once every millisecond). Eye tracking technologies record two important aspects of human reading behaviors: *fixations* (periods of relative stability in which readers pause to look directly at, or fixate, an area of interest), and *saccades* (the actual movements of the eyes from one location to another). Average fixation durations in reading last 200-250 ms, whereas saccades last roughly 50ms (see Rayner, 1998, [85] for a comprehensive review of eye movement behaviors associated with reading). Saccades are ballistic movements of the eyes (once they start, they cannot be stopped), and due to a process called *saccadic suppression*, human eyes do not register the input of visual information during saccades (which is why the world does not look blurry every time we move our eyes). Saccades can move in the forward direction, which in English is from left to right, but regressions are saccades that move the eyes backwards in the text (and encompass roughly 15% of saccades in typical reading). When reading, humans can clearly see roughly 7-8 letters to the right of fixation, and 2-3 letters to the left of fixation: this is known as the *perceptual span*, and it reflects that fact that attention is typically directed forward in the direction of reading.

Eye tracking has been used to understand text reading, and visual attention to scenes more generally, for over 30 years (see [37] for a review of eye movement behaviors across a variety of visual tasks). Many of the basic insights gained from the text reading literature can be broadly applied to code comprehension and offer a good starting place to the types of questions that can be answered using eye-tracking. Several important eye-tracking measures from the reading literature include: fixation durations (how long readers look at particular areas), fixation count or proportion (how many total fixations a person makes in the region), and probability of a regression to a region (whether a reader returns to a region after having already read it). Although the total amount of time spent reading a region is often highly correlated with the number of times it was fixated, these two different pieces of information can sometimes diverge to provide insights into more detailed reading strategies. Typically, sentences are dived into regions of interest, which can be as small as a single word or as large as a paragraph. Reading times are then calculated for these regions of interest and compared to reading times on other regions that differ in important characteristics (e.g., grammatical versus ungrammatical sentence endings, sensical versus non-sensical words). It is best to compare reading times on different regions *within* individuals, as people can vary widely in their reading patterns.

General findings from the reading literature [85] indicate that words that are easier to process (for example, because they are frequently encountered in the language or expected from the context) are looked at for less time relative to words that are more difficult to process (for example, because they are less frequently encountered, unexpected from the context, contain a spelling or grammatical error). Readers can also skip words that are highly expected, or very short, because they can be seen in the perceptual span, mentioned above. There is much evidence that readers process language incrementally—that is, they attempt to comprehend every word *as it is encountered*, rather than storing chunks in working memory and putting them all together at the end of the sentence. When readers encounter a word that is semantically confusing or that grammatically violates the sentence structure, they will often 1) stop and spend more time fixating the word, and/or 2) launch a regressive saccade to earlier parts of the sentence. These basic eye movement behaviors can thus offer insights into the ongoing cognitive processes readers use to comprehend text.

Computer science researchers have begun to apply these principles to the comprehension of code, especially in trying to understand the difference between expert and novice behavior. In general, experts spend more time reading meaningful lines in algorithms as compared to novices, whereas novices spend more time reading comments [11]. In one set of studies, researchers tried to identify differences between reading text and code, and whether these patterns differ for experts and novices. Busjahn et al. [17] developed novel metrics to identify how *linear* a subject's code reading pattern was, finding the most linear patterns in reading natural language text (unsurprisingly). They found that, when reading *code*, however, both novices and experts exhibited less linear reading patterns intermixed with more vertical movements. Interestingly, experts were even *less* linear than novices, instead adopting a more execution-based style of reading code where they tended to make many vertical movements to trace the execution of the code.

A second set of studies tried to identify general eye movement patterns over code for both experts and novices in the realm of defect detection. One such study [106] found that, when reading a small piece of unfamiliar code, programmers engage in two different reading phases: *scanning*, which they defined as the time it takes readers to fixate 80% of the programs' lines, and *reading*, in which fixations were more concentrated within specific liens. This study, and a subsequent replication [90], found that these patterns tend to hold up amongst both expert and novice readers, although experts tend to spend less time scanning the code initially. They also found that the more time readers spent in their initial scan of the program, the less time it took them to identify the defective line, suggesting that the initial overall sweep of the code played an important role in establishing a mental model of the code's functionality. Beelders and du Plessis [11] found that these general scanning patterns were unaffected by syntax highlighting (as compared to black and white presentation of code), which provides interesting insights into the types of cues programmers do and do not rely on.

Another set of studies tried to use eye-tracking to ask whether experts use beacons more efficiently than novices. Beacons, or stereotypical indicators of code function, are thought to drive the top-down comprehension of code by experts. Crosby, Scholtz, and Wiedenbeck [28] took an approach that was much more similar to the text reading literature. In two initial experiments, they had programmers rate different parts of code by their importance to understanding the overall functionality of the code. Then, in the eye-tracking experiment, they divided pieces of code into smaller regions of interest, and categorized these regions based on the functions they served (simple statements, complex statements, comments, keywords, etc.) and the importance given to them by the independent set of raters. They found that experts made many more fixations on complex statements as compared to novices, suggesting that they were able to identify and utilize these beacons more easily than novices. This shows how eye-tracking can be 1) combined with offline subjective ratings of evaluating code and 2) used to confirm which pieces of code are most important to facilitating understanding.

Other work includes: asking how commenting code into chunks helps comprehension [18], identifying differences in comprehension between C++ and Python source code [105], determining gaze patterns associated with debugging behavior [10], and determining eye movement patterns associated with traceability links in code [91].

The current literature seems to indicate that there is growing interest in the use of the eye-tracking methodology to understand how programmers understand and debug code – and how these processes differ between experts and novices. One drawback to this fledgling literature is that many of these studies have inconclusive results due to poor experimental design, incorrect statistics, poor eye tracking collection procedures, or failure of the authors to report critical information necessary to interpret their findings (tables or plots of means across conditions). Most of the papers are published conference proceedings and, due to their short nature, offer few methodological details. Moreover, the studies are conducted across a range of programming languages (C, C++, .Net, Java, Python), and differences across studies are often confounded with programming language differences as well. One drawback that will be harder to overcome includes the use of very short snippets of code that do not represent the true complexity of typical programming problems. There is a clear desire among the computer science community for information regarding code comprehension, and there is a need for rigorous, well-designed and controlled empirical studies to replicate many of these earlier findings.

Many of the researchers have identified end-state goals for their vision of what eye-tracking research would add to the code comprehension literature: to develop eye-tracking metrics that quantify "expert" programmer behavior. These metrics of expert behavior could then be used to: provide a metric by which to measure an individual's programming skill [90]; be leveraged to provide instruction or visual cues to novices to help them attend to the same areas attended to by experts [17], or to programmers of any skill level who are failing to attend to the important areas [28]; or, identify additional beacons used by experts to comprehend code [5]. These goals all seem to be aimed at answering the questions of how people understand code and how we can more effectively teach people these skills. As such, identifying expert behaviors across a range of activates, from comprehension to debugging to reverse engineering, will provide a basis for accomplishing these goals.

DISTRIBUTION:

3	MS 1319	Michelle Leger, 05836
1	MS 1110	Karin Butler, 09131
1	MS 0359	D. Chavez, LDRD Office, 1911
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories