Modeling and Automating the Cyber Reverse Engineering Cognitive Process

Patrick P. Dudenhofer
patrickdude@cedarville.edu

Cedarville University
251 N. Main St. Cedarville, OH 45314

Abstract - Software reverse engineers (SREs) face a significant cognitive load when analyzing unknown binary artifacts for security vulnerabilities or malicious intent. The ability to automate or augment these complex reverse engineering tasks would provide a substantial benefit both for the training and productivity of binary analysis work. Such computational support requires a formal model of the reverse engineer's knowledge and operations but little research effort has been expended toward understanding the cognitive aspects of the software reverse engineering process. SREs often begin the reverse engineering process by exploring the binary executable's artifacts to discover information cues that correspond with their own abstract knowledge of the cyber security domain. Upon discovering an interesting information cue, the SREs integrate that new data into their working hypothesis of the program's behavior. As additional information cues are uncovered, these cues shape and elaborate upon SREs' current hypothesis of the software's purpose and also serve as indicators for additional exploration vectors. This paper proposes a cognitive model detailing the mental constructs and processes required for successfully completing a software reverse engineering task. The cognitive model described will facilitate accelerated development of automation and interface aids for complex binary analysis tasks.

Keywords

Software reverse engineering, binary analysis, cognitive modeling, human-computer interaction

1. INTRODUCTION

In the domain of cyber security, software reverse engineering (SRE) is a complex problem [1] and a critical task for cyber security professionals [2]. Software security threats do not come with step-by-step instructions on how to identify, prevent, or even understand what a piece of software is intended to accomplish [3]. Without the availability of high-level, easily-readable source code and documentation, reverse engineers must determine what the malware is programmed to accomplish by reconstructing programming logic from the binary representation - the ones and zeros - of the executable files. Software reverse engineers (specifically, binary analysts) must be able to identify key software constructs and features from very low-level assembly code instructions and mentally reconstruct a program's control structures to determine the software's purpose [4]. The time and training constraints needed to develop proficient software reverse engineers can be partially addressed by designing automation and interfaces that support and augment reverse engineers as they analyze unknown software. Building a solid theoretical foundation describing the tasks and processes used by professional reverse engineers will inform and direct the development of advanced human-computer interfaces (HCI) and automated agents to aid cognitively complex software reverse engineering tasks.

1.1 Motivation

The number and sophistication of cyber-attacks has increased markedly in the past few years [5]. Managing all the incoming attacks is difficult and cyber defense professionals struggle just to "keep in the same place" [6]. Beyond increasing the workforce, cutting-edge software tools are desperately needed to stay abreast of emerging cyber security threats. Automated binary analysis agents and improved human-computer interfaces will speed up the process of standard classification and analysis techniques, decrease the learning curve for new cyber-security professionals, and increase the identification of and analysis throughput for potential software vulnerabilities and their corresponding exploits [7].

1.2 Research Problem

The term “software reverse engineering” refers to the activities involved in reconstructing a useful and meaningful representation of a program [8]. One must work backward (or reverse engineer) to undo the compilation processes that originally created the executable program from its source code to learn the intended behavior of the software under analysis [9]. Comprehending a program from its compiled binary file involves abstracting low-level data representations into high-level concepts [10] and using the resulting mental models of concepts, programming plans, and control flow representations to synthesize both environment and learned information together into a coherent model of the program [11]. Investigating unknown software without executing the code is called static analysis. Alternatively, running an executable to determine its behavior is known as dynamic analysis. In this paper the term "binary analysis" is used to refer to the process of program comprehension using visual inspection only of software artifacts (primarily assembly code representations) without actively running the software being analyzed. Binary analysis is a significant subset of the tasks typically performed when SREs attempt to understand an unknown program's purpose and behavior. Program comprehension, particularly with only a binary representation, is a cognitively challenging activity and knowledge-intensive task [12]. Both human-computer interfaces and automated agents should be designed to support the reverse engineer's cognitive processes allowing the engineer to focus on the tasks that are best performed by a human.

Though many tools have been developed to automate some of the tasks involved in binary analysis, most have not explicitly taken into consideration the complex cognitive aspects of reverse engineering. Software reverse engineers often create their own ad-hoc software tools [13] without thorough consideration for which tasks would actually benefit from automation or interface support. Designing automation aids that facilitate more effective reverse engineering techniques requires a knowledge of which tasks are cognitively difficult and why. Researchers have described reverse engineering tasks using formal ontologies [14] while others have characterized the reverse

engineer's goals and processes [15]. Other research has captured analysts' cognitive processes during dynamic cyber-attacks [16] and proposed a mathematical model of human-computer interactions in a binary analysis task [17]. Any automation and HCI advancements will likely be ineffective without task and cognitive models that reflect an expert reverse engineer's cognitive process [18]. Ultimately, researchers need to know how to efficiently allocate and apply automation within the software reverse engineering domain.

1.3 Research Direction

The primary question to be addressed here is "How can we leverage the cognitive process of an expert software reverse engineer to support the training of novice binary analysts via automated software agents?" With an SRE cognitive model, research and development can proceed to augment the binary analysis process with automated agents in order to offload cognitively difficult parts of the task.

This paper presents a descriptive framework of the binary analysis cognitive process along with a prototype analysis tool illustrating the cognitive model's effectiveness and applicability. Consequently, researchers will be able to focus their automation and interface design efforts in areas that will most benefit software reverse engineers using the very patterns human experts use to complete their work.

2. LITERATURE REVIEW

Program (or software) comprehension is a human-intensive process requiring the extraction of sufficient information from software artifacts or systems via analysis and intuition to accomplish a given task. Missing or out-of-date documentation increases the challenge of deriving the necessary knowledge from the software. When a compiled executable is the only available artifact for observation, comprehension is even more difficult for the reverse engineer. Much of the original programmer's thinking is lost in the translation from program concept to executable code [17].

2.1 Program Comprehension

Program "comprehension involves the assignment of meaning to a particular program," requiring specialized knowledge to perform successfully [20]. The general approach in most program comprehension tasks is to consider the program as a text that parallels natural language texts such as instruction manuals. Reading program code is a preferred option for comprehension, rather than documentation or execution, as it provides a factual and true representation of the program [21]. The perception (and mental model) of the text is affected both by analysts' past experiences and by how analysts break the text into structural segments while organizing their mental representational framework.

Fix, et al. evaluated the differences between groups of experts and novices attempting to comprehend a program [22]. They found experts extracted many different kinds of program information and integrated that knowledge with their mental representations while novices did not exhibit as effective mental representations of the program. Some of the skills the research highlighted for creating a successful mental representation included skill in recognizing basic recurring patterns, in understanding program structure, and in distinguishing links between program modules. Developing a good mental representation is essential for effectively comprehending program behavior.

2.2 Information Cues

In researching software maintenance tasks, Ko, et al. proposed a model of program comprehension that reflects a process of searching, relating, and collecting relevant information [24]. Developers form perceptions of relevance from clear, representative cues in the environment. In the cycle of relating and connecting fragments of information together, when no more relevant cues exist the developer stops that particular thread of relating and begins correlating other fragments of information. Once enough information fragments have been collected to implement a solution to the task, the developer stops performing the cycle of searching, relating, and collecting and proceeds to use the information he has gathered.

Kulkarni and Varma also investigated developers facing the problem of comprehending unfamiliar programs [23]. They discovered experts follow an "information scent" and place a perceived value on information based on cues to facilitate their navigation and searching in the program. The authors posit all developers follow the same information scents while comprehending an unfamiliar code and eventually develop a similar perception of program behavior regardless of expertise. They also found that developers used "an abstractive rather than extractive approach" to gleaning information from the source code. Experts, as they "forage" for information by means of identifying cues, overcome information overload by relying on cues to assist the development of a mental model of program behavior. Identifying and understanding both the explicit and implicit cues can aid program comprehension and mental model development.

Studies have shown developers can spend up to 35 percent of their time navigating through software code and associated artifacts [24]. Lawrance et al. found programmers performing a debugging task acted in a manner consistent with information foraging theory by following "scent" cues to navigate through the source code [25]. This theory led the authors to state that software engineering tools can and should support "scent following" mechanisms. Developers should identify informational cues associated with the information domain and the tools' supported tasks. Those cues will empower programmers to navigate more quickly through code toward their goal.

2.3 Software Reverse Engineering

Software reverse engineering is a broad term involving myriad methodologies and tools used to extract information and knowledge from software artifacts. Chikofsky and Cross define software reverse engineering as “the process of analyzing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction” [10]. Reverse engineering is a process of observation and examination, not alteration. As such, software reverse engineering is primarily an observation process to determine construction and

usage of software - performed in wide range of contexts and for many different purposes.

LaToza et al. suggest "expert developers can easily navigate in a complex code because they seek precise evidence of relevance to the task when faced with the decision to investigate an unfamiliar code" [26]. Experts perform better in software reverse engineering tasks because they have access to knowledge that novices do not [23]. Reverse engineers need to be able to identify relevant evidence and artifacts to inform future actions and analysis.

The binary analyst uses several different kinds of artifacts in the software reverse engineering process. Tilley classifies the artifact categories as data (factual information), knowledge (the sum of what is known - including relationships between data), and information (communicated knowledge) [27]. In terms of data (factual information) the primary artifact referenced in the SRE process is the binary executable file under analysis. This executable code is disassembled into thousands of assembly language instructions. These instructions accurately model how the program behaves on a computer's processor, but they do not always directly reflect the high-level abstract concepts and constructs many programmers use to describe how a program works. The process of compiling a program from a high-level language into a binary executable strips much of the semantic information and structure away from the assembly code representation making the program all the more difficult to comprehend. A software reverse engineer is almost always restricted to assembly code only in their analysis tasks [19]. Beyond an understanding of assembly languages, reverse engineers need to have a working knowledge of operating systems and associated system calls, memory management, and vulnerability exploitation and defense.

In gathering information, SREs use a wide range of tools, some commercial and others created in an ad hoc manner as specific needs arise. Each tool provides a unique environment for the representation of information pertaining to the observed binary file - potentially adding additional cognitive challenges for the reverse engineer in comprehending the program.

2.4 Abduction in Reverse Engineering

Abductive logic is used when a person who is assessing a situation is surprised by unexpected events or information and subsequently tries make sense of things though further inquiries [28]. Abduction is a form of logical inference that begins with a set of observations and then attempts to find the simplest and most likely explanation. A surprised person utilizes abduction to develop new hypotheses about a situation after recognizing the current situation invalidates a previously-held hypothesis [19]. The observer can continue to confirm or disprove new hypotheses though additional inquiries and use of deductive and inductive reasoning. These additional actions can refine the observer's knowledge and hypotheses of the situation [28]. Weigand states an expert's experience enables them to recognize situations encountered in the past and establish the relative importance of each perceptual cue [19].

Applied to reverse engineering, abduction provides a "generative means of inference" essential to an exploration of code requiring "non-deductive and non-inductive hypothesis generation" [19]. The reverse engineer uses abduction to handle falsified hypotheses and observational surprises, shift their plans of action, and modify their working hypotheses. Reverse engineers typically select hypotheses that are models of previously encountered behaviors or constructs. Detecting when a working hypothesis is proven incorrect is a critical aspect for a successful abductive inquiry in reverse engineering [11].

2.5 Sensemaking in Reverse Engineering

Sensemaking, a process where people attempt to understand complex situations to make reasonable decisions, is enabled by abductive inference [28]. Bryant, et al. make the case that software reverse engineering is a type of sensemaking process [11]. Similar to other sensemaking processes (such as the scientific method), reverse engineering requires information discovery, development of a mental model (hypothesis) using the discovered data, and integration of the mental model with additional artifact data and the analyst's background knowledge.

2.6 Looking Forward

Software reverse engineering can been described in several different, yet compatible ways. Generally it is described as a process of abstraction from low-level representations to high-level concepts. Others view it as a set of interconnected analysis tasks. As previously noted, Tilley describes software reverse engineering as transforming software artifacts into a mental model through pattern recognition to create "abstract system representations" [27]. Each of these descriptions capture unique facets of what makes SRE a challenging task.

While software reverse engineering has been described a few times in the research literature using cognitive models and related concepts, very little research has been focused on the binary analysis task itself or on how the cognitive models can be leveraged by automated software agents to support and aid novices in the field. This is a large gap in current cyber security research. The question remains: how can the concepts of abductive reasoning, sense-making, information scent, cues, cognitive modeling, and program comprehension be used to effectively support the novice software reverse engineer in static binary analysis tasks?

3. SRE COGNITIVE MODEL

Before effective reverse engineering automation aids can be created, researchers need to understand the tasks software reverse engineers perform in cyber security and the meta-process by which they accomplish those tasks. Once the process and tasks performed by reverse engineers are adequately described computational agents can model some of the same activities.

The software reverse engineering process requires binary analysts to integrate detailed background knowledge with newly discovered contextual information in order to extract an executable program's specifications and intentions. As noted before, comprehending the purpose and workings of a piece of binary software, without the aid of documentation and source files, is a difficult task. Binary analysis requires a broad understanding of computer

hardware, operating systems, programming languages, and software vulnerabilities. Current software analysis tools provide limited cognitive support to software reverse engineers either in the form of helpful interfaces or in automated software agents. A cognitive model of reverse engineering will enable researchers and developers to better understand the mental processes used in binary analysis of software.

Given expert reverse engineers employ a type of sensemaking process for comprehending assembly code, software developers can build pieces of an automated agent to support that process. Such an agent will need to mimic at least a few of the sensemaking steps that a binary analyst uses - otherwise it's unlikely to support the cognitive process of the reverse engineer. Creating a fully autonomous agent would prove difficult, but an automated aid that augments the SRE task and cognitively supports the binary analyst is possible.

3.1 Overview of the CURE Cognitive Model

The CURE (Cognitive Understanding of Reverse Engineering) model of SRE cognition (Figure 1) describes how so-called "interesting" properties or cues of a binary executable are elicited via a series of iterative experiments by an analyst. In general, the process of binary analysis consists of transforming information from a low-level language representation of an executable to a high-level mental model. To do so, binary analysts typically explore an unknown piece of software for interesting informational cues that they have previously memorized and now recall from their background knowledge of the SRE domain. Once such a cue is found the analyst creates one or more behavior hypotheses for the program based on their current knowledge of abstract program behavior concepts. The binary is then analyzed, through a series of iterations, for additional information that would support or negate the analyst's hypothesis until some confidence level for the veracity of the hypothesis can be achieved. The number of iterations required depends on the level of understanding needed concerning the binary. The desired level of understanding ranges from simple categorization to complete re-creation of a binary file's functionality. Simple categorization would require only a few

iterations of the locate-elaborate loop after identifying an initial information cue. However, functional re-creation of a binary artifact would require many, many iterations of the proposed cognitive algorithm.

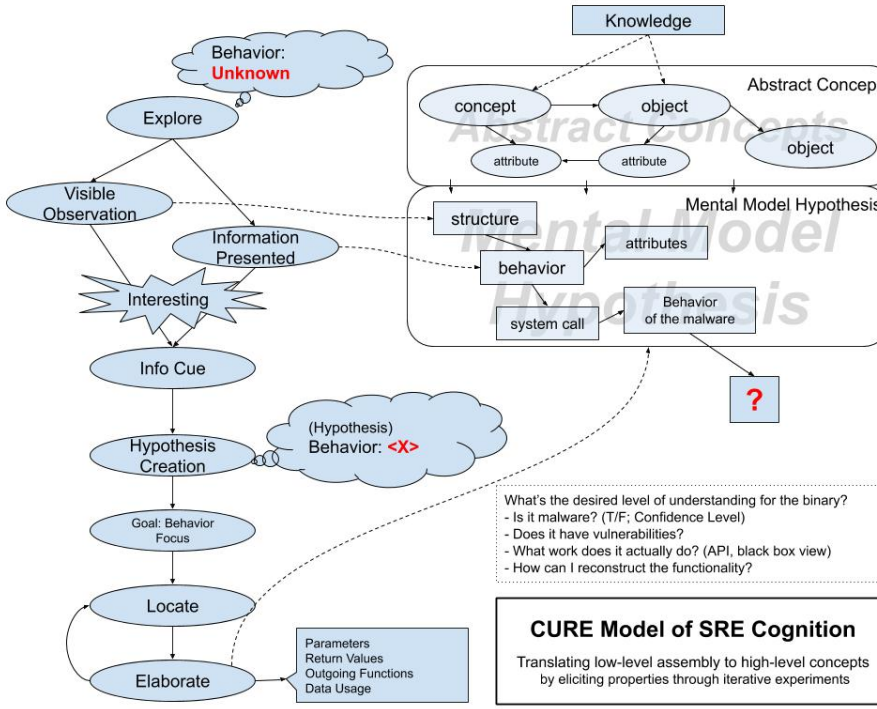

Figure 1. CURE Model of SRE Cognition

This model demonstrates a type of abductive reasoning - inference to the best explanation. Just because a set of observations have been made and support a certain conclusion (i.e. the hypothesis), they do not necessarily guarantee that particular conclusion.

3.1.1 Exploration

As the binary analyst begins the software reverse engineering process, only a vague idea of the binary executable's behavior may exist as the analyst's mental model. The analyst initially doesn't know the program's purpose or functionality and begins exploring the executable's external artifacts and its

assembly code for information cues. Until some sort of "interesting" cue is found, the analyst continues searching for pertinent structural or behavioral information.

3.1.2 Recognition of an Interesting Information Cue

Recognition of an information cue involves the analyst making a mental connection between his abstract background knowledge of program behavior (and related attributes) with a specific software artifact or chunk of assembly code. This is the "ah ha" moment after which the analyst can narrow his search space based on the implications of that artifact's presence within the program. Additional cues may be found as well - and so working memory becomes a stack of "interesting" information cues used to generate new hypotheses or support existing ones. These additional cues may interrupt the process or be returned to later in the process to further enable understanding of the binary's functionality.

3.1.3 Hypothesis Creation

After an information cue is discovered, the analyst creates a behavioral hypothesis for the still unknown software. This labeling of potential behavior draws on the analyst's previous knowledge of the abstract concepts in the field of computer science and software security. Additional artifacts from the binary will end up supporting or refuting the chosen behavior hypothesis of the analyst.

3.1.4 Locate-Elaborate Loop

With a hypothesis created, the analyst begins a second process of exploration with the goal of evaluating the hypothesis. The abstract concepts from the background knowledge of the analyst are slowly filled in with concrete artifacts as part of his mental model of the program. As additional information is gathered, either the required pieces of the mental model are located to support the analyst's hypothesis or else the hypothesis cannot be supported from the available artifacts and assembly code. Depending on the required level of understanding for the task at large, supporting information such as parameters, return values, outgoing functions, and data usage can be

elaborated and used to populate the analyst's mental model of the software's functionality.

If an artifact is found that does not match or fit within the analyst's current hypothesis, the current mental model must be modified or discarded. The discovery may either signal the end of the task (since a potential hypothesized behavior has been eliminated from consideration) or indicate the need to start from scratch with a new hypothesis that can accommodate the new information along with all the previously discovered structures and behaviors of the binary executable.

4. CURE ASSISTANT SRE AUTOMATED AID

Leveraging concepts from the CURE cognitive model, a software program called CURE Assistant [29] was developed to help novice reverse engineers recognize important and interesting artifact cues while performing binary analysis tasks (Figure 2). An undergraduate computer science senior design team at Cedarville University under the direction of the author implemented CURE Assistant to be an automated aid supporting program comprehension and cognitive model development of unknown binary files [30].

CURE Assistant was designed to support a novice in the discovery of information cues and artifacts that would normally be quickly recognized by an expert malware analyst. It is not a fully automated software agent. It is a python program that executes alongside radare2 [31], a command-line-based software reversing framework, to provide “interesting” artifact (i.e. cue) suggestions to the analyst while filling in gaps of their program comprehension and domain knowledge. Analysts can easily perform further investigations on suggested artifact cues within the standard interface of radare2 by using CURE Assistant’s interface linkages to the reversing framework.

CURE Assistant searches for interesting artifacts (i.e. snippets of assembly code) in the binary and suggests potential program behaviors based on the existence or non-existence of those artifacts. By providing both relevant artifacts and potential behavior descriptions that utilize those artifacts, CURE

Assistant supports the exploration, the information cue recognition, and hypothesis creation components of the CURE cognitive model.

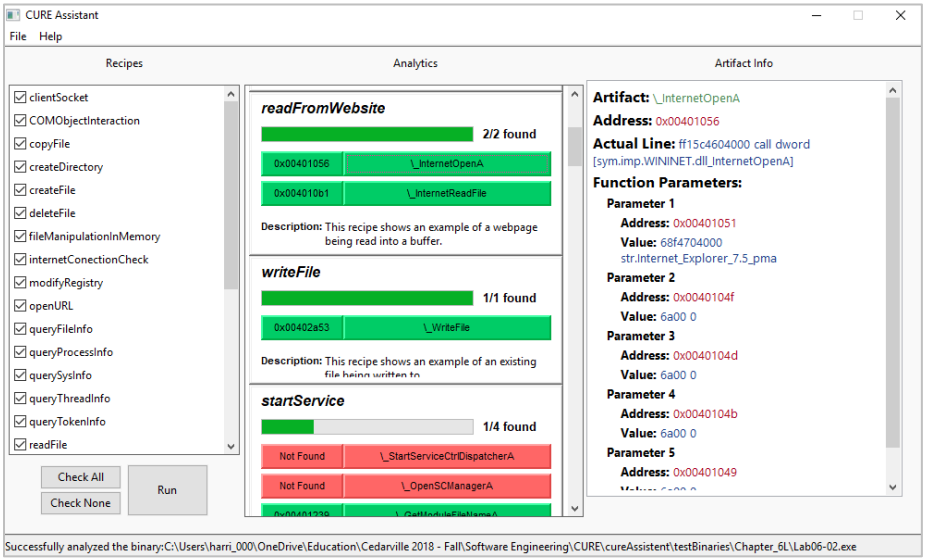

Figure 2. CURE Assistant Interface

The binary analyst can easily navigate directly to the location of CURE Assistant's discovered cues within the assembly code to view the artifact in its context among the rest of the code. CURE Assistant also provides an overview of potential behaviors for the binary under scrutiny based on the binary artifacts found. Each behavior is called a recipe within CURE Assistant and each recipe may have one or more artifacts associated with it. The list of matched and partially-matched behaviors enhances the analyst's decision-making capability by providing several potential behavior hypotheses to consider and further investigate. A novice practitioner may not have enough background knowledge in the SRE domain to recognize the cues within the assembly code or associate the cues with categories of program behavior. CURE Assistant augments the novice binary analyst's cognitive process via its identification of artifacts and association of those artifacts with known program behaviors. By selecting a recipe's artifact further information about the artifact, such as parameter values

for a function call, is displayed. In addition, the cognitive load of the novice is decreased as the complete list of interesting cues is available to be pursued later without the analyst having to keep track of each of the cues and their locations.

Each recipe of program behavior is stored as a JSON (JavaScript Object Notation) formatted file. Additional recipes with their associated artifact descriptions can be easily created and added to CURE Assistant's list of searchable behaviors. A user may select or unselect specific program behavior recipes to look for as they analyze a particular binary. Some of the program behaviors to choose from include Windows registry modification, service and thread manipulation, network connectivity, and Windows Native API library usage.

Initial use of CURE Assistant has shown it to be helpful in highlighting important cues and relating them together with program behavior concepts for novice binary analysts. Additional expert validation of the CURE cognitive model and further study of CURE Assistant in both training and classroom settings is planned.

5. CONCLUSION

Software reverse engineering is a cognitively challenging task. Reverse engineers need advanced automation support created to complete their critical work in a timely and effective manner. CURE Assistant illustrates the potential benefits to novice binary analysts when they have access to training and tools that reflect and augment the cognitive processes used by professionals. Utilizing a software reverse engineering cognitive model, such as the one proposed in this paper, will bring significantly enhanced system automation and interfaces for cyber security professionals as they extract program intent from complex binary executable files.

REFERENCES

[1] United States Air Force Scientific Advisory Board, "Report on Implications of Cyber Warfare, Volume 1: Executive Summary and Annotated Brief," 2007.

[2] C. Treude *et al.*, "An exploratory study of software reverse engineering in a security context," *2013 20th Working Conference on Reverse Engineering (WCRE)*, vol. 0, pp. 184–188, 2011.

[3] D. Song *et al.*, "Bitblaze: A new approach to computer security via binary analysis," in *In Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. Springer, 2008, pp. 1–25.

[4] G. Gannod and B. Cheng, "A formal approach for reverse engineering: a case study," in *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*, Oct 1999, pp. 100–111.

[5] S. Donovan and A. Scott, "Cybersecurity strategy and implementation plan (csip) for the federal civilian government (m-16-04)," Office of Management and Budget, 2015. [Online]. Available: https://www.hsdl.org/?abstract&did=788143

[6] A. Tanenbaum and H. Bos, Modern Operating Systems, 4th ed. Pearson Prentice-Hall, 2014.

[7] A. Smith *et al.*, "The role of expert systems in reverse code engineering tasks," in *9th International Conference on Cyber Warfare and Security (ICCWS 2014)*,West Lafayette, IN, March 2014.

[8] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *13th International Workshop on Program Comprehension (IWPC'05)*. IEEE, 2005, pp. 181–191.

[9] C. Eagle, *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[10] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13–17, Jan 1990.

[11] A. Bryant *et al.*, "Software reverse engineering as a sensemaking task," *Journal of Information Assurance and Security*, vol. 6, no. 6, pp. 483–494, 2012.

[12] W. Maalej *et al.*, "On the comprehension of program comprehension," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014.

[13] S. Heelan, “Vulnerability detection systems: Think cyborg, not robot,” IEEE Security & Privacy, vol. 9, no. 3, pp. 74–77, 2011.

[14] B. Chandrasekaran, J. R. Josephson, and V. R. Benjamins, “What are ontologies, and why do we need them?” *IEEE Intelligent systems*, no. 1, pp. 20–26, 1999.

[15] A. Bryant, “Understanding how reverse engineers make sense of programs from assembly language representations,” Ph.D. dissertation, Air Force Institute of Technology, 2012.

[16] C. Zhong *et al*., “Arsca: a computer tool for tracing the cognitive processes of cyber-attack analysis,” in *2015 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision.* IEEE, 2015, pp. 165–171.

[17] Z. Sisco, P. Dudenhofer, and A. Bryant, “Modeling information flow for an autonomous agent to support reverse engineering work,” *The Journal of Defense Modeling and Simulation*, vol. 14, no. 3, pp. 245–256, 2017.

[18] P. Dudenhofer and A. Bryant, “Establishing a cognitive understanding of cyber reverse engineering tasks,” *12th International Conference on Cyber Warfare and Security (ICCWS)*, Dayton, OH, 2017.

[19] K. Weigand and R. Hartung, “Abduction’s role in reverse engineering software,” in *Aerospace and Electronics Conference (NAECON), 2012 IEEE National*, July 2012, pp. 57–62.

[20] N. Pennington, “Stimulus structures and mental representations in expert comprehension of computer programs.” *Cognitive Psychology*, vol. 19, no. 3, pp. 295 – 341, 1987.

[21] T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: A study of developer work habits,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 492–501.

[22] V. Fix, S. Wiedenbeck, and J. Scholtz, “Mental representations of programs by novices and experts,” in *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*, ser. CHI ’93. New York, NY, USA: ACM, 1993, pp. 74–79.

[23] N. Kulkarni and V. Varma, "Supporting comprehension of unfamiliar programs by modeling an expert's perception," in *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE 2014. New York, NY, USA: ACM, 2014, pp. 19–24.

[24] A. Ko *et al.*, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *Software Engineering, IEEE Transactions on*, vol. 32, no. 12, pp. 971–987, Dec 2006.

[25] J. Lawrance *et al.*, "How programmers debug, revisited: An information foraging theory perspective," *Software Engineering, IEEE Transactions on*, vol. 39, no. 2, pp. 197–215, 2013.

[26] T. D. LaToza *et al.*, "Program comprehension as fact finding," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 361–370.

[27] S. R. Tilley, "The canonical activities of reverse engineering," *Annals of Software Engineering*, vol. 9, no. 1-2, pp. 249–271, 2000.

[28] S. A. Douglass and S. Mittal, *A Framework for Modeling and Simulation of the Artificial*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 271–317.

[29] Cedarville University, CURE Assistant, 2019, [Online]. GitHub respository, https://github.com/CedarvilleCS/CURE-Assistant

[30] B. LaChance, F. Trautmann, J. Tiberg and N. Harris, "C.U.R.E. Assistant Reverse Engineering Educational Software," Poster session presented at the Cedarville University Research & Scholarship Symposium, Cedarville, OH, April 3, 2019, [Online]. https://digitalcommons.cedarville.edu/research_scholarship_symposium/2019/poster_presentations/17/

[31] radare2, [Online]. GitHub respository, https://github.com/radare/radare2