SAND2019-0974C

Creating a User-centric Data Flow Visualization: A Case Study

Karin Butler[1], Michelle Leger[1], Denis Bueno[1], Christopher Cuellar[1], Michael J. Haass[1], Timothy Loffredo[1], Geoffrey Reedy[1], and Julian Tuminaro[1]

Sandia National Laboratories, Albuquerque NM 87123, USA,
kbutle@sandia.gov, maleger@sandia.gov

Abstract. Vulnerability analysts protecting software lack adequate tools for understanding data flow in binaries. We present a case study in which we used human factors methods to develop a taxonomy for understanding data flow and the visual representations needed to support decision making for binary vulnerability analysis. Using an iterative process, we refined and evaluated the taxonomy by generating three different data flow visualizations for small binaries, trained an analyst to use these visualizations, and tested the utility of the visualizations for answering data flow questions. Throughout the process and with minimal training, analysts were able to use the visualizations to understand data flow related to security assessment. Our results indicate that the data flow taxonomy is promising as a mechanism for improving analyst understanding of data flow in binaries and for supporting efficient decision making during analysis.

Keywords: visualization, data flow, vulnerability analysis, reverse engineering, taxonomy development, requirements, binary analysis

1 Introduction

1.1 Background

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. More efficient and effective processes that will assure that software does not have vulnerabilities are needed [1].

Ideally, automated tools would assess and protect binary software statically, without executing the program. Static binary analysis avoids 1) needing access to all of the supporting systems required to run the binary, 2) missing vulnerabilities introduced during the translation from source code to binary [2], and 3) introducing threats from actually running the code. Unfortunately, automatic static binary analyses do not scale to real-world software [3].

Currently, experts assess and protect systems by performing binary vulnerability analysis manually with assistance from automated tools [4]. These experts

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & 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.

use extensive domain knowledge of binary code, operating systems, hardware platforms, programming languages, and vulnerabilities; they engage in reverse engineering to understand binary programs [5], 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].

Current analyst tools have been developed and optimized to support understanding program *control flow*, the order in which individual statements, instructions, or function calls are executed or evaluated in a program. However, as the capability to detect control flow vulnerabilities has improved, attackers have started to take advantage of how data passing through program functions influences other program data and program decisions [7]. 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 control flow and data 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.

Unfortunately, data flow is difficult to understand, particularly when working from a binary. Analysts find that the current set of tools for understanding data flow is inadequate.

To begin to fill this gap in the analyst toolset, we used human factors methods to derive requirements for an analyst-centric interprocedural data flow visualization to assist binary reverse engineers in identifying and mitigating vulnerabilities in code. Working with experienced binary analysts, we used a rolling discovery process to derive our requirements through semi-structured interviews, applied cognitive task analysis knowledge audits, cognitive walkthroughs, and a two-stage modified sorting task. Our contributions include:

– a description of a modified sorting task, a human factors method to achieve consensus about mental models used across diverse tasks (Section 2.2),
– a taxonomy of essential features to support vulnerability analyst understanding of data flow in static analysis of binary code (Section 3),
– and an informal evaluation of the static requirements of our taxonomy, through proof of concept and analytic evaluation (Section 4).

1.2 Related Work

Current Inadequate Data Flow Visualizations Traditional static data flow analyses use unwieldy mathematical representations for computation [8]. Most visualizations of these analyses overlay data flow or other information onto a control abstraction, the control flow graph (CFG) [9][10][11], the call graph [12][13], a file view [14] or a condensed text-based view of the code [15][16]. 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 [17][18], delocalized data flow relationships [19][20], historical animated views [21] and hierarchical interactive views [22], and even hypothesis-driven understanding [23][24], but many of those visualization mechanisms do not appear to be implemented in the common reverse engineering platforms of today [9][25][26].

Visualizations of program dependence graphs (PDG) [27], annotated system dependence graphs (interprocedural PDGs) [28] and static value flow graphs [29] provide a reasonably intuitive view of many important data flow relationships. However, these are statically computed graphs that are not designed to be updated, they are cognitively overwhelming, and they tend to ignore values. One visualization of a dynamic data flow graph shows location, execution time, and certain values [7], making some relationships easier to understand than in other representations. However, these dynamic representations cover *one* potential set of relationships associated with a single execution, and thus they do not generalize well to static analyses. Other recent work provides insight into values [30], 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 [9] and Dream [31] provide the most intuitive advanced data flow representations today, encoding data flow information in automatically selected variable names. The Dream++ extension [32] 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* of the code rather than providing code folding [33], and analysts inject knowledge at a different layer of representation than that displayed (i.e., on the disassembly).

User-centered Design Our work is heavily influenced by two individuals who have thought deeply about supporting user decision making and understanding: Storey [5] and Victor [34]. 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. 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) [34]; easy movement between multiple levels of abstraction [35]; and natural interactive control mechanisms [36]. 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-experts [4]. The DARPA CHESS program is building research to support humans and computers working together to reason about the security of software artifacts [37]. Research groups such as [38] 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.

2 Approach

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. We describe these activities in more detail in Section 2.1.

These activities showed that vulnerability analysts need to understand a range of characteristics of 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. We considered conducting additional cognitive walkthroughs to identify essential data flow characteristics across the broad range of data flow understanding tasks, but we decided not to 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.

To develop visualization requirements that would support a range of data flow analysis tasks, we next focused on gathering information about analyst mental models from artifacts of their own projects spanning such tasks.

An activity that can reveal the mental models of users is a sorting task, a task that is more commonly used to inform the grouping and naming of categories in an interface [39]. 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. We hypothesized that binary analysts might reveal general purpose data flow elements through a sorting task [39] over their own meaningful data variable and value names.

In our case, however, we did not have a consistent set of elements for analysts to sort. Instead, we had artifacts that analysts had created to record analysis-relevant information from various completed projects. These artifacts were created using specialized reverse engineering tools, which allow analysts to add comments, to rename code elements like functions and variables, and to propagate assessment-relevant names through binary code. When an analyst encounters a previously-renamed element in another context, an assigned name can provide important information that has already been discovered about that element. Assigned names might reveal the general purpose data flow elements analysts needed to see in a visualization. However, these names vary across projects and across analysts according to analysis goals and personal preference, and they include information about other program features as well (e.g., memory utilization or control flow). Thus, we 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 for someone unfamiliar with all the projects 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. We had analysts sort the names they gave to data flow-related functions and variables taken from diverse, previously analyzed binaries, and we had experts perform a second stage of evaluation to find the commonalities and essential data flow information shared across these analysis projects. We describe the two-stage modified sorting task in more detail in Section 2.2.

In Section 3, we present the derived requirements and an example visualization, and in Section 4, we describe our informal evaluation. Specifically, we evaluated our visualization through a proof of principle by using the derived static requirements to generate data flow visualizations for small binaries. We then tested the utility of one of these visualizations with an analytical test to gain confidence in the produced requirements.

This research was reviewed and approved by the Sandia National Laboratories Human Studies Board.

2.1 Requirements Development from Interviews and Walkthroughs

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

The first round of semi-structured interviews were general cognitive task analysis interviews with three experienced analysts to identify 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 [40]. This data flow analysis task requires identifying where an attacker might control the data in a program and whether that data may influence security-relevant parts of code. The attack surface characterization task was chosen for the cognitive walkthroughs because it is 1) representative of many of the considerations when evaluating data flow and 2) amenable to a two-hour cognitive walkthrough.

In the second round of semi-structured interviews, three experienced analysts answered questions from an applied cognitive task analysis knowledge audit [41]. 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 attention allocation, decision making, and processes used by analysts during attack surface characterization. We selected the UNIX file utility version 5.10 [42][43], choosing from the AFL (American Fuzzy Lop) fuzzer bug-o-rama trophy case [44], a listing of vulnerabilities in real programs that were found by the program AFL-fuzz.[1] Three different experienced binary analysts with no experience with the chosen program were asked to characterize the attack surface of the file binary using static analysis only. They were tested individually. 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). During the two-hour test session, analysts were observed working in their chosen analysis environment while they thought aloud and answered questions posed by the human factors expert. See Appendix A for additional protocol details for the cognitive walkthrough.

We compiled the results of the interviews and walkthrough into a preliminary list of static data flow elements and interaction requirements for our data flow visualization.

2.2 Requirements Development from 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. To leverage the previous work of the expert binary analysts, we modified a sorting task [39] to take analyst-specific inputs and reveal mental models shared across analysts and projects. To determine the essential data flow elements across analysts and projects, we added a second stage to the sorting task. In this second stage, experts identified the commonalities and unique data flow elements that are essential for vulnerability analysis, informing our requirements for our data flow visualization.

The first stage of our 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 and

[1] 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 are real programs rather than small, designed programs, e.g., the Cyber Grand Challenge challenge binaries.

allowed the analysts to view the contextual information from the decompiled code for each 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. See Appendix B for instructions given to participants. Projects included a variety of applications and operating system drivers. The selected programs provided from 200 to over 500 names that had been assigned by the analyst. We asked analysts to spend up to 40 minutes going through the names and binning them into 7 to 10 different groups. This range of groups was recommended by the sorting task literature [39]. The groups were defined by the analyst to help teach someone else about how data values flow in the code. As expected given the time constraint, 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, at the end of the sorting period we asked analysts to review the entire list of names 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 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.[2]

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 as described below.

3 Results: Data Flow Visualization Requirements

We used the results from our 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.

[2] Because category names and descriptions were derived from proprietary assessments, we will not share these intermediate results.

To evaluate the utility of our requirements, we assigned visual design specifications to the elements in our requirements (taxonomy). We then produced a visualization of a binary and evaluated the utility of that 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 iterated on finding visualization design elements in an elaborated directed graph representation that could convey the required information. Using our data flow taxonomy, 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. Our final data flow taxonomy, including the elements and their visual representations are provided in Tables 1 and 2.

Using an iterative process of product creation and evaluation, we further developed the data flow requirements list while creating a data flow visualization for the Cyber Grand Challenge [45] binary CROMU_00034 (Diary_Parser)[3], choosing specific instantiations of visual design elements. Experienced binary reverse engineers frequently reviewed design choices and accessibility of data flow information.

4 Evaluation

Vicente recommended three ways to evaluate requirements developed through the application of human factors methods [48]: 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 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.

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? For this test, a novice reverse engineer just out of an undergraduate computer science program was asked to create a data flow visualization for two Cyber Grand Challenge binaries CROMU_00065 (WhackJack) and the KPRCA_00052 (pizza_ordering_system) using our list of data flow elements and visualization

[3] CGC Challenge binaries for DECREE are provided by DARPA [46]; versions ported to standard operating systems have been released by TrailOfBits [47]. We use the DARPA challenge name for binaries, but we provide the TrailOfBits name in parentheses.

Type	Sub-type	Visual Instantiation
value		oval or plus sign shape
	constant	no incoming value flow edges
	computed	plus sign shape
	constraint*	displays constraint description
	uncertain*	empty or displays '?'
location		rectangular shape
	local	dotted outline, when included
	heap	solid outline
	global	filled with gray
	shared memory*	NYR
aggregate		general aggregate NYR
	array	rectangle with double lines on sides
	structure	vertically stacked field locations
	and constraints	Boolean AND symbol (shield)
	or constraints	Boolean OR symbol (pointy shield)
code		text-only nodes (no edge or fill)
communication		most types NYR
	input*	STDIN: arrow shape, outgoing value flow edge
	output*	STDOUT: arrow shape, incoming value flow edges
annotation		
	initial configuration	value nodes filled with yellow
	data type*	NYR
	size	NYR

Table 1. Static requirements for information to be conveyed through nodes in data flow visualization to support vulnerability analysis of binaries. Sub-types marked with an asterisk (*) are expected to be updated by analysts throughout an analysis. NYR designates elements that are not yet represented. STDIN = standard input; STDOUT = standard output.

Type	Sub-type	Visual Instantiation
value flow		solid black line from source to destination node
function boundary		
	parameter	large black dot on edge
	return value	large black dot on edge
points-to		black dashed (one long, two short) line
comparison		black dotted line with long spaces
control influence		
	positive	black dotted line
	negative	gray dotted line
length		NYR, except colocation of length (source) in top-center of destination aggregate
sequencing		should not be represented
code influence		
	allocatable	points-to edge from code node to location
	freeable	points-to edge from code node to location
	readable	value flow edge from code node to location
	writeable	value flow edge from code node to location
synchronization		
	lifetime	NYR
	sometime	NYR
colocation		
	spatial	NYR
	subset	NYR
	overlap	NYR
lifetime		NYR

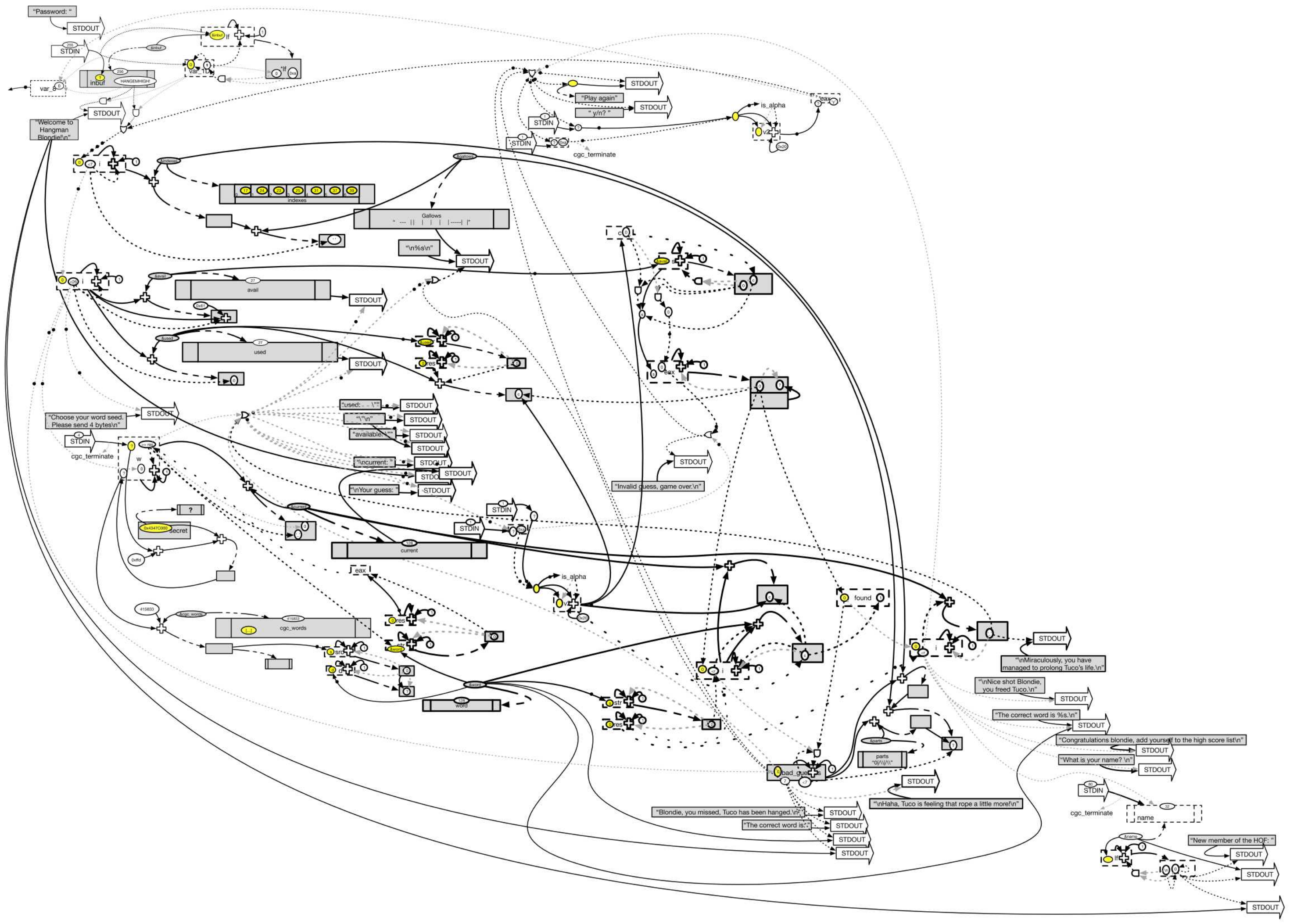
Table 2. Static requirements for information to be conveyed through edges in data flow visualization to support vulnerability analysis of binaries. Analysts are expected to be able to add and remove edges. NYR designates elements that are not yet represented.

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 list of data flow primitives. For example, we added STDIN and STDOUT communication nodes as a distinct type of location node, we called out that value computations and certain logical locations map to a single set of evidence (e.g., different uses of STDOUT should be represented by different nodes rather than by a single node throughout the binary), we annotated edges with function boundaries, we clarified that control flow enabled edges should come from the value nodes that trigger the related control flow in the binary, and we specifically relegated sequencing information to second-class information that is represented only when convenient.

The second proof of principle task identified a third Cyber Grand Challenge binary, EAGLE_0005 (CGC_Hangman_Game). This visualization was manually created for the entire binary and did not require modifications to our set of elements (see Figure 1); the 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.

To highlight how this visualization would be useful to binary analysts performing a vulnerability assessment, Figure 2 shows the portion of the EAGLE_0005 graph that includes the two vulnerabilities present in that code. In the upper left, up to 80 bytes are read from standard input; we denote this by showing the length of STDIN as 80 bytes. These bytes are read into name, a local array aggregate (i.e., a stack buffer) that has a length of only 32 bytes. This is an easily identifiable stack buffer overflow. The location of the name buffer is stored in the pointer &name as indicated by the black dashed line with one long line and two short dashes. The uninterrupted solid black line from this pointer to STDOUT together with the processing details indicate that the data is being passed without any checks, resulting in an easily identifiable format string vulnerability. These two vulnerabilities are relatively straight-forward to identify via a line-by-line analysis as well because they are wholly contained within a single function. However, the utility of the visualization is demonstrated in understanding how an attacker might exercise these vulnerabilities; for this task, an analyst requires interprocedural data flow understanding of nearly all of the 408 lines of code and data flow depicted in Figure 1. Current data flow visualizations do not enable effective visualization of an entire binary in this way. This 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.

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 (see [49] for the complete list of questions). An experienced reverse engineer who was not involved in the previous activities was given a 15-minute primer on understanding the graph elements using CROMU_00034, and then he was asked to answer the questions using only



"Password: "
STDOUT
STDIN
var_8
inbuf
HANGEMHIGH
"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"
"used: "
"available: "
"\ncurrent: "
"\nYour guess: "
current
secret
eax
cgc_words
word
found
"Invalid guess, game over.\n"
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: "

Fig. 1. 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.

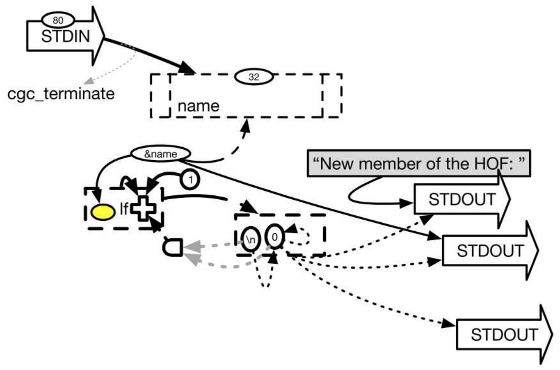


Fig. 2. The portion of the EAGLE_0005 data flow graph showing the two vulnerabilities known to be exhibited by that binary: a stack buffer overflow vulnerability, and a format string vulnerability.

the data flow visualization for the EAGLE_0005 binary. 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 and suggest a possible area for improvement of the visualization.[4] Overall, this result gives us some confidence that visualizations produced via our static requirements are useful for answering data flow questions. We believe that such visualizations have the potential to make larger analysis tasks more manageable without dramatically slowing smaller analysis tasks, though we have not tested this hypothesis yet.

5 Discussion

In this paper, we report the results of a case study of developing the design requirements for a new visualization, a data flow visualization to aid vulnerability analysts working with binary code to reason about and understand security-relevant data-flow information. We utilized several standard methods from human factors to identify a set of user-centric requirements that would be applicable to a range of real-world binaries and analysis goals. We also developed a two-stage modified sorting task to identify categories of data flow elements across heterogeneous work artifacts. During the proof of concept and analytical evaluation activities, with minimal training, analysts were able to use the visualizations to understand data flow related to security assessment. Our results indicate that this data flow taxonomy and visualization are promising for

[4] Our experienced analyst looking at the corresponding binary code had similar accuracy in the same amount of time. However, the analyst looking at the visualization tended to miss questions due to mis-interpretation of the visual elements present, while the analyst looking at the binary code tended to miss questions due to data influences or uses from other portions of the code.

improving analyst understanding of data flow in binaries and for supporting efficient decision making during analysis.

Our limited testing revealed some difficulty with the interpretation of pointers and their edges. This difficulty may be resolved with changes to how that information is depicted in our visualization, or it may require a revision to the taxonomy. We could also evaluate the ease with which analysts can learn and use the static visualization by using 1) a larger set of vulnerability analysts, 2) data flow vulnerabilities that are more difficult to identify manually in binaries, or 3) binaries with many more lines of code. As we describe in Section 5.1, though, we believe that further development of this taxonomy should be pursued using automated graph-building functions that have been integrated into analyst workflows.

In this case study we utilized a new procedure to distill heterogeneous analyst categorizations into consensus about the fundamental elements of the data flow visualization across varied code and analysis goals. In our two-stage modified sorting task, we relied on domain experts to identify the similarities and differences between the categories that resulted from 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 our 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 have influenced whether we found all the data flow elements that are important to vulnerability analysis. 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. We used an iterative development and design process during which reverse engineers frequently reviewed the effectiveness of the data flow elements and the design choices made in the visualization. We believe that others reproducing this research are likely to develop a similar set of requirements for understanding data flow in binary analysis.

5.1 Future Work

In this case study, generating the data flow visualizations was a time-consuming, manual process. Further development of a useful visualization requires determining how graph building can be integrated into analyst workflows. 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. For example, the data flow visualization should support the recording of unknowns and partial insights as they become known during the analysis. Additionally, during our preliminary data gathering, analysts indicated that they required 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. We believe that these interactive requirements are most important for successful integration of this visualization into analyst workflows, but such development remains future work.

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 [5]. 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. Examples of design features 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. Our work represents an attempt to create a more user-centric abstraction of data flow information to support comprehension, but further development will need to address the cognitive overhead of creating this abstraction. The insights from Storey and colleagues will continue to be important as new tools are developed, automatic analyses are advanced, and reverse engineering workflows evolve.

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, perhaps into something like a knowledge transfer diagram [50]. 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. Research approaches that support the design of new decision-making support tools, such as work domain analysis, could support development of these externalizations.

6 Conclusion

In this case study, we describe using human factors methods to derive requirements for interprocedural data flow visualizations that can be used to quickly understand data flow elements and their relationships and influences. To generalize requirements produced through semi-structured interviews, and through task- and program-specific knowledge audits and cognitive walkthroughs, we developed a two-stage modified sorting task that helps extract commonalities in analyst mental models of data flow across different types of programs. We used the results from the modified sorting task, augmented with results from the cognitive task analysis activities, to derive a data flow taxonomy (requirements for representation). We assigned elements of the taxonomy to visual representations in an elaborated directed graph representation, and we used these generalized requirements to manually generate and evaluate data flow visualizations for binary programs with different vulnerabilities. Analysts were able to use the data flow visualizations to answer many critical questions about data flow. Our results indicate that our data flow taxonomy is promising as a mechanism for improving analyst understanding of data flow in binaries and for supporting efficient decision making during analysis. However, future work and evaluation will require integrating the visualization into existing analyst workflows.

7 Acknowledgements

The authors would like to thank Danny Loffredo, Todd Jones, Doug Ghormley, Andy Wilson, Tiemoko Ballo, Bryan Kennedy, and the many binary analysts who supported this work. Their suggestions and interactions have been invaluable.

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.

A Cognitive Walkthrough Setup

We selected the UNIX file utility version 5.10 [42][43], choosing from the AFL (American Fuzzy Lop) fuzzer bug-o-rama trophy case [44], a listing of vulnerabilities in real programs that were found by the program AFL-fuzz. We chose file version 5.10 because 1) the core processing library libmagic is vulnerable to CVE-2012-1571 [51][5]; 2) many functions in the library are involved in parsing input data from multiple sources; 3) a successful analysis requires understanding

[5] 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.

interprocedural data flow; 4) we had access to source code for both the vulnerable version 5.10 and the fixed version 5.11;[6] 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.

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.

B Modified Sorting Task Stage 1 Instructions

We would like to better understand how analysts categorize data flow elements when they are working on a VA (vulnerability analysis) or RE (reverse engineering) 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

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

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.

References

1. Somers, J.: The coming software apocalypse. The Atlantic (September 2017)
2. D'Silva, V., Payer, M., Song, D.: The correctness-security gap in compiler optimization. In: 2015 IEEE Security and Privacy Workshops. 73–87 (May 2015)
3. Song, J., Alves-Foss, J.: The darpa cyber grand challenge: A competitor's perspective. IEEE Security & Privacy **13**(6) 72–76 (2015)
4. Shoshitaishvili, Y., Weissbacher, M., Dresel, L., Salls, C., Wang, R., Kruegel, C., Vigna, G.: Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. CoRR **abs/1708.02749** (2017)
5. Storey, M.A.D., Fracchia, F.D., Müller, H.A.: Cognitive design elements to support the construction of a mental model during software exploration. J. Syst. Softw. **44**(3) 171–185 (January 1999)
6. Bainbridge, L.: Ironies of automation. Automatica **19** 775–779 (1983)
7. Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: 24th USENIX Security Symposium (USENIX Security 15), Washington, D.C., USENIX Association 177–192 (2015)
8. Kildall, G.A.: A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '73, New York, NY, USA, ACM 194–206 (1973)
9. Hex-Rays, S.: Ida pro disassembler. https://www.hex-rays.com/products/ida/ (2008)
10. Quist, D.A., Liebrock, L.M.: Visualizing compiled executables for malware analysis. In: 2009 6th International Workshop on Visualization for Cyber Security. 27–32 (Oct 2009)
11. Zynamics: Zynamics binnavi product description page. https://www.zynamics.com/binnavi.html
12. Rech, J., Schäfer, W.: Visual support of software engineers during development and maintenance. Volume 32., ACM 1–3 (2007)
13. Hardisty, Z.: Radia github page. https://github.com/zoebear/Radia
14. Reddy, N.H., Kim, J., Palepu, V.K., Jones, J.A.: Spider sense: Software-engineering, networked, system evaluation. In: Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on, IEEE 205–209 (2015)
15. Ball, T., Eick, S.G.: Visualizing program slices. In: Visual Languages, 1994. Proceedings., IEEE Symposium on, IEEE 288–295 (1994)

16. Eick, S., Steffen, J.L., Sumner, E.E.: Seesoft-a tool for visualizing line oriented software statistics. IEEE Transactions on Software Engineering **18**(11) 957–968 (1992)
17. Müller, H.A., Klashinsky, K.: Rigi-a system for programming-in-the-large. In: Proceedings of the 10th International Conference on Software Engineering. ICSE '88, Los Alamitos, CA, USA, IEEE Computer Society Press 80–86 (1988)
18. Storey, M..D., Muller, H.A.: Manipulating and documenting software structures using shrimp views. In: Proceedings of International Conference on Software Maintenance. 275–284 (Oct 1995)
19. Livadas, P.E., Alden, S.D.: A toolset for program understanding. In: [1993] IEEE Second Workshop on Program Comprehension. 110–118 (July 1993)
20. Brade, K., Guzdial, M., Steckel, M., Soloway, E.: Whorf: a visualization tool for software maintenance. In: Proceedings IEEE Workshop on Visual Languages. 148–154 (Sept 1992)
21. Baker, M.J., Eick, S.G.: Visualizing software systems. In: Proceedings of the 16th International Conference on Software Engineering. ICSE '94, Los Alamitos, CA, USA, IEEE Computer Society Press 59–67 (1994)
22. Orso, A., Jones, J.A., Harrold, M.J., Stasko, J.: Gammatella: visualization of program-execution data for deployed software. In: Proceedings. 26th International Conference on Software Engineering. 699–700 (May 2004)
23. Rajlich, V., Doran, J., Gudla, R.T.S.: Layered explanations of software: a methodology for program comprehension. In: Proceedings 1994 IEEE 3rd Workshop on Program Comprehension- WPC '94. 46–52 (Nov 1994)
24. LaToza, T.D., Myers, B.A.: Visualizing call graphs. In: Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on, IEEE 117–124 (2011)
25. Àlvarez, S.: The radare2 book. https://radare.gitbooks.io/radare2book/content/ (2009)
26. Vector35: Vector 35 binary ninja product description page. https://binary.ninja
27. Würthinger, T., Wimmer, C., Mössenböck, H.: 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, Berlin, Heidelberg, Springer-Verlag 193–196 (2008)
28. Deng, F., DiGiuseppe, N., Jones, J.A.: Constellation visualization: Augmenting program dependence with dynamic information. In: 2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). 1–8 (Sept 2011)
29. Sui, Y., Xue, J.: Svf: Interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th International Conference on Compiler Construction. CC 2016, New York, NY, USA, ACM 265–266 (2016)
30. Hoffswell, J., Satyanarayan, A., Heer, J.: Augmenting code with in situ visualizations to aid program understanding. In: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, ACM 532 (2018)
31. Yakdan, K., Eschweiler, S., Gerhards-Padilla, E., Smith, M.: No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In: NDSS. (2015)
32. Yakdan, K., Dechand, S., Gerhards-Padilla, E., Smith, M.: Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In: 2016 IEEE Symposium on Security and Privacy (SP). 158–177 (May 2016)

33. Hendrix, T.D., Cross, II, J.H., Barowski, L.A., Mathias, K.S.: Visual support for incremental abstraction and refinement in ada 95. Ada Lett. **XVIII**(6) 142–147 (November 1998)
34. Victor, B.: Learnable programming. http://worrydream.com (2012)
35. Victor, B.: The ladder of abstraction. http://worrydream.com (2011)
36. Victor, B.: A brief rant on the future of interaction design. http://worrydream.com (2011)
37. Fraze, D.: Computers and humans exploring software security (chess). https://www.darpa.mil/program/computers-and-humans-exploring-software-security (2018)
38. Mangal, R., Zhang, X., Nori, A.V., Naik, M.: A user-guided approach to program analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015, New York, NY, USA, ACM 462–473 (2015)
39. Sherwin, K.: Card sorting: uncover users' mental models for better information architecture. https://www.nngroup.com/articles/card-sorting-definition/
40. Manadhata, P.K., Tan, K.M.C., Maxion, R.A., Wing, J.M.: 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)
41. Militello, L., Hutton, R.: Applied cognitive task analysis (acta): A practitioner's toolkit for understanding cognitive task demands. Ergonomics **41**(12) 1618–41 (1998)
42. Darwin, I.: Original source packages for file utility. ftp://ftp.astron.com/pub/file/ (2012)
43. Darwin, I.: Maintained source for file utility. https://github.com/file/file
44. Zalewski, M.: American fuzzy lop: a security-oriented fuzzer. http://lcamtuf. coredump. cx/afl/(visited on 06/21/2017) (2010)
45. Fraze, D.: Cyber grand challenge (cgc). https://www.darpa.mil/program/cyber-grand-challeng (2016)
46. DARPA: Darpa cgc challenges source repository. https://github.com/CyberGrandChallenge/samples/tree/master/cqe-challenges (2016)
47. TrailOfBits: Darpa cgc challenges ported to standard os. https://github.com/trailofbits/cb-multios (2016)
48. Vicente, K.J.: Ecological interface design: Progress and challenges. Human factors **44**(01) 62–78 (2002)
49. Leger, M., Butler, K.M., Bueno, D., Crepeau, M., Cuellar, C., Godwin, A., Haass, M.J., Loffredo, T., Mangal, R., Matzen, L.E., Nguyen, V., Orso, A., Reedy, G., Stasko, J.T., Stites, M., Tuminaro, J., Wilson, A.T.: Creating an interprocedural analyst-oriented data flow representation for binary analysts (ciao). Technical Report SAND2018-14238, Sandia National Laboratories, Albuquerque, NM (December 2018)
50. Zhao, J., Glueck, M., Isenberg, P., Chevalier, F., Khan, A.: Supporting handoff in asynchronous collaborative sensemaking using knowledge-transfer graphs. IEEE Transactions on Visualization and Computer Graphics **24**(1) 340–350 (2018)
51. NIST: Cve 2012-1571. https://nvd.nist.gov/vuln/detail/CVE-2012-1571 (2012)