UNDERSTANDING HOW REVERSE ENGINEERS MAKE SENSE OF PROGRAMS FROM ASSEMBLY LANGUAGE REPRESENTATIONS

DISSERTATION

Adam R. Bryant, Civilian, USAF

AFIT/DCS/ENG/12-01

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright Patterson Air Force Base, Ohio

The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/DCS/ENG/12-01

UNDERSTANDING HOW REVERSE ENGINEERS MAKE SENSE OF PROGRAMS FROM ASSEMBLY LANGUAGE REPRESENTATIONS

DISSERTATION

Presented to the Faculty of the
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

Adam R. Bryant, B.S., M.S., M.S.
Civilian, USAF

March, 2012

AFIT/DCS/ENG/12-01

UNDERSTANDING HOW REVERSE ENGINEERS MAKE SENSE OF PROGRAMS FROM ASSEMBLY LANGUAGE REPRESENTATIONS

Adam R. Bryant, B.S., M.S., M.S.

Civilian, USAF

Approved:

//SIGNED// 17 Feb 2012
Dr. Robert F. Mills Date
Committee Chairman

//SIGNED// 17 Feb 2012
Dr. Gilbert L. Peterson Date
Committee Member

//SIGNED// 17 Feb 2012
Dr. Michael R. Grimaila Date
Committee Member

Accepted:

//SIGNED//
Dr. Marlin U. Thomas
Dean, Graduate School of Engineering and Management

Acknowledgements

Thank you to the subject matter experts and participants who were involved with the studies described in this dissertation. Thank you to my research committee for guidance as I went through the process of learning research, and my supervisors, which have provided me the opportunity to earn a PhD.

Finally, I would like to thank my wife and each of my children, who have all been incredibly patient as I have tried to balance family time, work, and my studies for well over a decade now. My wife, particularly, has provided immeasurable support as I have been immersed in this problem for several years. I could never thank you enough.

Adam R. Bryant

Table of Contents

Page

Page

List of Figures

List of Tables

AFIT/DCS/ENG/12-01

Abstract

This dissertation presents a theory of how reverse engineers make sense of executable programs. The theory describes the process of sensemaking in reverse engineering as a goal-directed planning-based search activity, in which the reverse engineer interacts with an executable program using reverse engineering tools in order to construct a mental model and working understanding of the functionality of the program. This theory is developed through a case study, semi-structured interviews with expert reverse engineers, and observations of reverse engineers performing a reverse engineering task. The theory of sensemaking in reverse engineering is a step toward building autonomy into analysis tools so they will be able to discover vulnerabilities in complex software-based systems and analyze executable programs to determine whether those programs contain undocumented malicious functionality and should not be trusted.

UNDERSTANDING HOW REVERSE ENGINEERS MAKE SENSE OF PROGRAMS FROM ASSEMBLY LANGUAGE REPRESENTATIONS

1. Introduction

1.1 Overview

This dissertation develops a theory of the conceptual and procedural aspects involved with how reverse engineers make sense of executable programs. Software reverse engineering is a complex set of tasks which require a person to understand the structure and functionality of a program from its assembly language representation, typically without having access to the program's source code. This dissertation describes the reverse engineering process as a type of "sensemaking," in which a person combines reasoning and information foraging behaviors to develop a mental model of the program.

The structure of knowledge elements used in making sense of executable programs are elicited from a case study, interviews with subject matter experts, and observational studies with software reverse engineers. The dissertation describes the goals and procedures used in making sense of programs as well as the families of concept knowledge which make up reverse engineers' mental models of the programs they investigate. The results from this research can be used to improve reverse engineering tools, to develop training requirements for reverse engineers, and to develop robust computational models of human comprehension in complex tasks where sensemaking is required.

This dissertation's novel research contributions are: a conceptualization of the reverse engineering processes as an example of sensemaking, organization of procedural and conceptual knowledge aspects involved with performing reverse engineering tasks, and a theory of sensemaking in reverse engineering that draws upon a sensemaking process elicited from observations of people reverse engineering executable programs.

1.2 Motivation

Cyberspace has been described as the "control system . . . of hundreds of thousands of interconnected computers, servers, routers, switches, and fiber optic cables that make the nation's critical infrastructures work" [153]. Organizations put a great deal of trust in cyberspace in order to carry out their missions, but they cannot currently assess the trustworthiness of the systems on which their missions rely [95]. In order to manage their risks, organizations must be able to assess the vulnerability of their systems to attacks and system failures [32]. The current lack of technical capabilities to quickly assess these vulnerabilities makes it difficult for organizations to effectively manage or address the risks that come with operating in cyberspace [31].

The President of the United States published a National Cyberspace Strategy [153], which describes the need to "continuously assess threats and vulnerabilities to federal cyber systems" through the "development of tactical and strategic analysis of cyber attacks and vulnerability assessments." Automating "cyber vulnerability assessments and reactions" is one of the United States Air Force's key capability thrusts in the next twenty years [188]. However, approaches to automate analysis tasks have met with difficulty because of numerous unaddressed challenges in understanding the human comprehension processes involved in assessing complex systems [30]. Automating cyber vulnerability assessments and reactions in the future requires an understanding of how people currently perform these tasks [30].

Vulnerability assessments are activities that involve analyzing complex systems to discover vulnerabilities. *Vulnerabilities* are system flaws that create the opportunity for someone to gain unauthorized access and control of a system [71]. Vulnerability assessment activities seek to give analysts an understanding of how attackers can leverage system flaws to create the opportunity to do harm on a computer system or network.

Attackers often gain access to systems through *exploits*, which are programs, data, sequences of instructions, or sequences of commands that take advantage of vulnerabilities to perform tasks desired by an attacker [71]. Attackers may use *malicious software* to gain access to a vulnerable system, to deliver an exploit, or to retain access of a compromised system [180]. Malicious software, also called *malware*, refers to "instructions that run on a computer and make the system do something than an attacker wants it to do" [180]. This includes any type of executable software "designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and other abusive behavior" [133].

In many organizations, the assessment of vulnerabilities in software-based systems is performed by software reverse engineers [65]. A *reverse engineer* is someone who investigates a complex system through analysis and observation, and *reverse engineering* is the term used to describe the process by which they do this [65]. *Software reverse engineers* analyze and assess software, or computer programs, to understand what the programs do and how they work [65, 42]. Many software reverse engineers specialize in analyzing malicious software from assembly language representations [186] and seek to discover exploitable vulnerabilities in software programs [92]. In this dissertation, software reverse engineering describes tasks involved in the comprehension of programs from assembly language representations[1].

Researchers have begun work in automating some of the many challenging tasks involved with discovering vulnerabilities and analyzing malicious software such as improving dynamic analysis [16, 15], finding higher-level templates of malicious behavior

[1] This definition differs from the definition of "software reverse engineering" from the program comprehension community, which views it as a set of activities to create mental models and design documents from source code[192]. As of this writing, a search for "reverse engineer" on the employment website *Monster.com* produces 30 jobs on the front page, all 30 of which are for people with skills in analyzing a program from binary (assembly language) representations.

[43, 44], correctly disassembling executable programs [108], and creating better ways of crafting control flow for input to a program [183]. Recently, visualization techniques have been developed to explore how different representations can enable more effective and efficient reverse engineering [82, 156]. But these techniques have not addressed the cognitive aspects of reverse engineering executable programs, which will be essential to advancing the autonomy of vulnerability assessment technologies [30].

The theory of sensemaking presented in this dissertation addresses these challenges. This theory was developed from an integrated series of studies that investigate how reverse engineers make sense of executable programs. The studies described the components of reverse engineering situations, how reverse engineers think about the programs they analyze, and how they actually make sense of and use information from reverse engineering tools. In the next section, the research problem of the dissertation is described. After that, the contributions of the research are outlined, and an overall outline of the dissertation is provided.

1.3 Research Problem

Some reverse engineers are able to quickly assess complex software-based systems for vulnerabilities and exploit them in ways that automated technologies cannot [26]. However, very little is understood about the cognitive processes that enable these reverse engineers to comprehend the essential elements of the program so quickly.

The Air Force's 2010 - 2030 Technology Horizons report depicts future investments in key technology areas such as "human behavior modeling," "human-machine interfaces," and "information fusion and understanding" as vital to achieving the ability to automate vulnerability assessments and reactions [188].

To achieve part of that strategy, this dissertation describes an integrated sequence of studies conducted to understand the human comprehension processes in understanding executable programs. The studies in this dissertation culminate in a theory about how people make sense of programs to construct a *mental model* of the parts and properties of a disassembled program. This mental model theory of understanding executable programs produces a number of additional research questions and identifies ways in which human information processing can be automated or augmented by future developments in software reverse engineering tools. This dissertation describes the questions, rationale, and detail of the studies which went into the development of the theory.

1.4 Approach to Solving the Problem

An assumption that guides the research is that a person can be said to have comprehended a situation by "making sense" of the situation [114]. The person is said to have made sense of the situation after successfully developing a "situation model" [198] which leads the person to effective behavior in the task. In the cognitive science literature, a situation model (alternately referred to as a "mental model" of a situation) is composed of the objects, actors, events, causal relations, and temporal relations that are involved in a situation [189]. A person's situation model is also informed by the concepts that are relevant to the aspects of the situation [161, 101, 197].

The research question addressed in this dissertation is: "How do reverse engineers make sense of executable programs while reverse engineering?" Particularly:

Q1: What conceptual elements are involved with making sense of an executable program during reverse engineering?

Q2: What procedural elements are involved?

Answers to these questions are found by studying the tasks involved [111], the automation used [68, 170], and the knowledge required to perform the work [90, 55, 93]. Data to answer the questions can be collected by studying the situational characteristics of a reverse engineering task [70, 111, 67], talking to experts [209], and observing people while they perform the tasks [52, 72]. This dissertation answers the research questions by collecting and analyzing data gathered through three complementary methodologies:

1. A case study [211] to explore the situational aspects involved in performing a software reverse engineering task,
2. Semi-structured interviews [209] with subject matter expert software reverse engineers to elicit concepts and processes involved in reverse engineering, and
3. An observational study in which concurrent verbal data [72] was elicited from participants as they performed software reverse engineering tasks.

The case study (Chapter 4) presents an analysis of how the task environment, characteristics of the task, and cognitive activities interplay in a reverse engineering task performed by the researcher. The task was to analyze the disassembled instructions of a program in a user-level debugger to discover how and where the program generates a splash screen window and to modify the program so the window does not appear. The case study was used to develop intuitions about how conceptual and procedural knowledge are used in software reverse engineering and to better understand problem solving in a reverse engineering task.

The semi-structured interviews conducted with subject matter experts are described in Chapter 5. The content of the semi-structured interview was developed from the results of the case study and involved questions about the nature of software reverse engineering work, the goals, activities, decisions, and information cues reverse engineers use in reverse engineering software, and discussions of tacit knowledge used in reverse engineering. The subject matter expert interviews were conducted and analyzed to gain an understanding of the different types of tasks involved in reverse engineering programs and to identify how these tasks relate to each other. The interview data was also used to establish the appropriate terminology, determine the tools used by reverse engineers, and to identify types of tasks that would be helpful to use in a subsequent observational study.

The observational study is described in Chapter 6. The observational study was designed based on aspects identified from the subject matter expert interviews as being relevant to understanding how reverse engineers make sense of programs from assembly language representations. The task, tools used, and types of data collected in the observational study were determined from the results of the subject matter expert interviews.

In the observational study, verbal protocols were collected as participants performed software reverse engineering tasks. Participants used a debugger to understand an algorithm that processes a user's serial number in a program by reverse engineering the program from disassembled instructions and data. During the task, the participants were instructed to think aloud, and their verbalizations were collected and analyzed. Video data of task performance was also collected from each participant's computer monitor and the researcher took notes about each participant's task performance. The data from the observational study was analyzed and interpreted to develop a theory about how reverse engineers make sense of programs from assembly language representations.

1.5 Research Contributions

There are four research contributions in this dissertation:

1. The description of the representational gap in software reverse engineering and conceptualization of reverse engineering as a "sensemaking" task,
2. The decomposition of situational factors in the process of understanding executable programs,
3. A systematic elicitation of the structure and content of concepts and procedures in reverse engineering from subject matter experts, and
4. A theory that describes the process of sensemaking in reverse engineering.

All of these contributions to the body of knowledge provide elements of a foundation which begins to bridge a representational gap that exists between low-level executable representations of programs and high-level abstract representations that reverse engineers use in practice [30, 33]. Higher-level reasoning in reverse engineering is essential in order to connect the low-level data representations of executable programs to higher-level abstractions representing actions, events, and intent in programs in order to meet the Air Force's goal of "automated vulnerability analysis" in the next two decades [188].

The first contribution is the description of the representational gap in software reverse engineering and conceptualization of reverse engineering as a "sensemaking" task (Chapter 2). This contribution relates the fields of literature in situation awareness, sensemaking, and mental models with other bodies of literature on program comprehension and reverse engineering programs from assembly language to argue why higher-abstraction approaches are necessary to solving problems in reverse engineering. This contribution also captures and represents the problems of comprehending executable programs in a way in which they can be approached by qualitative and knowledge elicitation techniques used in other problem domains. This description of the problem and conceptualization of the cognitive aspects of reverse engineering executable programs has not appeared before in the research literature.

The second contribution of this dissertation is the decomposition of situational factors involved in the process of understanding executable programs (Chapter 4). Reverse engineering executable programs involve extensive interaction between actions on a computer system and mental reasoning processes. As such, much of the problem can be considered a human computer interaction problem which is situated in an environment [206]. This contribution characterizes the different abstractions that are used in reverse engineering to represent the elements of the environment, knowledge, and the task itself, which are involved in a reverse engineering situation. This contribution was necessary in order to determine and organize the higher-level concepts, procedures, and interactions that are required to computationally realize the "situation" that intelligent reverse engineering tools will reason over in the development of more autonomous reverse engineering capabilities in the future. This decomposition is not found elsewhere in the research literature.

A third contribution of the dissertation is a systematic elicitation of the structure and content of concepts and procedures involved in reverse engineering executable programs (Chapter 5). This provides the framework for developing the “higher-level” goals that automated analysis tools would pursue, and for establishing the knowledge domains which these tools would need to have access to in order to perform effective reasoning and problem-solving tasks in the reverse engineering domain. This type of task decomposition for reverse engineering executable programs is not found in the research literature.

The fourth contribution of the dissertation is a theory about the process people use to make sense of executable programs. The theory of how people make sense of executable programs and the analysis of the verbal protocol data that provides support for the theory is presented in Chapter 6. The theory involves a cycle consisting of seven sub-processes: 1) goal representation, 2) planning, 3) carrying out a plan, 4) sensing information, 5) interpreting information, 6) updating the mental model, and 7) generating a hypothesis. This contribution to the body of knowledge provides the conceptual foundation necessary to develop algorithms to mimic the sensemaking loop that humans use to interact with the environment and reason in higher-level ways when analyzing executable programs. It is also proposed as a theory of sensemaking which, in future research, can be applied to other problem domains in order to establish the generality, boundaries, and other implications of the theory. This process of sensemaking is unique in that it describes sensemaking at a level of abstraction and interaction with the environment which, unlike the other theoretical models of sensemaking discussed in Chapter 2, produces claims which can be empirically tested and provides clues to how this process could be computationally realized.

In addition to these research contributions, the research efforts described in the dissertation produced a number of new questions which are relevant to future research. These questions are discussed in Chapter 7.

1.6 Definitions and Conventions Used in This Dissertation

A working assumption of this research effort (and a fundamental premise of human-computer interaction) is that people are information processors that solve problems in interactive environments by processing information, applying stored knowledge, and performing actions [38]. Following the convention used throughout the artificial intelligence and cognitive science literature, *knowledge* is used to describe a person's beliefs or the structure, contents, and representations of a person's memories and thoughts rather than statements that have been proven factual [130, 6]. Knowledge is referred to in terms of concepts (declarative knowledge) and procedures (procedural knowledge) stored in a person's memories [6].

A *task environment* refers to the interface provided to the problem solver, whether the problem solver is a person or a computational agent [23]. In this dissertation, the term *task environment* and *environment* are both used to describe the presentation of information and affordances to a person performing a task and the underlying structure of the system being interacted with, whether it is in a controlled experimental task or a naturalistic, or real-world, setting [113]. The term *affordance* is used to describe any object, item, display, or control in a task environment that a person can manipulate [139]. The task environments involved in this dissertation consist of computer graphical user interfaces providing the information and affordances that are presented to a person to allow the person control over the state of a disassembled program.

A background assumption of this dissertation is that when people perform tasks like reverse engineering programs, the processes can be represented as a number of activities in pursuit of goals and sub-goals [135]. Another assumption is that software reverse engineering is a *complex problem-solving* domain, which indicates that among other things, background knowledge is required to perform tasks in the problem domain [154].

1.7 Outline of the Dissertation

This chapter introduced the motivation for studying how people understand programs from executable or assembly language representations. It described the problem in terms of specific research questions and briefly enumerated the contributions of this dissertation in providing answers to these research questions.

Chapter 2 provides a literature review which details how the concepts of situation awareness and sensemaking apply to the problem of software reverse engineering and highlights knowledge gaps that exist in the literature. Chapter 3 provides an overview of the design of the data collection and analysis methodologies that were used to answer the research questions presented in this chapter. It discusses and justifies the decisions used in selecting, designing, and carrying out all three methods and describes how each of the methods provides an integral piece in answering the research questions.

Chapter 4 presents a case study of the situational aspects of reverse engineering an executable program including the task environment, the goal structure of the task, and the knowledge needed to perform effectively in the reverse engineering task. Additionally, Chapter 4 discusses how the results from the case study were used to inform the development of the subject matter expert interviews.

Chapter 5 describes processes, approaches, knowledge, tools, and techniques elicited from interviews with subject matter expert reverse engineers. The chapter also describes how the knowledge elicited from the subject matter expert interviews was analyzed, represented, and verified, and how it was used to inform the development of the observational study.

Chapter 6 describes the observational study and describes how the data was segmented, coded, transformed, and interpreted to answer the research questions.

Finally, Chapter 7 summarizes the overall findings of the dissertation into a theory of sensemaking in reverse engineering. It also discusses the implications of this theory and indicates areas for future research.

2. Literature Review

2.1 Overview

Chapter 1 introduced the problem of understanding how people make sense of executable programs and presented the research questions:

Q1: What conceptual elements are involved with making sense of an executable program during reverse engineering?

Q2: What procedural elements are involved?

This chapter reviews the literature applicable to eliciting how software reverse engineers make sense of programs from executable representations. In particular, this chapter describes the current state of knowledge about situation awareness (Section 2.2), the human processes of *sensemaking* which enable people to develop situation awareness (Section 2.3), and the concepts of *mental models* and *situation models* (Section 2.4) which are central to understanding sensemaking in reverse engineering.

After the discussion of mental and situation models, Section 2.5 describes what is known about a related problem: how programmers think about and mentally represent knowledge about programs when reading source code. Section 2.6 reviews the differences between programmers comprehending programs from source code and reverse engineers making sense of executable programs. Finally, this chapter describes the gaps in the literature which motivate the necessity for researching how people understand programs from executable representations.

2.2 Situation Awareness

Situation awareness is a term used in the human factors literature to describe a person's perception of the relevant elements in a task environment, the integration of perceptual information with the goals of the task, and the ability to project the state of these elements into the future [68]. The term situation awareness is often used more casually to refer to a person's understanding, comprehension, and interpretation of a situation [169, 170]. Situation awareness has also been construed as an awareness

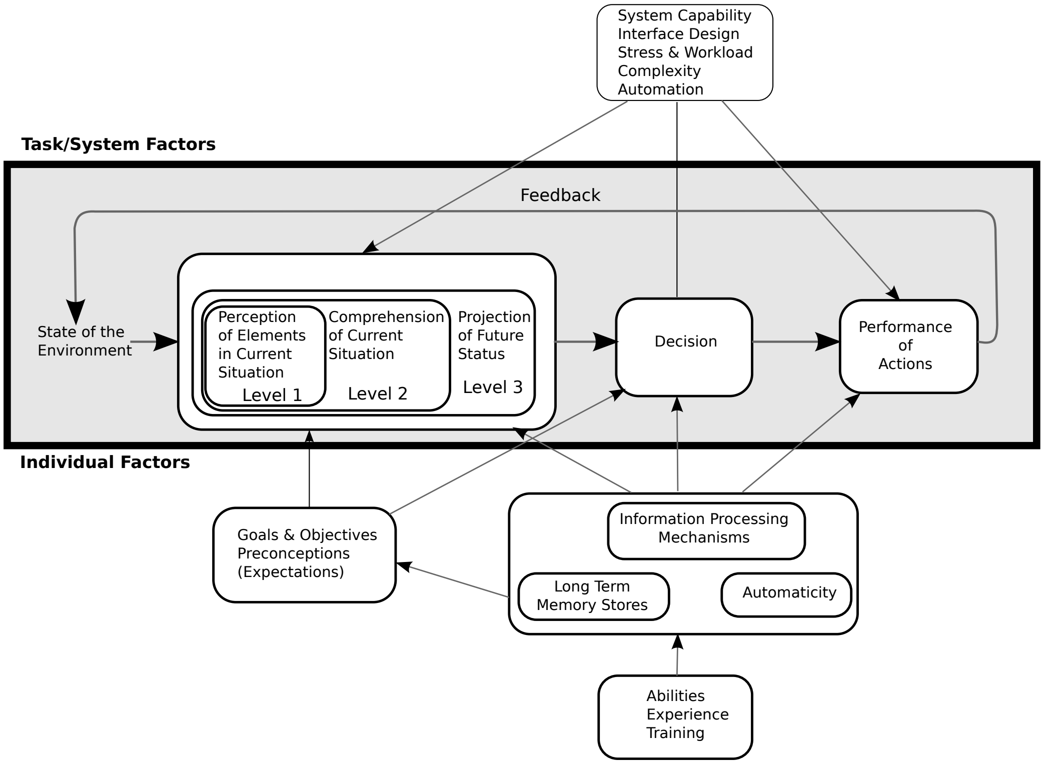


Figure 1 Endsley's Model of Situation Awareness [68].

of information that is directly available from a person's working memory or which is activated and retrievable from a person's long-term memory [169].

The most prominent model of situation awareness in the literature comes from Endsley [68]. Endsley's conceptual model of situation awareness ranks a person's awareness of a situation by three loosely-defined levels:

- Level 1: Perception of elements in the environment,
- Level 2: Comprehension of the current situation, and
- Level 3: Projection of future states.

2.2.1 Perception of Relevant Elements. Endsley's model defines Level 1 Situation Awareness (SA) as the perception of relevant elements in the environment. *Elements* refer to all of the "things an operator needs to perceive and understand" that might be involved in a situation [68]. In this dissertation, elements are taken to mean objects, information items, and affordances (or controls that people can manipulate in the environment) along with their relevant attributes.

Level 1 elements correspond to elements in the task environment and their properties which are perceptually available to the person performing the task [68]. Some notional examples of Level 1 elements in a reverse engineering task environment include:

- An indicator that shows whether a program is stopped or running,
- A display of disassembled assembly instructions,
- The presence of a dialog box,
- The appearance of a window or text strings, and
- The presence of breakpoint indicators on instruction rows.

These elements are perceptual in nature and do not require significant mental processing for a person to determine their presence [68]. Within a single task environment, the elements that are relevant to a person's mission and goals can change depending on the specific situational context, and the goals, plans, and knowledge of the person solving the task [68].

2.2.2 Comprehension of the Current Situation. Level 2 SA describes a person's ability to integrate the goals of a task or mission with perceptual elements from the environment [68]. This ability to comprehend a situation provides a person the ability to interpret elements in the environment in the context of a broader picture of why they are important. The broader contextual picture includes concepts relating to a person's goals, the overall mission of a person's organization, likely signs of trouble in a task, and so on [68, 169, 25].

Where Level 1 elements involve perceptually available pieces of the task environment, Level 2 elements require a person to perform mental processing for them to be meaningful [69]. These elements may require a person to recognize something from the task environment as an instance of a mentally stored concept [69]. They can also be mental transformations of one or more Level 1 elements into a new mental concept [68]. Level 2 elements involve components of the task situation, the goals of the person performing the task, and person's knowledge [67].

A notional example of a Level 2 element in a reverse engineering task environment is a contiguous block of hexadecimal data which represents an array in a higher-level programming language. Since they require interpretation, Level 2 elements can be thought of as items that someone without prior knowledge in the task domain would not be able to interpret. Someone with experience working with arrays and seeing them in a stack or in a memory dump can interpret the presented perceptual information (a collection of hexadecimal values) as an instantiation of a higher-level concept (an array), but by definition, someone without the knowledge to interpret the presence of the contiguous hexadecimal values would not.

2.2.3 Projection of Future States. The third and highest level of Endsley's situation awareness model is Level 3: the ability to project the state of elements in the environment into the future [68]. The projection of future states enables a person to detect when problems are about to occur in the task or to determine whether or not a plan will succeed [68].

Level 3 elements represent possible future states of Level 1 and Level 2 elements [68]. A simple example of a Level 3 element in a reverse engineering task is a person's prediction about whether or not a conditional jump instruction (`jnz`, `jz`, etc.) will be taken when the instruction pointer advances to the next instruction. To predict whether a program will change control flow at a conditional jump instruction, a person has to synthesize information about the current instruction, the state of the program's register values, and the effects of the state on future instructions in order to make an inference about the relevant future state.

Level 3 activities involve prediction, interpretation, understanding, and making sense of the underlying causal structure of the environment [68]. Performing these mental activities requires attention, working memory, and the availability of necessary information from the environment and background knowledge [68].

2.2.4 Observations From the Literature on Situation Awareness.

There are a number of concepts underlying the literature on situation awareness which should be addressed in studying how people make sense of executable programs. These concepts provide a foundation of working assumptions, used throughout the rest of the dissertation.

First is the observation that a key frame of analysis in understanding situation awareness is the *situation.* Implicit in the discussion above is the observation that the situation involves:

- A person (or agent or team),
- An environment which has a number of *states*, or configurations,
- Goals the person seeks to achieve,
- Events that take place in the environment,
- Cause and effect relationships which make the person's actions meaningful in the environment, and
- A temporal context that orders actions and events in a linear sequence.

Second, the situation presented to a person in a human-computer interaction task is presented primarily through *information* that the person can perceive and comprehend and *affordances* which allow the person to manipulate parts of the situation [38, 69, 139]. This is similar to the sensor and effector framework used to decompose and separate concerns in the design of agent-based artificial intelligence systems [165]. Everything that the person in the situation does to understand or make decisions is a property of the person, whereas everything the environment does, either in response to or independent of the person's actions, is a property of the environment [213]. The information and affordances in the situation are what connect the person and the environment [213].

Third, is that information and affordances can represent multiple levels of meaning simultaneously [68]. Each of these levels of meaning can occupy a person's attention, potentially at the expense of another level of meaning [68].

The next section discusses background literature on how a person in a task environment comes to gain an understanding of the meanings involved in a situation.

2.3 Sensemaking

The process of coming to an understanding of a situation is referred to in the literature as *sensemaking*[1] [204]. Whereas situation awareness refers to the ability to attend to Level 1, Level 2, and Level 3 elements as a situation unfolds [69], sensemaking refers to the process that enables one to come to an understanding of the meaning and relevance of the elements that make up the situation [114, 67]. The sensemaking process is described as an ongoing integration of knowledge from a mental model of a situation, available information about the context of a situation, and perceptual data from the environment [164, 67]. Sensemaking has also been theorized as the process that enables people to make intuitive decisions [112].

Sensemaking is described by Klein, et al. [115] as connecting inferences and observations, integrating knowledge and conjecture, finding explanations for ambiguous data, diagnosing ambiguous symptoms, and identifying problems. Sensemaking also refers to comprehension of the significance of ambiguous events and data in the environment [204]. All of these different functions of sensemaking describe a group of separate but related cognitive processes (Figure 2). The collection of different processes implies that sensemaking is in reality may refer to a *class* of reasoning pro-

[1] also "sense making" and "sense-making"

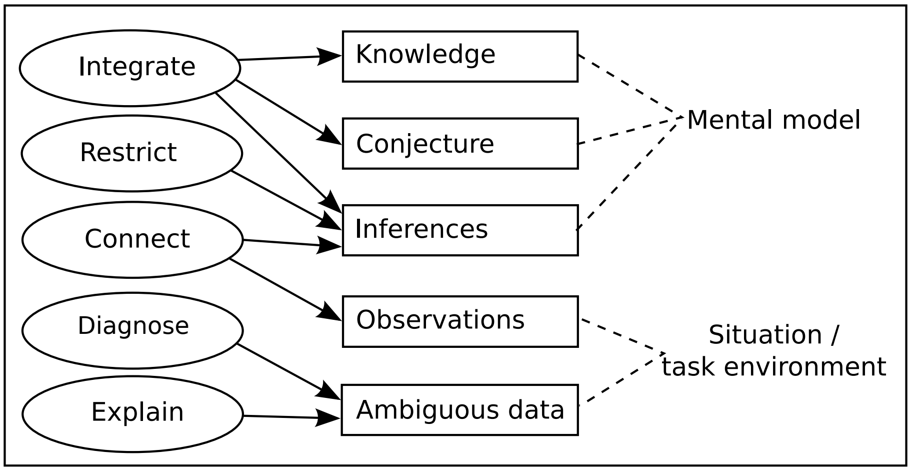


Figure 2 Various Functions Involved in Sensemaking.

cesses, rather than one single process, which encompasses a number of different types of inference and learning mechanisms [120, 103].

2.3.1 Integrating Knowledge, Conjecture, and Inferences. The first function of sensemaking is involved in integrating knowledge, conjecture, and inference [112]. *Knowledge* is sometimes used to describe elements in the set of a person's beliefs and in the set of facts from objective reality [152]. In this dissertation, however, *knowledge* is instead used to describe the contents involved in procedural and declarative memories. This means that the term "knowledge" can be used to represent beliefs and memories that are true as well as those that are not true [93, 131].

Conjecture describes propositions without factual base [152]. A conjecture is similar to a hypothesis, in that it is a proposed theory about the relationship of real, mental, or mathematical objects, but unlike a hypothesis, a conjecture has not been proved mathematically or verified by a direct test in the environment [152]. People develop conjectures and reason with them in the course of everyday life [104, 151].

Inference describes constructions of logical conclusions that are developed through the process of reasoning over propositional statements [144]. Inferences are developed from deductive, inductive, and abductive reasoning operations, but are not always logically valid [144, 104]. Abductive inferences in particular are defined based on the

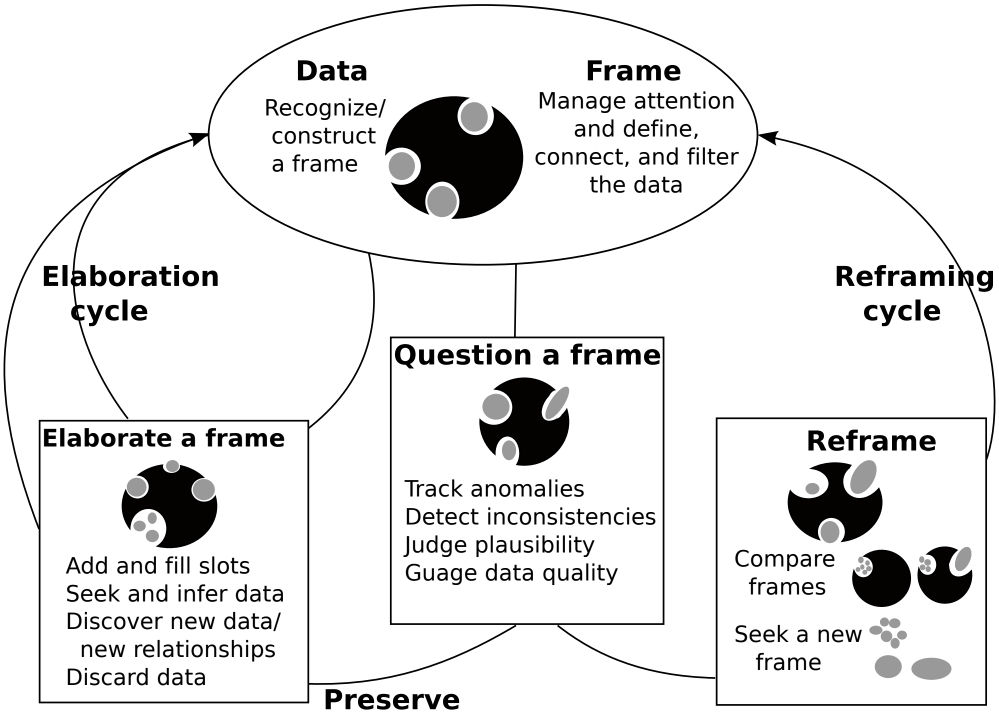


Figure 3 Sensemaking as Reconfiguring Frames (from [115]).

logical fallacy of "affirming the consequent" [104, 165]. When people integrate knowledge, conjectures, and inferences, they are involved in sorting out what they believe to be true, what they believe to be false, and what they do not know [144]. People's ability to integrate knowledge, conjecture, and inference allows them to accrue evidence toward justifying a position on a given matter, or toward refuting a position on the matter [104].

2.3.2 Restricting Inferences. In describing a conceptual model to explain sensemaking, Klein, et al. [115] theorize that problem solvers simultaneously recognize and construct knowledge representations from available data while managing currently existing knowledge representations called *frames*. In Klein's model, the two processes of creating frames and managing frames create constraints on internal reasoning processes that enable people to cope with decision making in complex real-world environments [112].

Constraining inferences to those that are relevant and plausible makes it possible for a person to adequately solve a problem without considering all available data or all possible configurations of the problem's variables [109, 124, 143]. This is important because without the ability to limit possible explanations and possible hypotheses, abductive inference has been shown to be computationally intractable in many cases [34, 210].

When people solve complex problems that present many possibilities, they rely on knowledge models to constrain their reasoning and planning processes [116]. Effective constraints on reasoning are essential to making complex problems tractable [138, 102, 101]. This suggests that the sensemaking process involved in restricting reasoning possibilities is aided by the existence of organized knowledge models that make constraints available to reduce logical possibilities [102, 212].

2.3.3 Connecting Inferences and Observations. Another function of sensemaking is that of connecting observations in the environment with inferences [115]. When people solve problems, they make predictions about what they will perceive [11]. When a person infers the future state of elements in the current situation and that inference turns out to be correct, the person's reasoning can be described as "business as usual," because there is no reasoning required to interpret the elements in the environment and no sensemaking process is necessary [112]. However, when the state of the elements in the environment do not match the state that was inferred, the person has to expend mental effort to integrate the unexpected results, which is where a sensemaking process is required [204].

When a person is presented with information from the task environment that does not match the state they inferred, it forces the person to decide between keeping the new information, keeping the inference, or maintaining the inconsistent beliefs [3]. The problem of reconciling and integrating sources of information from the environment with knowledge appears in many applications of intelligent behavior, but the atomic processes by which humans do this is not yet well understood [21, 29, 10].

When people have difficulty integrating information with their mental models, it can lead to poor decisions and reasoning errors [107]. One cause of this is that people are not always able or willing to identify gaps in their knowledge [3]. For instance, a person may have difficulty attending to information from the environment which contradicts a previously held mental model of the situation [105]. Previously-held beliefs can also prevent a person from integrating new knowledge, leading to many types of decision biases and experimentally predictable decision-making errors [175, 101, 102, 116].

2.3.4 Diagnosing from Ambiguous Observations. A fourth function of sensemaking is to diagnose problems from observations and ambiguous data in a situation [115]. This function can involve spotting problems in a plan of action or diagnosing problems that have already come into play in the situation [132]. In tasks involving troubleshooting, people attempt to build explanations for events based on inferences from information about previous events [132, 39]. Their inferences allow them to match their situation to features of general or past situations and to use causal knowledge from those situations to make predictions about likely future states in their current situation [106]. A person's inferences allow simulating the future state of the environment to determine if the transitions in states of the involved elements in the task environment will work to accomplish their plans, or if there were problems in how actions taken changed the state of the environment in a plan was already conducted [136, 106].

2.3.5 Explaining Ambiguous Data. A fifth function of sensemaking is to explain data that does not fit in the context of the understood situation [115]. Explanation involves the integration of information from the environment with a mental model in a way that can best account for the information and inform reasoning processes [147, 5, 99].

An example of sensemaking as presented by Klein, et al. [112] consists of ambiguous data and neonatal intensive care nurses whose task was to interpret the symptoms of a baby with sepsis. In the vignette, a less experienced nurse saw symptoms of sepsis in a baby but did not have the experience to connect those symptoms with the diagnosis and resulting urgent treatment that was required. A more experienced nurse saw the same symptoms but had the knowledge to connect the information from the environment (the symptoms) with her own knowledge to form the diagnosis and get the baby the needed treatment [112].

2.3.6 Sensemaking as Structure and Data Loops. Conceptual models of the sensemaking process represent it as an iterative or loop-based process. Zhang, et al. [215] present an integrated model of sensemaking as an iterative mental model modification which is supported by observations of students making sense of stories in news articles (Figure 4). Zhang's model, like the Klein model, describes a high-level process of integrating task and problem knowledge with existing knowledge structures. It describes sensemaking as:

1. Identifying gaps in data and structure,
2. Actively seeking for information and structure, and
3. Accretion, tuning, and restructuring of mental models.

The activity of identifying gaps involves learning that there are inconsistencies between knowledge and perceptual information [215]. Gaps between knowledge and perceptual information indicate that there may be a problem with a person's current set of assumptions [112]. When people identify knowledge gaps, it leads them to either modify their knowledge structures or to seek information that matches their knowledge structures [115]. When making sense of a knowledge gap, people seek information to build, support, or refute hypotheses [99].

Instead of seeking information, people can also seek structure to organize information that has already been perceived or interpreted [150]. The activity of seeking structure involves accretion, tuning, and restructuring of a mental model [162]. *Accretion* is the addition of new information to the set of a person's prior knowledge.

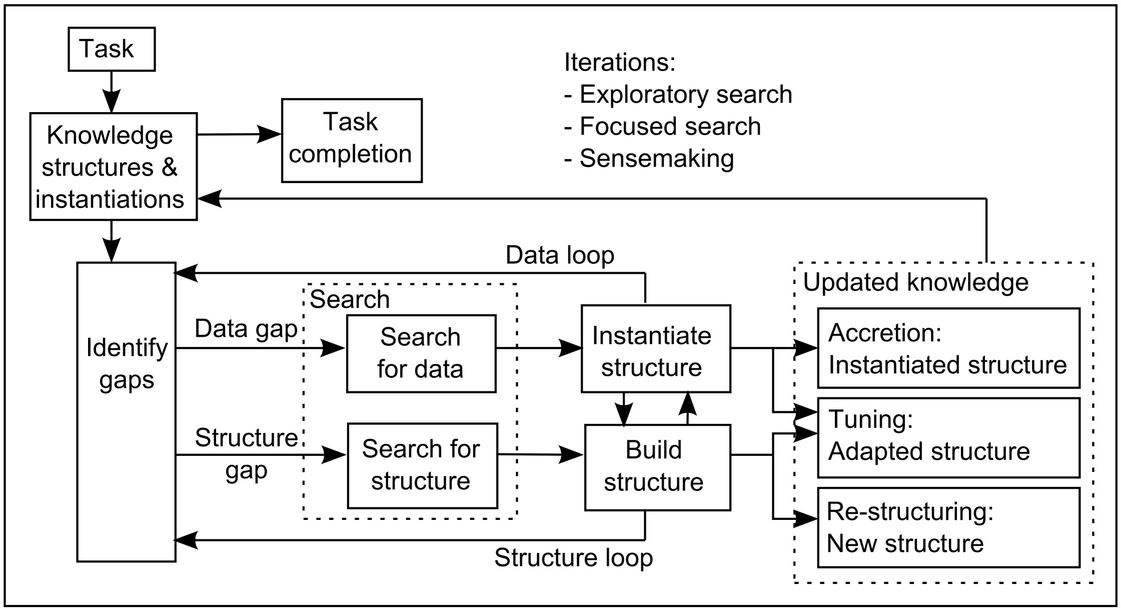


Figure 4 Sensemaking as Mental Model Manipulation From [215].

Tuning is a process of adjusting a person's background knowledge to match new knowledge. *Restructuring* involves re-evaluating and re-forming an entire knowledge schema to better accommodate the perceptual information [162, 215]. These activities take place until a reasonably explanatory mental model is completed or until the decision task is complete [215].

Pirolli and Card [150] present a model that describes the sensemaking process with a sensemaking loop and a foraging loop. The sensemaking loop (Figure 5) involves the development of a mental schema, hypothesis, and representation, while the foraging loop involves activities for searching for evidence from the environment [150].

2.3.7 Summary. The literature on the sensemaking process presents a few assumptions. First is that sensemaking can be a static or an interactive process. Static sensemaking involves simply the interpretation of information into a mental model representation. Interactive sensemaking however, involves providing inputs to the task environment in a situation in order to make changes and gain more information, a process which is often called *information foraging* [150].

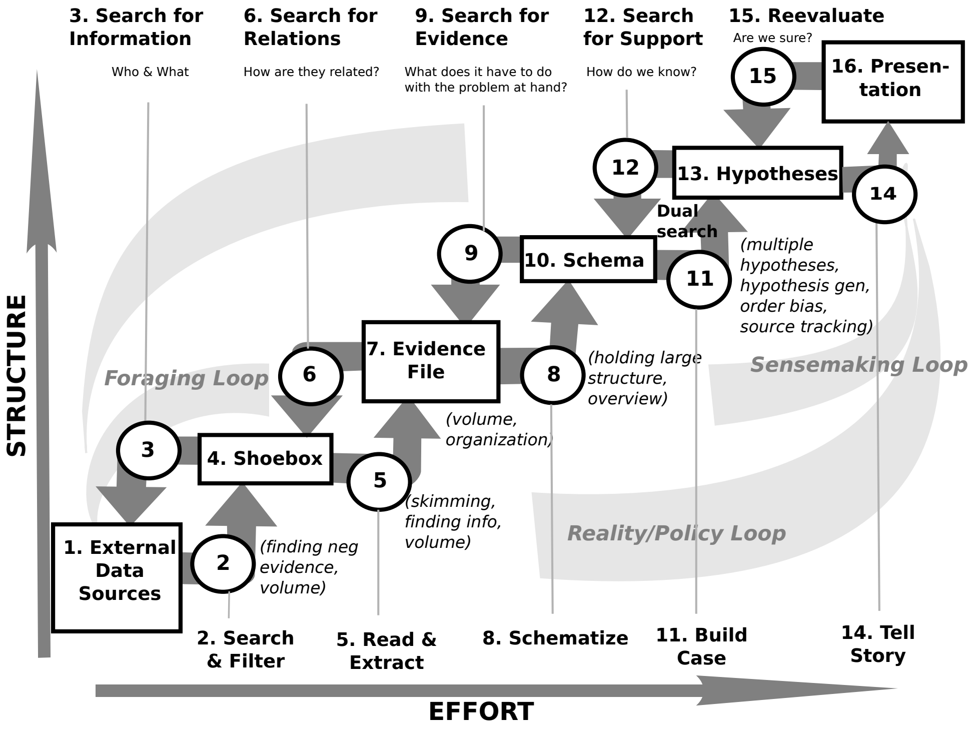


Figure 5 Sensemaking as Foraging and Data Loops [150].

Second is that sensemaking is an iterative or cyclical process. This means that outputs from one iteration through a sensemaking loop are inputs to a second iteration in the loop. Since each iteration in a sensemaking loop provides a person information about whether actions taken in the last iteration were productive or not, this means that sensemaking must involve some type of learning from experience, as well as learning from new information.

Third is that across all of the literature reviewed, sensemaking processes involve the following activities:

- Generation of hypotheses,
- Reasoning about the consequences of a hypothesis,
- Seeking information to support or refute the hypothesis, and
- Integration of new information with previous information in a mental model.

The key product of the sensemaking process is a mental model. The mental model is the theorized mental representation that is initially generated to mentally store information about the situation. The mental model can be added to and subtracted from (accretion), have values changed (tuning), and re-organized (restructuring) [161]. The next section discusses the literature involving mental models.

2.4 Mental Models

A mental model is a hypothesized representation used to depict a person's knowledge structures relating to a particular environment, situation, object, or concept [138, 80]. Results of empirical work suggest that during task performance, problem solvers query and manipulate mental models to predict the future state of a problem or situation [101]. In addition, mental models have been shown to guide participants' search processes in problem-solving tasks [116]. The presence and use of mental models by problem solvers is supported by a number of experiments that have used the theory of mental models to predict systematic errors in reasoning that are not easily explainable without the mental model construct [102, 116].

Peoples' awareness and understanding of situations are impacted by the quality of their mental models [101]. High-quality mental models tend to be highly structured and accurate. In general, experts perform better than novices because experts have highly-structured stored memories of interaction that closely meet the needs of their tasks [101, 60]. Organized knowledge allows experts to quickly assess the relevant aspects of their situations, assess *meaningful* states in the task, and perform tasks effectively [60]. However, people with little experience lack the stored patterns of concepts that would help them determine which elements in their situations are relevant [210, 214, 150].

Experts also tend to have more refined knowledge of how to perform tasks than novices, including knowledge of the actions needed to accomplish their tasks and representations of available states and actions [173]. These features of experts' procedural knowledge enable them to perform faster, better monitor progress toward problem goals, and estimate the difficulty of tasks more accurately than novices [60, 173].

2.4.1 Knowledge of Memorized Patterns. Both declarative and procedural knowledge tend to be organized around memorized patterns [162, 195]. In a study of memory, chess masters were able to reconstruct most or all of the positions on a chess board after having seen the board for only five seconds but when novices were given the same task, they could only recall two to three pieces [60]. However, if the pieces on the chess board were arranged randomly instead of in a pattern that chess players typically encounter, the expert lost the advantage in reconstructing the board [60]. These results indicate that the experts were not recalling individual pieces on the board but rather stored patterns of pieces that they had experience with and could easily reconstruct from memory [60]. People are able to use memorized patterns of concepts and procedures to mentally represent entities, parts of entities, properties of entities, actions, processes, and categories of entities [121].

One factor that makes studying mental models challenging is that mental models are not directly observable [138]. In order to study peoples' mental models, researchers attempt to elicit problem-solving behavior in ways that allows them to infer properties of internal mental model representations [102, 67]. By using knowledge elicitation methods, researchers are able to construct situation model representations of the mental models of participants in the task [67, 110]. Unfortunately, there is no single best representation formalism for capturing and representing a situation mental model [93, 110, 67], and particularly, a person's situation model of the state of an executing program.

Various computational representations exist to help researchers communicate about mental models [171, 161, 93, 4]. These representations vary depending on the type of knowledge they are intended to describe. Any model of a system loses fidelity in the process of modeling and makes particular commitments to details about the knowledge architecture [104]. Because of these commitments, each available formalism has its own benefits and drawbacks, which can make it more or less appropriate for representing certain types of knowledge or for solving certain types of problems [131]. Knowledge types represented in computational models of human memory are categorized according to the type of human memory theorized to store and retrieve them, namely procedural and declarative knowledge [4, 195].

2.4.2 Procedural Knowledge. *Procedural knowledge* consists of how-to rules that enable a person to perform tasks in a situation [4]. Representations of procedural knowledge express task processes in terms of rules involving concepts and information requirements [179, 4]. Problem solvers are able to select and apply specific sets of task operations (or methods) in order to achieve higher-level goals and the sub-goals from which they are structured [38]. Procedural patterns of knowledge can be thought of as organizations of goals, operators, methods, and selection rules within a domain [38]. Procedural knowledge can also include heuristics and reasoning short-cuts which are specific to a problem domain [143, 124, 148, 94].

2.4.3 Declarative Knowledge. *Declarative knowledge* includes *episodic knowledge*, which consists of memories indexed by particular contexts or experiences [195] and *semantic knowledge* (also called *conceptual knowledge*), which consists of facts, objects, and relations between concepts [162]. Semantic knowledge consists of the components of knowledge related to memory of facts, objects, and situations [4, 6]. Knowledge of situations seen previously is stored in episodic memory [195]. Episodic knowledge involves the knowledge that allows people to reconstruct information from their mental indexes into episodic memories of an environment [195].

2.4.4 Representations of Mental Models. A number of *knowledge representations* have been proposed to enable the development of computational models of cognitive processes. These include theories that ascribe particular representational views of cognition, such as a neurophysiological representations [141] and “symbol system"-based cognitive architectures [6, 134]. Other knowledge representations exist for specific scientific or engineering purposes [131, 93].

A *schema* is a representation that describes how a set of mentally stored concepts is structured [162]. A schema contains features or attributes that describe and categorize objects [162]. According to schema theory, people classify new entities into categories based on how those entities relate to concepts they are aware of [162, 12]. The categories of a concept representation and its attributes compose a schema representation of knowledge [162]. Conceptual schemas are representations that can store information about elements and affordances that exist in the task environment [162, 139]. Schema theory claims that schemas are manipulated during the process of problem solving in order to change the content and structure of a mental model [138].

Another way to represent concept knowledge is through a frame-based concept representation [130]. A *frame* is a data structure that is designed to store a "stereotyped situation" that represents knowledge people hold in their memories. A frame contains one or more *terminals*, which are slots for data that may be filled with values representing the characteristics of a particular situation. Frames are organized into *frame systems* that are "networks of nodes and relations" that represents a person's background knowledge with respect to a given problem domain [130]. Minsky proposes that people have unique frame systems for objects that are important to them, but that these frame systems are composed from a set of basic or generic frame structure components [130].

In frame theory, terminals in a given frame have a default assignment that change as the needs of a situation dictate. Frames can be modified in the course of a task, and transformations between frames account for changes in the visual space, actions in the environment, cause and effect relationships, and changes in the current "conceptual viewpoint" [130]. In assessing a situation with a frame system, after a person perceives a situation, they develop a frame to match the situation [130]. A process of pattern matching then assigns values to the terminals of the frame so it can be instantiated [130].

A *script* is a representation formalism used to communicate episodic knowledge of a temporally-indexed sequence of concepts [171]. Mental models based on scripts enable people to specify sequences of inputs or actions that are needed to accomplish each function in their current task strategy [38].

2.4.5 Overview of the Literature on Mental Models. The discussion on mental models in this section makes a few key points about mental models. First, they are not directly observable, so methods at eliciting a person's mental model can only provide information about some of the properties of the mental model [138, 67]. Second, mental models represent both the primary output of the sensemaking process as well as one of the primary inputs to the process [215]. This means that mental models are constantly changing and being updated as a person interacts with the environment [215, 115].

By analogy, these points indicate that when people try to make sense of executable programs, they do so by forming, querying, and changing a mental model, interacting with a task environment, and interpreting new information. The research literature from the field of program comprehension also investigates how people make sense of programs and develop and maintain mental models of the programs, but from source code representations and with the intent to maintain the program, rather than to find vulnerabilities in it. In the next section, the body of literature in program comprehension is briefly described in order to better outline what is known and what is not known in understanding how people make sense of executable programs.

2.5 Cognitive Processes in Understanding Programs

Throughout the literature, the terms "software reverse engineering" and "reverse engineering" are used to refer to various activities involved in reconstructing a mental model or other meaningful representation of a program, typically from the program's source code [185]. Tilley [192] describes the activities of reverse engineering as: sifting, reading, processing, and mentally connecting relevant information and events from the program's source code and other artifacts through "mental pattern recognition" to create "more abstract system representations" [192]. At the surface, this description of reverse engineering is very similar to how Klein et al. [115] describe the process of making sense of a situation.

One of the keys in how people comprehend programs involves the abstraction of low-level data into high-level concepts [77]. Rather than taking an approach that reverse engineering is purely a process of abstraction [35], reverse engineering can better be understood as connecting information from the environment with mental models of concepts [19], programming plans [181], and control flow representations [178].

2.5.1 Mental Representations of Programs. In the reverse engineering literature, *concepts* are used to describe the knowledge structures that programmers or reverse engineers have of how programs work [61]. Concepts are sometimes depicted as mental constructs that are roughly equivalent to *classes* in an object-oriented framework, and locations in source code files represent instantiated extensions of those concepts [196, 157]. Rajlich [157] describes the importance of intensions, or meanings to the process of program comprehension. For several years, researchers in program comprehension have sought to assign concepts to areas of a program's source code [157]. In Rajlich's model (Figure 6), a "concept triangle" represents the process of program comprehension as a combination of:

- Naming,
- Annotation,
- Traceability,
- Recognition,
- Location of labels, and

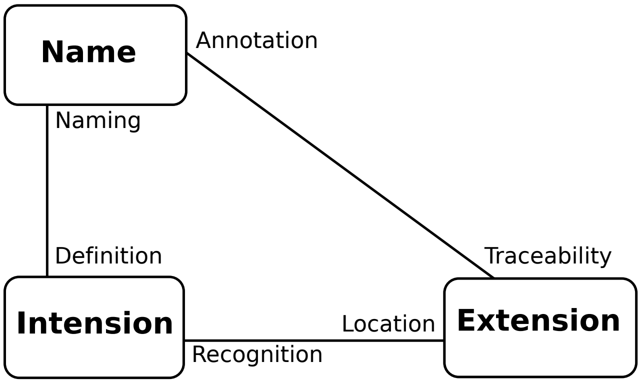


Figure 6 Rajlich's Concept Triangle Model of Program Comprehension [157].

- Extensions (plans in the program).

Although "automated concept assignment" is alluded to in [19], the literature does not provide any indication that tractable concept assignment algorithms exist, particularly for programs represented only in assembly language.

A broader view of the term "concept" in the literature uses the term to refer to user- or programmer-defined "concerns" that provide a straightforward mapping to something a person cares about in the development of a program [64]. In this view, a concept such as "saving" can correspond to the programmer's desire to have functionality that allows a user to save a file [64].

Another approach to understanding peoples' mental models of programs is to map "plans," or actions taken by the program's designer toward some intended goal or to locations in the program's source code [100, 181]. The term "plan" was borrowed from the field of artificial intelligence [155], in which a plan is a set of intended actions an agent selects to transition between states of a problem space in order to reach a goal [165]. Plan recognition is the process of inferring that a plan has been conducted by mapping the actions observed from an agent's behavior to a conceptualization of the states of a more abstract *plan* in a knowledge base [79].

Plans represent "reusable patterns of data flow and control flow" [119]. Plans are different from algorithms and procedures in that they are more abstract patterns that can be "composed in complex ways," and represent the ways that programmers think about programs [119, 181]. In the program comprehension research field, there have been attempts at assigning abstract mental plans to instantiated plans in program source code [100, 119, 2]. There is a great deal of ambiguity between what constitutes a plan to accomplish something in a program and the implementations of those actions in source code [119]. Additionally, without the structure provided by higher-level languages, there are an extremely large number of possible implementations for any given plan, many of which are not located in a single location in source code [119]. There are also many given plans that could map to a given implementation in source code [119]. Finally, constructs in programming languages such as the `goto` statement make even complex approaches to plan recognition intractable [119].

The processes programmers use to integrate observations from source code and background knowledge have been described in general as "top-down" or "bottom-up" approaches [203, 202, 200]. Bottom-up information seeking processes involve "exploration" activities in information foraging [150] and top-down processes involve "directed search" [40] for information. Tilley [192] calls this process *information exploration* and notes its primary importance in reverse engineering. Information search involves focusing on a goal to retrieve the information, figuring out how to retrieve the information, and following the plan of action to retrieve the information [207].

2.5.2 Processes in Understanding Programs. Reverse engineering is a goal-directed activity that takes place in a temporal context [193]. Because of this, elements of complex problem solving, namely constructing a goal representation, forming a plan, and carrying out a plan [135, 154] are needed to account for the goal-directed nature of problem solving in the task environment. From the descriptions of reverse engineering and the structure of knowledge presented above, the steps in the process of understand a program can be described in the more general terminology of making sense of a situation:

- Finding relevant information (goal-setting, planning, and acting to carry out information foraging activities),
- Sensing information (such as reading source code),
- Interpreting information,
- Updating a mental model (connecting information with existing concepts), and
- Generating hypotheses and assumptions.

These aspects of program understanding are described throughout the remainder of this section.

2.5.3 Creating a Goal Representation. In simple planning problems like the blocks world domain, or Towers of Hanoi, it is taken for granted that a goal is present and a goal representation is straightforward [73]. In small-scale tasks, goals can be specified in terms of a fixed set of predicates [73, 165].

In so-called "large world" problem domains [20], determining the goal and a suitable representation of the goal state is a challenging problem with aspects that change during task performance [154]. In more complex problems, the goal may not be known from the outset or may only known in a very vague way [154, 17]. Particularly in problems where information is to be retrieved from an information processing system, people may not be able to adequately describe what information they want, so the development of a representation is an essential part of the process [17].

In real-world tasks, the goal evolves as more is learned about the problem [40, 150]. Instead, as the person gains information about the goal, the person's mental representation of the goal is added to and taken away from until it is a close representation of the desired final state [215]. Depending on the person's mental model at the beginning of the problem-solving process, the initial goal representation may look very different from the goal representation at the end of the problem-solving process.

2.5.4 Planning. Once a person develops and understands the goal, a plan connects the current state of the environment to the goal state through a sequence of actions [165]. Constructing the plan involves generating a set of actions and inferring resulting states to determine how well those actions might be able to achieve the goal state [165].

In generating actions, people might consider how each action will change the state, so that they can construct a mental path of state modifications that eventually will end up at a state that meets the goal condition [199]. Understanding programs involves mental reasoning processes as informed by seeking information in the reverse engineering task environment, so plans should be able to refer to both types of action sequences.

2.5.5 Carrying Out the Plan. Carrying out a plan is the process of following the sequence of planned actions [199]. As the plan is carried out, a person may need to construct sub-goals and sub-plans to proceed along a sequence of actions, or back up in the problem-solving process, often called "pushing" or "popping" goals [199]. Whereas planning involves selecting or reviewing the actions to be taken, carrying out a plan includes actually taking the actions themselves.

2.5.6 Sensing Information from the Environment. When people carry out tasks, they are constantly perceiving and reacting to information from the environment, monitoring the state of the environment for relevant changes, and keeping track of progress as the plan proceeds [75]. Sensing information is the act of perceiving information from the environment and involves perceptual activities like directing intention toward an item on a display and encoding the perceptual information [69, 6]. In the ACT-R theory of cognition as information processing [4], the person perceiving a task takes in information cues from the task environment through shifting attention and encoding the item.

2.5.7 Interpreting Information. Interpreting information involves cognitive processing taken after perceiving and encoding an item from the environment [6]. Interpreting information is different from sensing information because the process involves recognizing the information by matching the features from the environment to a chunk in declarative knowledge where sensing information only involves the encoding of the features [6].

Interpreting information from a program can be thought of as connecting conceptual meanings to locations in a program [157, 192]. Concepts in programs can refer to plans [181], abstract classes of objects [196], or concepts from the situation model, the program model, and the domain model of a program [200]. *Plan recognition* in reverse engineering refers to matching items in a program's code to abstract patterns of intentions, or "plans," that represent the functionality the program's designer likely intended [155].

While attempts have been made to assign abstract mental plans to instantiated plans in program source code, these have only been applied in very small programs with limited results [100, 119, 2]. These approaches have had difficulty gaining traction because there is a great deal of ambiguity between what constitutes a plan to accomplish something in a program and the implementations of those actions in source code [119]. Additionally, without the structure provided by higher-level languages, there are a large number of possible implementations for any given plan, many of which are "delocalized" throughout multiple locations in a program's source code [119]. Many given plans can also map to a single implementation in source code [119]. Finally, constructs in programming languages such as the `GOTO` statement make standard search and pattern-matching approaches to plan recognition completely intractable [119, 155].

2.5.8 Updating the Mental Model. In the process of updating knowledge, new knowledge is added, existing knowledge is modified, and relationships between pieces of knowledge are changed [215]. Updating the mental model consists of the processes of accretion, restructuring, and tuning knowledge in a mental model [161]. Through this process, new knowledge is added, existing knowledge is modified, and relationships between pieces of knowledge are changed, depending on the type of knowledge received previously. Updating the mental model involves an additional level of processing beyond comprehension of information. Updating a mental model involves integrating the person's current mental model with a series of integrated models that are formed when new information is processed to develop a complete model [216]. The step of updating the mental model serves as a way to simplify all of the information encountered up to that point so they can be retrieved and reasoned with as the problem-solving task proceeds [216, 101].

2.5.9 Generating Hypotheses. Reverse engineers are known to develop questions and hypotheses about the program they are studying [28]. Tilley [192] describes the "iterative refinement of hypotheses" as one of the most important areas of reverse engineering. Hypotheses are testable propositions which can be verified through confirming or disconfirming evidence. Hypotheses that relate cause x and effect y are referred to as *causal hypotheses* [24]. Testable hypotheses in reverse engineering enable a person to acquire new knowledge which can be systematically verified through seeking out evidence from a task environment [152]. Hypotheses connect concepts

2.6 Reverse Engineering Executable Programs

Reverse engineering executable programs is cognitively challenging because it requires understanding diassembled machine code, it requires the use of specialized reverse engineering tools, and there is a great deal of knowledge to master in order to be proficient.

First, reverse engineers primarily work with disassembled executable code instead of source code [65, 92]. Disassembled executable programs can contain hundreds to hundreds of thousands of assembly language instructions [63]. Assembly language representations can faithfully model what programs do and how they execute on a computer's processor [9, 190]. However, assembly instructions provide very low-level abstractions of the program's functionality, which are at a different level of abstraction from the concepts many people use to describe how computer programs work [86].

Second, software reverse engineering tasks require skill in using a number of software reverse engineering tools [65]. *Reverse engineering tools* are programs reverse engineers use that enable gathering data and controlling the execution of programs [37]. These tools include program disassemblers, hexadecimal editors, decompilers, deobfuscators, unpackers, and more [37]. Many software reverse engineering tools are created by reverse engineers in an ad hoc manner, so they have come to be known for being difficult to use and for not having human cognitive factors incorporated into their designs [85].

Third, a great deal of knowledge is required to perform as a reverse engineer [92, 65]. Among many other areas, reverse engineers need to have knowledge of assembly language [65], operating system calls and how they work [65, 92], memory and process layout [22], and potentially attack and defense techniques [180, 186, 91, 92, 22].

Several aspects of reverse engineering executable programs are different than reverse engineering programs from source code. Reverse engineers typically analyze programs written in higher-level programming languages and that have been translated into machine code by a compiler and disassembled into assembly through a separate disassembler. When a program is compiled, objects in the higher-level programming language are translated into machine code and many of the traces of semantic information in the source code are removed by the compiler [1]. In understanding a program from assembly language, concepts are not readily provided by an object-oriented structure because assembly language does not have the same types of programming abstractions [182].

Reading source code from higher-level programming languages provides a great deal of semantic information, intentional information and structure to a programmer

[192]. On the other hand, analyzing programs from assembly language provides a very low level of data abstraction that allows very few assumptions [174].

Assembly language representations lack indications of higher-level meanings that are present in a source code representation of a program. When reading source code, a programmer has access to comments that other programmers have written about the behavior of functions or the purpose of variables. Additionally, source code representations of programs involve variable, function, class and object names which may preserve extensive information about the intended meaning of an item. These items can serve as "hints" for someone reading source code. If knowledge of the meaning of a function is desired, a reverse engineer must reconstruct that meaning from observation and analysis of the program [65].

Some of the characteristics of executable code that make it challenging to read are:

- Executable code is very complex,
- Even small programs have hundreds to hundreds of thousands of instructions, and
- Behavior of a program depends not only on the sequences of the instructions, but also input data provided from the outside world, such as files read in by the program, current values in memory, or network data captured by the program [182].

Executable programs also lack the higher-level semantics that source code contains. Executable programs sometimes have functions, but when source code passes through a compiler, functions can be in-lined into assembly language [76]. There is also no concept of buffers, arrays, or types in assembly language representations [182, 97]. All of these are essential abstractions that programmers use to think about and understand programs [187].

Song, et al. [182] also describe that understanding executable programs requires taking a whole system view. Analyzing a program's executable code requires one to consider not only the instructions and data, but also the operating system, the processor architecture, and software.

In many security-relevant applications of analyzing executable programs, programs may intentionally make the instructions difficult to understand and analyze through employing code obfuscations [50].

Program comprehension is qualitatively different in several aspects from the way reverse engineers understand code in assembly language.

2.6.1 Information in Reverse Engineering Tasks. Information cues provide a person status as to the current state of the system (the configuration of the task environment) and the current state of the problem-solving process [69]. Information cues can indicate that it is time for a problem-solving or task-environment decision to be made or how a decision should be made to pursue a course of action [69].

Reverse engineering task environments like disassemblers and debuggers provide the critical information that impacts how tasks must be accomplished [65]. Information seeking in reverse engineering can be seen as browsing or foraging for information based on the relevance of the information that has been found, similar to information seeking in other unstructured problems [150]. Reverse engineering executable programs is an interactive activity, which requires gathering and interpreting information from reverse engineering tools, and using the tools to manipulate parts of the system being studied [22, 92].

2.6.2 Types of Reverse Engineering Information. Reverse engineers gather information about programs from a number of software applications commonly referred to as *tools*. These tools and their representations provide the *task environment* in which reverse engineers' activities are situated. The task environment includes the debuggers, disassemblers, hex editors, and other tools with which the reverse engineer gains information about and interacts with a program under study.

Information elements presented in reverse engineering tasks are dynamic in how they are represented, how they can be decoded, how they are executed, how they can be accessed by a person, and how they change with other elements. The information elements in reverse engineering tools change according to time, space, value and context. Additionally, elements in reverse engineering environments are coupled to other elements such that making changes in one part of the environment affects other parts without providing feedback to the user.

There are a number of types of information presented by a typical reverse engineering task environment. For two popular environments, the OllyDbg debugger [140], and Hex Rays' IDA Interactive Disassembler [88], some of the main representations are:

- Assembly instructions,
- Data bytes,
- Observable features,
- Control flow information, and
- Functional information.

2.6.3 Assembly Instructions. Reverse engineering a program from its executable form is often required when the provenance of the program is unknown and/or the source code of the program is unavailable. This is the case in problems such as malware analysis, vulnerability discovery, or system assessment. When source code is not available, reverse engineers work with assembly language representations of programs and with data residing in the computer's memory directly. Programs contain many thousands of assembly language instructions. Assembly instructions perform operations on memory and the processor and have very close mappings to machine code understood by the processor [187], [87]. The x86 family of instruction set architectures[2] is large and complex and provides hundreds of individual instructions. Most instructions are data movement, arithmetic, or control-flow operations [97].

Assembly code lacks most programming abstractions which are in higher-level languages and which many programmers are used to [182]. Many professional reverse engineers consider assembly language representations the major characterizing feature of *reverse engineering* activities [65], [91], [92].

Assembly languages themselves are the source of some complexity. On processors based on the x86 architecture, the byte sequences of assembly language instructions represent operations that a disassembler can translate into register names, memory operands, or instruction opcodes such as `push`, `mov`, `jmp` and `call`. In x86, different assembly instructions may produce the same effects on a system. For example, the `nop` instruction performs an exchange between the value of a register and itself, producing no side effects on the state of the program besides incrementing the instruction pointer [97]. Any other set of instruction with no side effects can be considered equivalent to the *nop* instruction.

Instructions also have variable length bytes that can make disassembly challenging. The operation the processor executes when it interprets an instruction can vary depending on byte sequences embedded in the instructions like the instruction prefix or the `MOD R/M` byte [97]. The x86 architecture allows flexible addressing where the instruction pointer can read an instruction starting at any addressable byte, even one that falls in the middle of another instruction [97]. This means a multi-byte sequence can represent several instructions, depending on the first byte read.

2.6.4 Program Data. Program data is also the source of some complexity. Programs contain data mapped in various sections of the program's virtual memory space. A string of bytes in a program's data section simultaneously represent several

[2] Because of the wide use of x86-based processors in desktop computers, the 32-bit x86 instruction set architecture is used to describe assembly language representations. Many of the same concepts apply to other instruction set architectures, such as PowerPC, MIPS, SPARC, and ARM.

different structural interpretations. Like instructions, the interpretation of data bytes depends on which byte is read first and what data type the bytes represent [16]. The same sequence of bytes can represent a program-specific data structure, integers, or floating point values, which the program reads and passes to a function as arguments, pointers to other memory locations, a string of ASCII characters, or executable instructions. Reverse engineers must be able to determine how data is used in addition to determining what data is used by instructions in a program. Other data that represents a program's state include:

- Values in the processor's registers,
- Values on the program's stack memory,
- The processor's flag values, and
- File or registry contents the program may access and change.

2.6.5 Observable Features. Assembly-level representations of programs enable collecting low-level observable data about how software interfaces with the operating system and hardware. Observations made through instrumentation of programs or operating systems can provide additional sources of information. Running programs have directly observable changes like windows that open, files that are created, and information or affordances that change in the the programs' displays. Reverse engineers can use these observations to make inferences about program behaviors and to find the locations of code that cause those behaviors.

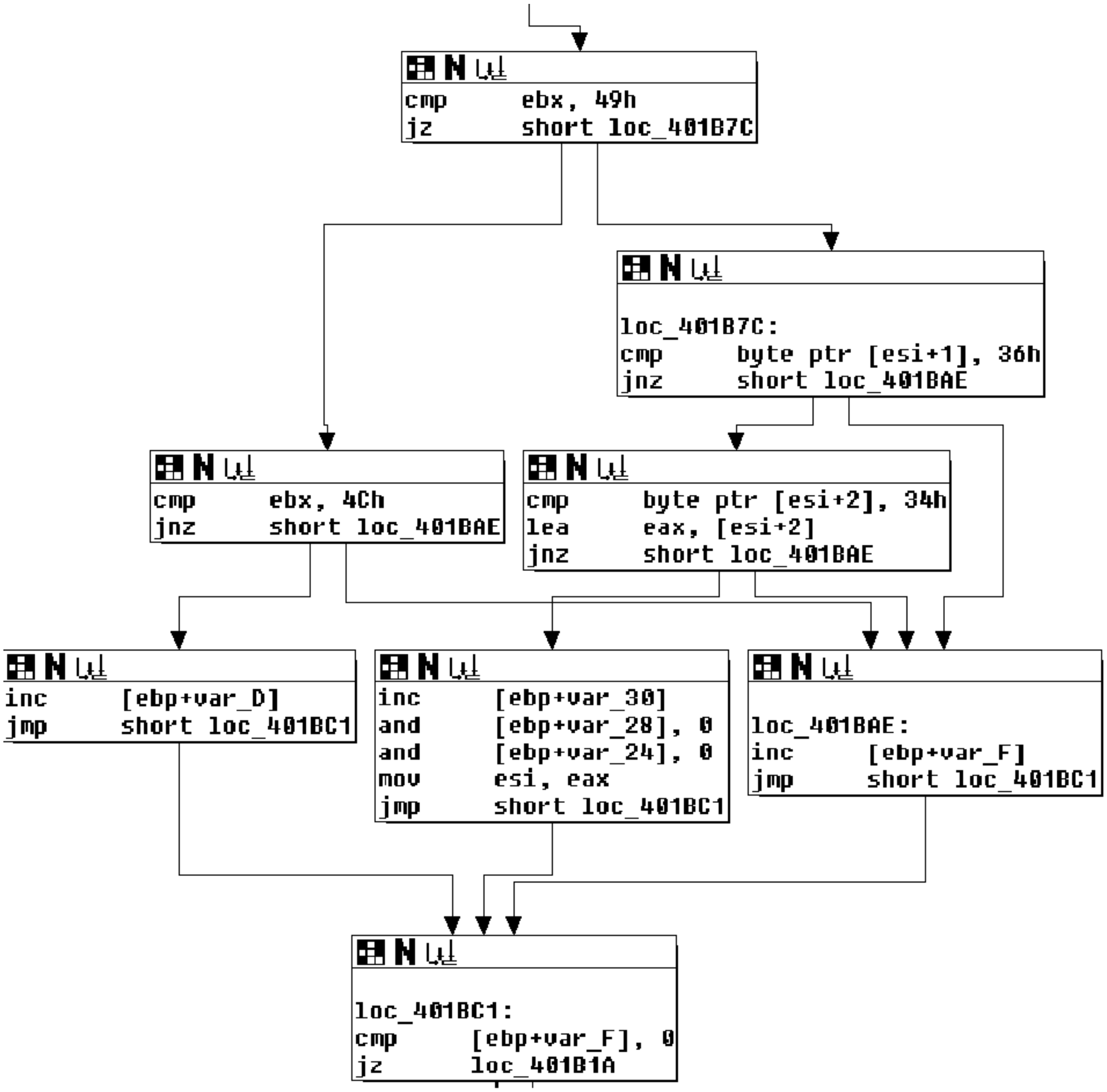
Some program or system changes are not directly visible, but can be detected by using tools to seek information about the changes. Tools that list running processes, list windows or files that are open, and display the processor's workload provide information about non-visible program changes. Monitoring tools that come with operating systems can provide some of this information. If built-in tools do not provide the right information or are not trusted, reverse engineers can use other third-party tools or write their own tools to gain more information about programs. Writing tools for this purpose is referred to as *instrumentation*.

2.6.6 Control Flow Information. Reverse engineering tools present control flow information which visually represents logical relationships within programs. Control-flow information allows people to infer causality between different instructions in a program and between different blocks of instructions. Control-flow relationships are often presented in graphs or control-flow diagrams that provide visual cues to help decompose programs in meaningful ways. IDA provides the ability to step through a control flow graph view in a debugger in a representation similar to the one shown in Figure 7.

In Figure 7, each block has a small number of instructions and goes to one or more blocks. In the top basic block of the figure, a comparison is made between `EBX` and the immediate value `49h`, which subtracts `49h` from `EBX` and sets the zero flag if the answer is zero. If the zero flag is set, then the next instruction jumps to the path to the right. If the zero flag is not set (the values are not equal), the next instruction jumps to the path on the left[3]. As a person debugs a program in the graph in IDA, upon reaching the `jnz`, `jz`, or `jmp` instructions, one of the departing arrows blinks to indicate the path which will be taken.

2.6.7 Functional Information. Functional information breaks programs into subroutines, instruction sequences, and basic blocks demarcated by control flow instructions like `jmp` or `call`. Functional decomposition allows reverse engineers to abstract away details within functions and subroutines and view them as black boxes where only the inputs and outputs need to be considered. Functional decompositions also allow reverse engineers to assign meaningful labels to areas of programs to ascribe meaning to those areas.

[3] Red and green arrows to indicate true and false paths are not shown in the black and white diagram.



```
cmp     ebx, 49h
jz      short loc_401B7C
```

```
loc_401B7C:
cmp     byte ptr [esi+1], 36h
jnz     short loc_401BAE
```

```
cmp     ebx, 4Ch
jnz     short loc_401BAE
```

```
cmp     byte ptr [esi+2], 34h
lea     eax, [esi+2]
jnz     short loc_401BAE
```

```
inc     [ebp+var_D]
jmp     short loc_401BC1
```

```
inc     [ebp+var_30]
and     [ebp+var_28], 0
and     [ebp+var_24], 0
mov     esi, eax
jmp     short loc_401BC1
```

```
loc_401BAE:
inc     [ebp+var_F]
jmp     short loc_401BC1
```

```
loc_401BC1:
cmp     [ebp+var_F], 0
jz      loc_401B1A
```

Figure 7 Graph-Based Debugging in IDA [88].

2.7 Summary

This chapter provided an overview of the research relevant to making sense of programs from assembly language. First, it described Endsley's three-level model of situation awareness [68] in the context of reverse engineering. Next it described the various processes of comprehension from the literature which represent tasks that can be considered "sensemaking" tasks. After that, it explored the literature about understanding and representing mental models of problem solvers. Finally, it discussed the literature in program comprehension and the application of that body of knowledge to the problem of reverse engineering from assembly code representations of programs.

Situation awareness describes a "current" mental state involving the perception of relevant elements in the environment, comprehension of the current situation, and projection of future states (Section 2.2). Sensemaking refers to various processes involving integrating background knowledge, conjecture, and information from the environment and making inferences that can be described as the process to come to a comprehension of the current situation (Section 2.2). Sensemaking is typically envisioned as a cyclical process in which a person modifies and updates a mental model to to provide updated information about items in the task environment (Section 2.3).

The mental models that people manipulate during sensemaking can be characterized as procedural knowledge and conceptual knowledge, or alternately "how" and "what" knowledge (Section 2.3). Various representational formalisms such as schemas, frames, scripts, and plans have been designed to depict different types of mental models in computer-readable form, but each representation has its own benefits and drawbacks for representing different types of problem-solving behavior (Section 2.3).

When people attempt to understand programs from source code, they gather information from the representations presented by source code, artifacts, documentation, and use it to update their mental model of what the program does and how it works (Section 2.5). There are a number of processes believed to be involved in program comprehension, such as abstraction, plan recognition, concept assignment, naming, annotation, tracing, recognition, and locating labels (Section 2.5). Researchers have attempted to automate these processes directly, but have typically run into problems with tractability (Section 2.5).

Finally, reverse engineers work from lower levels of abstraction than those provided by source code. In reverse engineering programs from executable representations, software reverse engineers must work from observations rather than from reading descriptions (Section 2.6). This distinction may make the activities and cognitive processes different than those typically considered in the program comprehension literature (Section 2.6).

The research literature does not address how reverse engineers make sense of programs from executable representations. In order to determine how people make sense of programs from assembly representations, a set of studies were carried out to better understand reverse engineering tasks, to learn what experts considered to be the process, and then observing people as they performed reverse engineering tasks. These studies include a case study of a reverse engineering situation, analysis of semi-structured interviews with subject matter expert reverse engineers, and an observational study. The overall methodology approach is described in Chapter 3. After that, the case study is presented in Chapter 4, the subject matter expert interview study is presented in Chapter 5, and the observational study is presented in Chapter 6. Following these chapters, Chapter 7 discusses the conclusions and areas for future research.

3. Methodology

3.1 Introduction

An essential step to meeting the Air Force's future goal of automating vulnerability assessments in complex systems is understanding how reverse engineers make sense of executable programs. Chapter 1 introduced the research problem and motivation and Chapter 2 reviewed and synthesized the relevant literature.

This chapter outlines an integrated sequence of studies to determine how conceptual and procedural knowledge are used in reverse engineering tasks. First, the overall methodology is described to justify why this methodology is the most appropriate for answering the research questions (Section 3.2). Section 3.3 describes a case study methodology to understand the conceptual and procedural elements involved in a reverse engineering situation. Following that, Section 3.4 describes the design of a semi-structured interview methodology to determine and organize the high-level procedural and conceptual elements that experts believe are involved in reverse engineering tasks. Section 3.5 outlines the design of an observational study which collected data from participants performing reverse engineering tasks to elicit and represent low-level processes of sensemaking in a conceptual theory.

3.2 Design of the Overall Methodology

To adequately study how reverse engineers make sense of executable programs, multiple research methods are used. Using multiple methods produces converging evidence which provides stronger support for theoretical descriptions of phenomena, particularly when those methods are qualitative in nature [46]. In this dissertation, three different research methods are used to understand how people build and reason about their mental models of programs when reverse engineering programs from executable representations. The three methods are a case study to investigate a reverse engineering situation, a semi-structured interview study with subject matter experts, and an observational study where reverse engineers were observed performing the task.

Knowledge elicitation (KE) methods are systematic approaches to understanding aspects of human information processing, decision making, and problem-solving [90]. KE methods have been used to study cognitive work of pilots [68], air traffic controllers [177, 69], unmanned aerial vehicle recovery teams [62], nuclear power plant workers [160], electronic warfare technicians [127], nurses [55], and fire fighters [112].

There are a number of KE methods, each with their own aims, outputs, benefits, and limitations [52]. In some cases, KE methods are used to uncover the processes underlying human intelligence [135]. In others, KE methods are focused on capturing the requirements for knowledgeable performance in a task in order to augment human performance in error-prone or difficult activities [69]. Some KE methods are used for generating domain-specific training requirements [45, 129] or to train people to make better decisions under uncertain conditions [54, 128, 127]. Others provide decision inputs to system design [68, 129], aid researchers in developing models of task interaction for user interface design [38], or develop computational cognitive models of performance in a task [38, 98, 194].

A number of overarching methodologies under the umbrella of "cognitive task analysis" prescribe standard sets of KE methods which have been found useful for training, system design, and process improvement purposes [55]. The outputs of cognitive task analysis include tables and descriptions of knowledge demands [127], collections of subtle decision cues that have been important in critical incidents [54], expert and novice solutions to problems [129], and ideas to aid in curriculum development or training needs [45]. While potentially useful, the purpose of the dissertation is not to develop training or system design recommendations. Instead, the purpose of this dissertation is to understand the conceptual and procedural elements of knowledge involved in reverse engineering executable programs and how those elements are organized.

Very little foundational research has been done concerning how people reverse engineer executable programs. However, from the literature review in Chapter 2, it is clear that software reverse engineering is a complex problem-solving task that involves integration of background knowledge with situational information from reverse engineering tools to learn about the properties of an executable program of interest. Because of these features of software reverse engineering, the research problem in this dissertation required methods that allow learning about situations involved in reverse engineering tasks, knowledge involved, and the actions that are performed. For that reason, the methods in this dissertation research include:

1. A case study of a software reverse engineering task to examine the information, affordances, goals, knowledge, and hypotheses involved in a situation surrounding a reverse engineering task,
2. Semi-structured interviews with subject matter expert reverse engineers to examine common goals, standard approaches, steps in reverse engineering tasks, and knowledge requirements, and
3. Observation of reverse engineers in a "think aloud" study to examine the behavioral processes involved in a reverse engineering task.

Research efforts to understand other complex problems involving cyber security have used case studies [31, 84] and subject matter expert interviews [27, 58] to describe the structure and content of the domain as well as the contexts in which decisions are made. Additionally, empirical studies in the program comprehension literature have relied on observational studies to learn about participants' mental models and processes [203, 202, 205, 201, 13, 53, 200, 83]. Case study methods can provide information about the situational context of problem solving, subject matter expert interviews provide information about task characteristics, and observational studies provide information about knowledge and processes involved which makes these methods well-suited for the research in this dissertation.

Since the methods used in this dissertation are qualitative, they involve interpretation from the researcher to detect patterns and uncover hypotheses [142, 66]. This is a strength of the methods in that they allow rich description of a phenomena of interest that is not possible with an experimental approach [211, 18, 81]. However, the expressive power comes at the cost of a loss of control over variables which can introduce bias in the data collection and interpretation [48] and potentially limit the applicability outside the selected case [66]. Bias must be minimized or controlled as much as possible in order to allow for robust conclusions to be made from analysis of qualitative data [211, 117, 48]. For this reason, each of the methods in the dissertation are used to establish and reinforce the findings of the others [128, 56]. In addition, the steps of the various methodologies and logic used in linking data to conclusions are presented in detail to allow readers to evaluate the merits of the analytical approach and to be able repeat the research and obtain generally similar results [211].

Qualitative analysis allows the researcher to discover patterns and themes in the data [142]. A qualitative approach was used to analyze the data from the studies in this dissertation and to integrate it together to answer the research questions. Quantitative methods were used when possible to provide additional confidence for the qualitative interpretations. In the next three sections, the design of each of the studies are described. In the following three chapters, the data and analysis from these studies are presented.

3.3 Study 1: Case Study

A case study of a reverse engineering situation was first conducted to understand and narrow down potential elements of conceptual and procedural knowledge and to identify where they originate. The study investigated the task of reverse engineering a program using the OllyDbg debugger [140] to determine the situational factors involved in reverse engineering an executable program.

A case study was needed as part of the overall methodology because the variables of interest as to how people make sense of executable programs were not clear at the outset of the research. The case study is also needed because there is no detailed examination of situational factors involved in making sense of executable programs found in the research literature (see Chapter 2). The case study presents analysis of video and audio data captured from screen recordings of a reverse engineering task performed by the researcher, analysis of verbal data taken from the task, and an investigation of information and affordances in the task environment used by the researcher in the task.

3.3.1 Motivation for Using a Case Study Method. Case study methods are used often in exploratory research because they excel at helping researchers uncover variables and potential causal relationships in a domain of interest [211]. A case study is ideal for answering "how" and "why" questions and for gaining deeper insight into the nature of a problem and into complex issues [18].

Case study methods have a number of defining characteristics which set them apart from other research methods. Hitchcock and Hughes [89] characterize case study methods as those in which the researcher is "integrally involved in the case" and which provide "rich and varied descriptions of events" that involve the chronological relation of relevant events, the combination of descriptive and analytical views of the events, focus on the individual actor and the actor's interpretation of events, and which indicate specific events in the case for attention.

Case studies attempt to portray the experience of being a part of the particular situation [78]. Benbaset, et al. [18] investigated case study methods in information systems and found that researchers use case study research to provide answers to problems "in which research and theory are at their early, formative stages." Case study research was also found to be appropriate for "sticky, practice-based problems where the experiences of the actors are important and the context of action is critical" [18].

3.3.2 Relation to the Research Questions. The problem of understanding how people make sense of executable programs involves situational and contextual factors, and theory in this domain is at an early stage of development. The case study was chosen for the research problem in this dissertation because the problem involves primarily exploratory research [48]. In exploratory research, case study methods provide tools to allow the researcher to develop and build theory at the same time [18] and to become as familiar as possible with the domain and the experience of the participants while maintaining an objective stance [48].

Solving the research problem of the dissertation involves understanding the conceptual and procedural elements involved in reverse engineering tasks. The verbal record and the model of the situation can be used to determine conceptual elements.

The case study was the most appropriate method to generate this type of information because theoretical models of mental processing in reverse engineering executable programs do not exist (see Section 2.6 for a further discussion). Since the variables of interest and causal relationships had not been established in the literature, it was not yet appropriate to attempt to apply experimental and statistical techniques [211, 18].

Standing on its own, the case study method would provide information that was useful in exploring possible ways that people can make sense of executable programs. However, the findings in the case study were also used to develop the organizational foundation for the rest of the dissertation. The case study data were used to develop the interview questions used in the semi-structured interviews (Section 3.4) and in the design of an observational study (Section 3.5). The case study data were also used to define the characteristics of the program the participants would reverse engineer and to determine what data to collect (see Section 3.5 for discussion of the design of the observational study).

3.3.3 Unit of Analysis. In a case study, a *case* can be a series of events or situational context [211]. In many case studies such as in Bryant [32], the case has an organizational context which provides the backdrop for decisions, motivations, goals, and more. In this case study, however, the context is a problem-solving *situation* rather than an organizational context.

Case studies can be characterizes as either *exploratory*, *descriptive*, *explanatory*, or *improving* types of designs [158]. Case studies can also be characterized as either *holistic* or *embedded* and comprising either single cases or multiple cases [211]. Holistic case studies are those in which the case comprises the primary unit of analysis, while embedded case studies are those in which multiple units of analysis are studied within a single case [211].

The method used in this study is an exploratory embedded case study design aimed at understanding the situational context in which reverse engineering is performed. In this embedded case study, the task environment, the task, and knowledge components are the embedded units of analysis within a single case. The case in this study is a situation in which a single participant (the researcher) reverse engineers a type of program called a *crackme* from assembly language instructions.

Data collection methods in case studies can comprise either *first degree*, or direct, methods which include examples and think-aloud protocols; *second degree*, or indirect methods, in which the researcher does not interact with participants; and *third degree* methods, in which the researcher studies available data compiled from elsewhere [118]. This case study involves first degree data collection through direct observations, a think-aloud protocol, and analysis of the crackme program and the OllyDbg debugger.

The data collection methods in this case study involved the following:

- One participant (the researcher) completed a reverse engineering task and verbalized throughout the process of solving the challenge in a manner similar to a "think-aloud" protocol.
- Video data from one of the three sub-tasks in the reverse engineering task was analyzed to infer a sequence of task actions from observable information.
- The researcher's concurrent verbalizations during the problem-solving process were recorded and transcribed into a text document.
- The task environment and crackme program were analyzed to provide additional information about the situation.

3.3.4 Analysis. Qualitative data analysis is used to analyze the results from this case study. Two types of qualitative data analysis are hypothesis generation techniques and hypothesis confirmation techniques [176]. Hypothesis generation involves the coding, "memoing," and classification of the data, followed by examination of patterns and relationships in the coded data [163]. In hypothesis generation, the researcher should not define "too many hypotheses" before the analysis has begun [176, 163]. Instead the hypotheses are determined through an iterative process of analysis, pattern recognition, and interpretation. Hypothesis confirmation techniques, such as negative case analysis, triangulation, and replication, are then used to establish hypotheses that have been generated [176, 163].

3.4 Study 2: Semi-Structured Interview Study

The second study undertaken was a qualitative analysis of semi-structured interviews with subject matter expert reverse engineers following semi-structured interviewing procedures as outlined in Wood [208]. To understand the elements of a complex task, it is often valuable to consult experts who have a great deal of experience performing the task [209]. Subject matter experts (SMEs) are people with highly refined knowledge about their specific work domains which are well-tuned to the tasks required in those domains [60, 90].

Semi-structured interviews are interviewing techniques where the questions follow a format in which all participants are asked the same questions in the same order, but the participants are permitted the opportunity to further describe and elaborate answers to the questions [55, 209]. Additionally, semi-structured interview methods give researchers the ability to ask further questions or elicit examples to clarify meanings [208, 51].

Semi-structured interviews allow researchers to elicit insights and rich qualitative data from experts about the requirements and processes involved in task performance [55]. SME interviews are also used to generate knowledge requirements of a task [55], and to learn about the tools, automation, and processes used in a task [209].

SME interviews have also been used to generate design requirements for subsequent observational studies [45], aid in the selection of prototypical tasks [208], and narrow the focus of future inquiry [209, 112].

3.4.1 Purpose of the Study. Semi-structured interviews with SMEs were used in this dissertation to determine the characteristics of reverse engineering tasks and how they contribute to how reverse engineers generate and reason over mental models of executable programs. Specifically, the characteristics sought from the interviews were:

- The work domains in reverse engineering,
- Goals and activities involved in reverse engineering tasks,
- Decision points and information cues in reverse engineering tasks,
- Specialized knowledge requirements, and
- The role of automaticity and tacit knowledge in reverse engineering work.

These characteristics were chosen because, first, they were identified as the components of knowledge-based problem-solving work from the literature (see Section 2.3), and second, they were identified as areas where knowledge gaps existed about how people develop and reason about mental models of executable programs. Knowledge about these characteristics of reverse engineering would later be used as factors to help decide on an appropriate reverse engineering task for the observational study (Section 3.5).

The semi-structured interviews asked questions which were related to 1) the concepts and processes involved in reverse engineering, and 2) the design of an observational study. The questions concerning concepts and processes involved questions about the goals used in reverse engineering executable programs, the approaches involved, how decisions are made, information and cues that are important in the task, and elements of tacit knowledge involved. The questions concerning the design of an observational study involved questions about the overall organization of problem domains in reverse engineering, the knowledge required by participants, and specialized tools and equipment used in the process.

The responses from the subject matter expert study were recorded, transcribed, segmented, and coded according to qualitative analysis methods as described in Cohen [46]. The coded data was analyzed from the concept level (sentence-based segmentation), and theme level (answer-based segmentation) to detect emergent concepts and themes involving conceptual and procedural aspects of reverse engineering executable programs as well as to provide information to assist in the design of the observational study discussed in the next section. More detail about the design and analysis of the subject matter expert study is presented in Chapter 5.

First, the design of the study is described, Next, the process by which the interview data were analyzed is presented. Afterwards, the analysis of the interview data is presented, and the themes emerged from the interviews are presented in the context of how they help understand the conceptual and procedural aspects of reverse engineering unprotected executable programs.

3.4.2 Design of the Study.

Before the interviews were conducted, the study was planned to ensure that the information would provide valuable information to answer the research questions and not waste the SMEs' time. This section describes how the interview questions were constructed, how the interview questionnaire was developed and pilot-tested, how the subject matter experts were selected, and how the interviews were administered.

3.4.2.1 Development of the Interview Questionnaire.

Before interviewing subject matter experts, the researcher conducted an extensive review of literature related to computer security analysis, software reverse engineering, program comprehension and situation awareness. A portion of this literature review which is most relevant to the research problem of the dissertation is recorded in Chapter 2.

Additionally, the researcher completed hands-on training in reverse engineering and assembly language which provided a practical familiarity with the issues and constraints involved in reverse engineering. The literature review, training, and results of the case study were all used to determine the goal of the interviews and to aid in developing the questions for the SME reverse engineers.

The questions were derived from knowledge elicitation-based interviewing techniques presented in Wood [208] and Crandall et al. [55]. The complete list of questions used in the interviews appears in Appendix B. The interview questions covered the following areas:

- Work domains in reverse engineering,
- Goals and activities involved,
- Decision points and information cues,
- Specialized knowledge requirements, and
- Automaticity and tacit knowledge.

The interview questions were aimed at eliciting two primary types of information:

1. Information about processes and knowledge used in reverse engineering and
2. Information to help design an observational study (Chapter 6)

Since the interview questions integrated these two types of information, they will be considered together.

3.4.2.2 Purpose for Each of the Question Groups. The first group of interview questions presented questions to get an overview of reverse engineering work and the reverse engineering process. In this set, the SMEs were asked to provide an overview of the different types of reverse engineering tasks they perform, to identify five authentic types of problems an expert would be solve, to break down the reverse engineering task into between 5 and 7 major steps, and to describe how different steps in the process vary. These questions were derived from interviewing techniques discussed in Crandall et al. [55]. The goal in this group of questions was to break down the overall process of reverse engineering into a hierarchy of smaller processes which can be considered "sub-tasks" or "sub-processes" within the larger scope of reverse engineering work. The answers to these questions were intended to provide an additional source of information to support the findings concerning the goals and plans involved in reverse engineering task from the case study (Section 4.4.2).

The next group of questions related to the goals and sub-goals used in reverse engineering tasks themselves. The SME participants were asked what they consider to be their main goals while reverse engineering, how they approach a plan to carry out those goals, what cues tell them when they have to change their approach, and how their approach to reverse engineering has changed from when they first started. The purpose of these questions was to provide yet another source of information with which to triangulate the results described from the analysis of goals in the case study (Section 4.4.2) and also to aid in structuring a procedural decomposition of the general executable program understanding task.

The third group of questions involved decisions and decision cues in reverse engineering tasks. The participants were asked to describe difficult decisions they have had to make in reverse engineering, when these types of decisions appear, what cues tell them they will have to make those decisions, how they determine the different options, and how they choose between the different options. This series of questions was also intended to help aid in the procedural decomposition of the reverse engineering task, but from the aspect of how reverse engineers use and seek information to support their decisions in the task. From this, the interview was intended to elicit the types of information sought and how it affects the procedural aspects of the task. The elements of information-seeking and use can be thought of as a second source of information to support the results from the case study, and to help establish the external validity of those findings (Section 4.4.4).

The fourth group of questions discussed the specialized knowledge used in reverse engineering software. SMEs were to describe a particularly difficult or complex reverse engineering task they had performed. They were to describe the aspects that made it difficult and the conceptual knowledge that helped them tackle the problem. They were also asked about skills, abilities, and conceptual knowledge that separate experts from novices. This group of questions was intended to provide additional support and guidance in organizing the conceptual knowledge involved in reverse engineering software. It also provides an additional source of evidence to support the findings in the case study (Section 4.4.6).

The fifth group of questions involved specialized tools and equipment involved in reverse engineering tasks. The SMEs were asked to describe standard and specialized tools they use and to discuss the capabilities they provide. They were also asked to describe their reverse engineering set up, their must-have tools and equipment, and the information and capabilities that these tools provide to them. This set of questions was intended to provide support for the choice of task environment used in the case study (Section 4.2) and to provide practical guidance for the construction of a reverse engineering task for participants to solve in the observational study (Chapter 6).

The final group of questions involved tacit knowledge and automaticity in task performance. The SMEs were asked what types of decisions or steps that have become automatic. They were also asked what steps in the process would likely be difficult for a novice without significant experience reverse engineering. This set of questions was intended to help determine the procedural aspects of knowledge that may be difficult to elicit in the observational study, and to identify conceptual or procedural elements that may have been missed in the think-aloud protocol used in the case study (Section 4.4).

3.4.2.3 Pilot-Testing of the Questionnaire. The interview questionnaire was pilot-tested with four software and hardware reverse engineers to ensure the questions made sense and would generate the desired information. Additionally, the questionnaire (as well as the design of the overall methodology) was vetted by the researcher's institutional review board (IRB) in the course of a human subjects exemption process. The IRB members validated the appropriateness of the questions and provided helpful guidance on the overall methodology. The IRB documentation is presented in Appendix A and the full questionnaire is found in Appendix B.

3.4.3 Selection of Subject Matter Experts. After the questionnaire was developed, vetted, and approved, five subject matter experts reverse engineers were identified from government and government-support contractor reverse engineers who attended the Reverse Engineering Workshop sponsored by the Department of Defense Anti-Tamper and Software Protection Initiative Office in January of 2010 or who had performed reverse engineering tool development research for that office. The duties of the reverse engineers in this group primarily consist of performing reverse engineering and anti-tamper related work and have established reputations with the Department of Defense. Within this group, five SMEs were asked to participate based on their reputation among their peers as experts and based on their level of experience reverse engineering software, a method used commonly in other subject matter expert studies [55, 56].

Each SME held an advanced degree in computer science or a related subject and had several (six to 12) years of hands-on experience in software reverse engineering. Additionally, each SME had also developed large-scale software programs to automate or improve the capability of their own reverse engineering task performance, so it was anticipated that they would be intimately familiar with the details of their own process and have helpful insights into the general process of reverse engineering executable programs.

The five SMEs were distinct by geography, training, education, and employment. The SMEs resided in four different areas of the United States. Each of the SMEs learned reverse engineering through a combination of self-motivated practice, college coursework, and duties involved in their employment. They received training in reverse engineering from different sources, with three being completely self-instructed and another two gaining reverse engineering experience as an extension of other low-level engineering or security work, which varied greatly. None of the SMEs attended the same college for their undergraduate or graduate degrees, and all of the SMEs worked for different organizations (as government employees and civilian contractors). To protect the privacy of the interviewees, the SMEs are identified as SME 1, SME 2, SME 3, SME 4, and SME 5 throughout the chapter.

3.4.4 Administration of Interviews. Each interview lasted approximately two hours and was conducted between February and May 2011. Two of the interviews were conducted in person and three were conducted over the telephone. In all cases, during the interview session the researcher took notes to outline the general themes in the answers to the questions and to capture critical concepts. All of the interviews were recorded with a mini-cassette recorder.

When administering the interview, after providing a few moments to get situated, the researcher read the participants the purposes of the study from the interview instrument. If the SME had no questions or concerns, the researcher asked each of the SMEs the questions without as little variation as possible. If the SME did not understand a question, or asked for the researcher to be more specific, clarification was given, but no other elaboration or description was offered.

Participants were given as much time as needed to answer the questions and provide examples, but were not guided by the researcher. If the researcher did not understand the SME's answer to the question, the researcher asked for clarification using generic requests such as "what do you mean by X," where X is the concept in question. The terminology and responses given by the subject matter experts was carefully reviewed and noted in order to prevent the researcher's pre-existing views of reverse engineering to interfere with the findings of the study. Specifically, the researcher was cautious to not use terminology in the interview which came from previous experience or previous interviews so that ideas, information, terminology, and the organization of concepts were not transferred from one SME to another through the researcher's interactions. As soon as possible after each interview was completed, the researcher transcribed the interview into text documents for further analysis.

3.4.5 Controlling Threats to Validity. To reduce potential bias from *translation competence* [208], in which the experts' responses are filtered through the researcher's conceptual lens before being presented, recordings and transcriptions were used to capture the terminology and findings of the study. Whenever possible (or where human comprehension was not necessary), automated methods were used to extract and organize the interview data. To avoid the potential bias from leading questions in which information from one participant flows by translation to the interviewer into subsequent interviews [208], a standard set of questions was used in each of the interviews. Additional questions were kept short and only asked in order to provide clarification or to get the SME participant to elaborate on a response. Additionally, to reduce potential bias from relying on one expert's viewpoint, five experts from separate areas were consulted and the data from each of their interviews was treated as impartially as possible. Detailed results from the subject matter expert interview study is presented in Chapter 5.

3.5 Study 3: Observational Study

An observational study was conducted where software reverse engineers performed a reverse engineering task during another "think-aloud" study. The data from the observational study was coded and segmented as in the case study above, and was analyzed to determine the conceptual and procedural aspects of reverse engineering executable programs (as well as to verify the information about conceptual and procedural aspects identified from the case study and subject matter expert study).

From the observational study, the participants' performance in the task was analyzed and interpreted to characterize the conceptual and procedural elements of their problem-solving processes as they sought to reverse engineer a program. The details of the design and analysis of the study are presented in Chapter 6 and the integrated conclusions of the dissertation are presented in Chapter 7.

The third study involved carefully studying people while they performed a challenging reverse engineering task. Watching people perform tasks can provide data about how they solve problems rather than how they think they solve them [135, 72]. When watching task performance, it is useful to have the participants verbalize their thoughts so they can be recorded and transcribed [72].

Verbal data from an observational "think aloud" study can indicate reasoning strategies that people use in problem solving, as well as the concepts, hypotheses, and information they use in the tasks [99, 214]. All of this information provides researchers insight into participants' mental models [55, 162]. Observational data allow researchers to infer goals, reasoning strategies, memory retrieval, and decisions used in the task and can aid in the mapping of a participants' actions to conceptual knowledge [199]. Thinking aloud during task performance has also been found to not significantly interfere with underlying cognitive processes as long as participants do not attempt to explain their thinking [72].

3.5.1 Relevance to the Research Problem. The purpose of this study was to develop a theory of program understanding from the observations of reverse engineers performing a reverse engineering task. The theory would relate the behavioral processes of reverse engineers in terms of transitions between abstract "states" in a person's problem-solving process. The theory can generate testable predictions about procedural knowledge in reverse engineering for future empirical study and provide a way to move toward reverse engineering tools that can learn and interact with their task environment.

3.5.2 Design of the Observational Study. In the observational study, reverse engineers performed a complicated task in which they were to analyze a program without having access to source code or documentation about the program. They were to reconstruct the functionality of an algorithm within the program by figuring out how the program worked from its assembly language representations. The observational study was carried out using a think-aloud protocol in which participants verbalized their thoughts as they reverse engineered the program.

In order to design the study so it answered the research questions appropriately, care had to be taken in the design of the task, the solicitation and instruction of participants, and the design of the data collection. First the selection of the task is described, then the solicitation of participants, and finally the methods of participant selection and instruction.

3.5.3 Selection of the Task. After undertaking the case study and conducting the subject matter expert interviews and literature review, it was clear that several considerations needed to be taken into account in the selection of the reverse engineering task used in the observational study. These considerations included:

- Choosing a domain - the approaches used in reverse engineering depend on the problem domain, but the domain of analyzing unprotected programs contains most of the comprehension aspects of the other domains of software reverse engineering.
- Factoring out prior knowledge - the primary elements of knowledge used in reverse engineering are knowledge of the assembly language, knowledge of the system call interface of the relevant operating system, and knowledge of how to use the tools.
- Sufficient challenge - the task should be difficult enough to be challenging for the participants, but should not contain so much detail that the analysis revolves around the details rather than problem solving in the task.

- Unclassified information - the task should not present issues which would introduce classified information or require the classification of the study.

Each of these considerations were taken into account when investigating whether to use a vulnerability analysis task, a malware analysis task, a software protection task, or the analysis of an unprotected program (the four domains which, as will be discussed in Chapter 5, were identified from the subject matter expert interviews). It was determined that using either a vulnerability analysis, malware analysis, or software protection task would limit the amount that could be learned about the general understanding of executable programs because each of those domains require much additional domain knowledge.

More complicated tasks would also make it more difficult to find participants with the skills to complete the task, and it would unnecessarily burden the participants in the study. Additionally, with vulnerabilities, malicious software, and software protection tasks, there was the concern that either the techniques observed or the findings could result in the classification of the research, which was undesirable.

Once it was decided to use an unprotected program for the study, additional concerns involving the availability and legality of reverse engineering an existing program, the participants' potential familiarity with the program, the size of the program, the desired difficulty of the task, and additional knowledge that would be required of the reverse engineers were all considered. Consideration of all of these factors led to the decision to use a *crackme* program, which is a program that is specifically designed for people to reverse engineer. The crackme program that was chosen was selected because it was expected to be unfamiliar to the participants, the goal of the crackme would be easy for participants to understand, and it would still be challenging enough for the participants to take at least an hour to solve it.

3.5.4 The Angler Task. The participants were told to complete a crackme problem called *Angler* that was downloaded from the *crackme.de* website [172] in

February 2010[1] (Figure 8). Angler is a type of crackme called a *keygen* (standing for key generation), a program which in a typical example presents the user with two text boxes: one for the user's name and one for the user to enter a serial number. The person is to reverse engineer the code for the program, and then either discover a valid key for a given user name, or write an algorithm that when given a user name produces a valid license key that unlocks the functionality of the program.

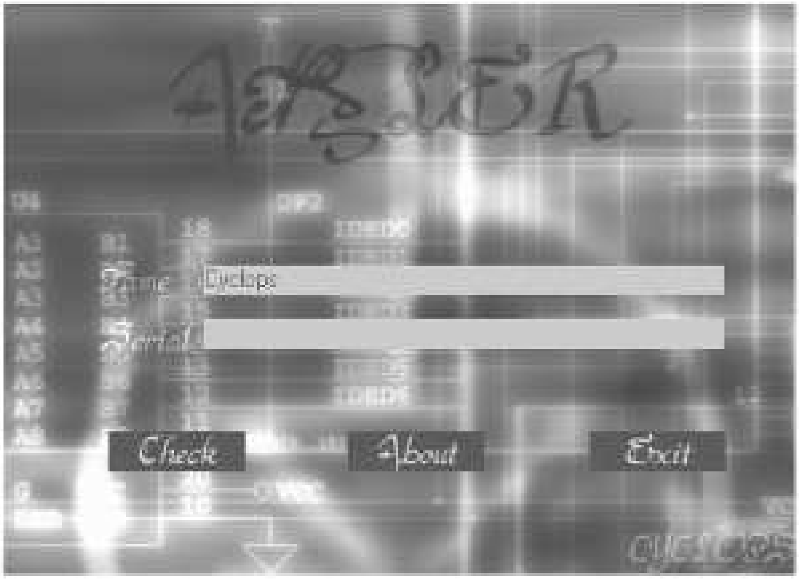
In a key-based software licensing scheme, the program verifies whether or not a user's license is valid. The verification code in a program can be reverse engineered to create an algorithm which produces a valid license key for a given name. This kind of keygen challenge involves applying many of the skills and tools which the subject matter experts indicated were also necessary for analyzing malicious software, probing software for bugs and vulnerabilities, and understanding protections: namely interpreting the functionality of a program and its behaviors from examining the assembly instructions and system calls used in a program.

The Angler crackme program (Figure 8) presents the participant with a semi-transparent visual window with two entries for text. It has three buttons, labeled "Check", "About" and "Exit." The button labeled "About" simply brings up a dialog box that says "Angler by Cyclops." When a user submits a name and serial number combination the application tests it for whether it is a valid combination. If the combination of name and serial number are valid, the program presents a text box saying "Correct serial! Now make a keygen and tut[2]!"

3.5.5 Selection of Participants. The researcher solicited participants for the verbal protocol through an e-mail invitation sent out to the Air Force Institute of Technology and to a cross-organizational reverse engineering working group representing organizations across Wright-Patterson Air Force Base. The solicitation produced

[1] Shortly after obtaining the crackme, the crackme.de website was taken down from the Internet. The Angler program can also be obtained from the website of the program's author at *http://cyclops.ueuo.com*.

[2] "tut" is short for tutorial. The participants were instructed that writing the tutorial was not important for the purposes of the task.



Name Cyclops

Serial

Figure 8 Main Window of the Angler Crackme Program.

four reverse engineers from the Air Force Research Laboratory, Aeronautical Systems Center, and Air Force Institute of Technology. For logistical reasons, the solicitation was limited to reverse engineers in the Dayton, Ohio area with access to Wright-Patterson Air Force Base. Participation was voluntary, and no monetary incentive was offered or given to the participants.

The participants were asked questions about their programming experience, their experience in reverse engineering, and their educational background. The solicitation explicitly requested that participants have knowledge of reverse engineering and experience using tools such as OllyDbg [140] WinDbg [125], Immunity Debugger [96], and the IDA Interactive Disassembler [88]. These tools were identified as the primary tools used in reverse engineering executable software from the subject matter expert study.

The participants' background knowledge in reverse engineering was not evaluated. Rather, the participants were instructed about the knowledge requirements from the solicitation and self-selected to participate. Implications about the diversity of the candidate pool are discussed in Chapter 6.

3.5.6 Data Collection. The task setup involved two computers, a participant computer and a researcher computer. The participant computer ran a Windows XP operating system that was hosted within a VirtualBox virtual machine. The virtual machine image was preloaded with all of the software tools (aside from custom tools) that were identified by the subject matter experts as important to solving reverse engineering tasks. The tools included IDA, OllyDbg, WinDbg, Immunity Debugger, PEiD [145], LordPE [49], and several other common tools.

The virtual machine also was loaded with reference documentation for the Intel instruction set architecture, the Win32 application programming interface, the Microsoft Developer Network (MSDN) library, documentation for the C, C++, Python, Java, and Lisp programming languages, and documentation to accompany the included reverse engineering tools.

The participant's task environment was instrumented to be accessed remotely through another workstation by sharing the participant's desktop over a virtual network computing (VNC) connection using the TightVNC server and client software [191]. The participant computer ran the virtual machine which had everything the participant needed to perform the reverse engineering task and a VNC server, while a machine in the same room, but on the cubicle opposite from the participant computer ran CamStudio [36] software and a VNC client which enabled the researcher to see all actions on the participant's computer screen. The researcher's computer was outfitted with a microphone which was wired into the participant's cubicle to enable the collection of combined audio and video data. The researcher conducted the study with himself as the participant in order to ensure all of the equipment worked properly and the task would provide the appropriate level of challenge.

Each participant was scheduled for a single reverse engineering session to be held in a data collection room on Wright-Patterson Air Force Base during July, 2011. When making the appointment for data collection, the reverse engineers were given the option to bring their own tools or tool plug-ins to be installed or to identify tools and plug-ins that the researcher would install in the virtual machine in advance. None of the participants expressed that this would be necessary given the reverse engineering set up that was offered.

Each participant visited the building individually and was escorted to the data collection room and given an overview of the research and instruction on how to verbalize thoughts during task performance. Each participant was explicitly instructed to only verbalize his or her thoughts and to not attempt to explain the task or thought processes. The researcher stressed the importance of only verbalizing thoughts rather than explaining and demonstrated examples of poor, acceptable, and high-quality concurrent verbalizations with a simulated coffee-making task to help the participants understand how they should verbalize during task performance. The participant was seated at a small cubicle in an unoccupied, quiet room in front of the participant computer which contained a mouse, keyboard, monitor, and the microphone. The participants were instructed as to the different reverse engineering tools available, the documentation available, and were permitted as much time as was needed to become familiar with the task environment. Paper was also available so participants could make notes, as recommended by Wood [208].

During the performance of the tasks, the researcher was seated out of the participant's view in the opposite cubicle. The researcher reminded participants to verbalize when they fell silent for more than a few seconds using simple prompts such as "please remember to verbalize during the task" and "remember to talk aloud" as discussed in Trickett and Trafton [194]. Other than those reminders, the researcher was silent throughout the task. The researcher also ensured voice and video capture worked properly and eliminated other potential distractions (such as keeping the lights in the room from automatically turning off during the participant's session).

The researcher monitored each participant's problem solving through the VNC connection to the participant's computer and took notes about the person's goals, apparent strategies, the concepts used to describe the problem, and problems faced in solving the reverse engineering challenge or using the tools. Audio data of each participant's verbalizations and video data of the researcher's computer monitor (showing all actions on the participant's computer monitor) were recorded using the CamStudio software.

After the task, participants were asked to recall what they thought their strategies were, what parts of the task they thought were the most difficult, what would have made the task easier, and what they felt they needed to pay attention to. The analysis and results of the observational study are described in Chapter 6.

3.6 Related Research

Other researchers have performed studies using similar methodologies to extract knowledge of how people perform problem-solving activities in complex real-world domains. Seamster et al. [177] performed a cognitive task analysis of air traffic controllers to determine the elements of expertise. In that study, paper problem solving, performance modeling, and structured problem solving were used to elicit knowledge from 18 air traffic controllers consisting of experts, intermediates, and novices. In these tasks, experts were found to use fewer strategies, but showed more strategy types. Experts also showed more workload management than less experienced air traffic controllers.

Roth et al. [159] performed a cognitive task analysis of how train dispatchers manage and control trains to improve efficiency and safety operations. That study involved two days of field observation, structured interviews with expert train dispatchers, and a second day of field observations. The results were used to determine what aspects make train dispatching difficult. The results characterized activities in terms of different strategies for adapting and planning ahead, maintaining a big picture, and acting proactively.

Pirolli and Card [150] presents the results of a cognitive task analysis of sensemaking in intelligence analysis work. The paper alludes to interviews and verbal protocols but does not describe their analysis in detail. The results attempted to provide a "broad brush" characterization of the cost structure of trading off between exploration and enrichment; trading off scanning, recognizing, and selecting items for analysis; shifting attentional control; and performing follow-up searches.

Dixon et al. [62] performed a cognitive task analysis of unmanned aerial vehicle (UAV) operators to include human factors into the system engineering design of unmanned aerial vehicles. In that study, a goal-directed task analysis [68] was carried out along with work domain analysis and a control task analysis. Some of the insights gained from that task analysis were that maps used by ground communication personnel are not appropriate as input to the search mechanisms of the UAV system. It also outlined that the team roles that UAV operator, video analyst, and ground searchers have to play in performing search and rescue operations. The results from that task analysis helped UAV developers ask the correct questions about how humans would use the UAVs earlier in the development process.

Crandall and Getchell-Reiter [54] used a knowledge elicitation method called the *critical decision method* to elicit indicators of sepsis from neonatal intensive care nurses. Baxter et al. [14] performed a cognitive task analysis of a neonatal intensive care unit. The task analysis collected data from context familiarization meetings, a critical decision method [54], and observations. The paper describes the development of a cue inventory, a situation assessment record, and a list of temporal issues.

Researchers have also studied how people understand programs, albeit with different aims and from different perspectives than those in this dissertation. Hendry et al. [86] had 232 students sketch out how a search engine works in order to study their mental models of complex systems. From this analysis, they constructed a conceptual metaphor "a search engine is a series of text transformations." Storey et al. [184] researched programmers' conceptual models in order to define design recommendations to help people construct mental models when programming. The design recommendations were that programming tools should have elements to enhance bottom-up and top-down comprehension, integrate bottom-up and top-down approaches, facilitate navigation, provide orientation cues, and reduce disorientation.

Vans [200] watched maintenance programmers and classified their representation of programmers' mental models into *the situation model*, *the program model*, and the *domain model*. In the top-down model, programmers chunk information about the program, instructions, modules, the problem domain, and inferred plans and synthesize it all into their existing knowledge base. The knowledge base includes schemas of concepts and concept families used in developing the program model, situation model, and domain model of a program [203, 202].

Fix, et al. [74] presented an experiment aimed at verifying five characteristics of mental representations. These representations involved:

- The hierarchical structure of the program,
- An explicit mapping from the code to the goals of the program,
- The ability to recognize recurring patterns,
- Connecting pieces of knowledge, and
- Grounding concepts in the program text.

Each of these studies elicited knowledge through different combinations of interviews, assessments, and observations to determine how people solve complex problems. This study describes a similar undertaking to understand the cognitive work of software reverse engineers. Knowledge elicitation methods are the most appropriate way to important to developing an understanding of how reverse engineers make sense of programs. The methods are aimed at eliciting the goals and activities reverse engineers undertake in their tasks, domain-specific knowledge and how it is used, and a description of the process of sensemaking as applied to understanding programs from assembly language.

Additionally, the knowledge and behavioral processes represented through this dissertation can be used as descriptive accounts to provide the basis for more precise models of cognitive processes in reverse engineering in the future.

3.7 Validity

The methodology was designed in order to maximize the validity of the research results. Validity in qualitative research involves construct validity, internal validity, external validity, and reliability [158]. Construct validity is concerned with how well the measured variables reflect the concepts or "constructs" that are intended to be measured [117]. Construct validity in this dissertation involves ensuring the qualitative methods and analysis methods represent the constructs of interest. The constructs of interest in this dissertation are elicited from the data in a bottom-up fashion which is described further in subsequent chapters, so the validity of the constructs is established within the overall methodology itself.

Internal validity involves the strength of the causal relations in the hypotheses [117, 158]. Since the research methods used in this dissertation did not involve the test of a hypothesis, internal validity relates to the hypotheses that were uncovered in the course of this research. To ensure internal validity, the data from each of the studies were collected carefully and systematically, and the data collection and analysis processes are described in detail in Sections 3.3, 3.4, and 3.5 so they may be replicated by other researchers.

External validity is a type of validity that deals with how well findings generalize outside of the particular case being investigated [117, 163]. External validity in this research method is established through conducting multiple studies to triangulate the findings between them, rather than relying on a single study. The three research methods provide three different sets of circumstances from which to evaluate the elements of conceptual and procedural knowledge involved in reverse engineering executable programs. The findings from the case study, subject matter expert study, and observational study should converge to produce similar (or compatible) findings, which they do.

Reliability concerns the extent to which the research is repeatable, particularly by different researchers. Reliability in this research is established through careful, controlled, and documented data collection, documentation of decisions made in the research process, and explicit description of the logic linking the data to the conclusions. These methods are described above and in Chapters 4, 5, and 6.

3.8 Conclusion

This chapter briefly introduced the three methods that were undertaken to answer the research questions presented in Chapter 1. Situational, verbal, and observational approaches are used to collect data, and primarily qualitative analysis methods are used to analyze the data, as is appropriate for an exploratory study. The research methods involve seeking to understand the conceptual and procedural elements of the task by analyzing situational aspects of task performance, talking in detail with expert reverse engineers and watching reverse engineers perform reverse engineering tasks.

The next chapter presents the case study of a situation in a reverse engineering task referred to in Section 3.3. After that, Chapter 5 discusses the semi-structured interview study with subject matter expert reverse engineers introduced in Section 3.4. Chapter 6 presents the data and analysis from the observational study introduced in Section 3.5. Finally, Chapter 7 re-addresses the research questions introduced in the dissertation and presents the results and conclusions from the overall methodology approach.

4. Case Study of a Reverse Engineering Task

4.1 Introduction

The overall methodological approach of this dissertation, presented in Chapter 3, involves a case study of a reverse engineering situation, subject matter expert interviews, and an observational study, all aimed at describing the conceptual and procedural aspects of knowledge involved in reverse engineering executable programs.

This chapter presents the first study, a case study of a reverse engineering situation, to investigate the role of various conceptual and procedural aspects of the situation in the process of reverse engineering an executable program. First, the chapter outlines the methods used to collect, transform, and analyze the data used in the case study. Then, it presents a situation from a simple reverse engineering task and explores the task characteristics, task environment characteristics, and knowledge characteristics that are involved with the task. Finally, the chapter then discusses the results of the case study and its implications in the context of improving our understanding of how people make sense of executable programs.

This case investigates the situational factors involved in a reverse engineering task. This study draws data from three primary sources:

1. Verbal data from a one-participant think-aloud protocol of task performance,
2. Analysis of a reverse engineering environment (the OllyDbg debugger [140]) used throughout performance of the task, and
3. Notes and diagrams taken during a number of subsequent completions of the reverse engineering task.

A video recording of the computer screen during the task and the associated verbalizations during the task were collected using the CamStudio screen capturing software [36] with a microphone on a computer running a Windows XP operating system [126]. OllyDbg version 2.01 was used as the primary reverse engineering tool with the HxD hex editor [123] used for hexadecimal editing. This same task environment was later analyzed for information content and affordances and was used to complete subsequent unrecorded trials through the task in which notes were taken.

This section first gives an overview of the task, then it describes the data collection and analysis related to the think aloud protocol and the task environment. After that, the situational elements of reverse engineering are discussed.

4.1.1 Reverse Engineering a "Crackme" Program.

The reverse engineering situation investigated in this case study is that of solving a simple reverse engineering challenge delivered in an executable program called a *crackme*. Crackmes are executable programs which provide reverse engineers challenges to practice and hone reverse engineering skills in a legal and ethical manner. A typical crackme is presented as a small executable program packaged in a compressed file along with a text file containing instructions on the objective and rules of the task.

Crackmes are developed in Visual Basic, C, C++, .NET languages, or directly in assembly language. The tasks involved in solving a crackme can range in difficulty from those that would be simple for a novice reverse engineer to those which would be very challenging for a team of professional reverse engineers [172]. On various websites, people can find crackme challenges that vary based on their difficulty to complete, the operating systems they on which they run, the programming language used to develop them, and the compilers used to convert them to machine code. Crackme challenges can involve tasks such as:

- Finding a program's serial number hidden in memory,
- Reverse engineering a cryptographic algorithm,
- Writing a key generation routine to reconstruct a serial number,
- Deobfuscating a program, and
- Subverting software protection mechanisms.

Reverse engineers examine crackmes from disassembled instructions with no source code available [172].

4.1.2 The "Splish" Task. To simplify the analysis of the understanding task (and focus on the aspects involved in making sense of the program rather than the program itself), the crackme chosen is a from crackmes in the "very easy" category. Also, the crackme is from a group of crackmes that the researcher had solved four years previously, so the challenge was known to be easy to complete and not contain malicious code.

The crackme challenge used in the case study is a program called *Splish.exe*. Upon starting the program, a splash screen appears which is labeled "The Reverse Engineering Academy," which was involved with developing a series of reverse engineering tasks for training reverse engineers on the website *www.crackme.de*[172]. The application is 232 KB in size containing 237,568 bytes worth of instructions, data, and uninitialized values (232 KB $\times 2^{10}$ bytes per KB $= 237,568$ bytes). Source code for the program is not available, nor are debugging symbols.

4.2 Information and Affordances in the Task Environment

Before discussing the process of reverse engineering the executable, it is import to understand the elements of the situation contributed by the task environment. Data about the OllyDbg task environment was collected by inspecting the OllyDbg debugger with the Splish program loaded in memory (together these constitute the *task environment* for the majority of the task). Within each pane of the debugger's user interface, the various sources of information and controls were visited through a breadth-first approach.

The elements were listed and categorized as to whether they represented information elements, affordances, or both. Information elements were classified according to the information type displayed (for instance "text," "hexadecimal values," and "icon") and the meaning of each of the information elements were characterized (for instance "a comment about the instruction", "the bytes representing the instruction's opcode", "the current value of the instruction pointer").

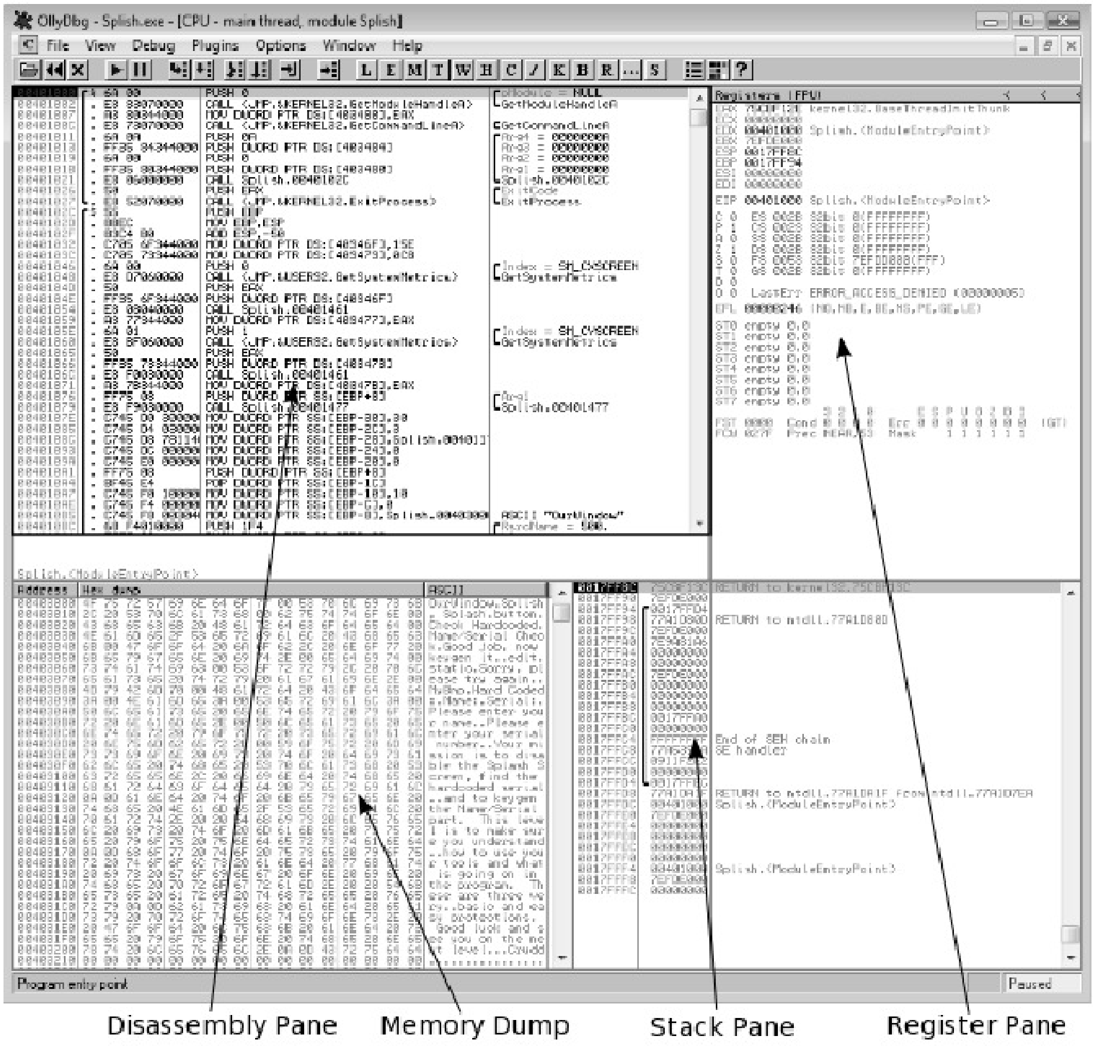
Information elements were identified as any simple or complex signals (like text values, colors, or shapes) presented to the user. Because of their great number, the individual values from the program (the program's instructions and data values) were not recorded. Instead, more abstract labels were given to represent the types of information and affordances involved.

Affordances were identified as any user interface control with which a user can change the state of the program, the debugger, or the representation presented. Elements that represented affordances in the task environment were classified according to the actions required to activate them (for instance "click," "double-click," and "mouse over") and the functions that they performed ("set selected row," "toggle checkbox," and "open comment dialog"). The affordances were described in terms of their functionality. Because of the great number of affordances, the user interface of the debugger was only represented in as much as it was exercised in the progress of the reverse engineering task. OllyDbg is a robust and capable reverse engineering tool, and it has a number of other features which provide additional information and functionality which were not explored because they are out of the scope of the research effort.

4.2.1 Information in the Splish Task. The OllyDbg environment presents a lot of different information to a reverse engineer simultaneously. In OllyDbg, the main window is the CPU Window, which consists of four primary information panes (Figure 9):

- Disassembly pane,
- Register pane,
- Program stack, and
- Memory dump.

There is also an information pane directly below the disassembly pane which provides status information related to the instruction selected in the disassembly


OllyDbg - Splish.exe - [CPU - main thread, module Splish]
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S
Registers (FPU)
Address
Hex dump
ASCII
Disassembly Pane
Memory Dump
Stack Pane
Register Pane

```
00401000  /$ 6A 00          PUSH 0                                   ; /pModule = NULL
00401002  |. E8 33070000    CALL <JMP.&KERNEL32.GetModuleHandleA>    ; \GetModuleHandleA
00401007  |. A3 80344000    MOV DWORD PTR DS:[403480],EAX
0040100C  |. E8 73070000    CALL <JMP.&KERNEL32.GetCommandLineA>     ; [GetCommandLineA
00401011  |. 6A 0A          PUSH 0A                                  ; /Arg4 = 0000000A
00401013  |. FF35 84344000  PUSH DWORD PTR DS:[403484]               ; |Arg3 = 00000000
00401019  |. 6A 00          PUSH 0                                   ; |Arg2 = 00000000
0040101B  |. FF35 80344000  PUSH DWORD PTR DS:[403480]               ; |Arg1 = 00000000
00401021  |. E8 06000000    CALL Splish.0040102C                     ; \Splish.0040102C
00401026  |. 50             PUSH EAX                                 ; /ExitCode
00401027  \. E8 52070000    CALL <JMP.&KERNEL32.ExitProcess>         ; \ExitProcess
0040102C  /$ 55             PUSH EBP
0040102D  |. 8BEC           MOV EBP,ESP
0040102F  |. 83C4 B0        ADD ESP,-50
00401032  |. C705 6F344000  MOV DWORD PTR DS:[40346F],15E
0040103C  |. C705 73344000  MOV DWORD PTR DS:[403473],0C8
00401046  |. 6A 00          PUSH 0                                   ; /Index = SM_CXSCREEN
00401048  |. E8 D7060000    CALL <JMP.&USER32.GetSystemMetrics>      ; \GetSystemMetrics
0040104D  |. 50             PUSH EAX
0040104E  |. FF35 6F344000  PUSH DWORD PTR DS:[40346F]
00401054  |. E8 08040000    CALL Splish.00401461
00401059  |. A3 77344000    MOV DWORD PTR DS:[403477],EAX
0040105E  |. 6A 01          PUSH 1                                   ; /Index = SM_CYSCREEN
00401060  |. E8 BF060000    CALL <JMP.&USER32.GetSystemMetrics>      ; \GetSystemMetrics
00401065  |. 50             PUSH EAX
00401066  |. FF35 73344000  PUSH DWORD PTR DS:[403473]
0040106C  |. E8 F0030000    CALL Splish.00401461
00401071  |. A3 7B344000    MOV DWORD PTR DS:[40347B],EAX
00401076  |. FF75 08        PUSH DWORD PTR SS:[EBP+8]                ; /Arg1
00401079  |. E8 F9030000    CALL Splish.00401477                     ; \Splish.00401477
0040107E  |. C745 D0 300000 MOV DWORD PTR SS:[EBP-30],30
00401085  |. C745 D4 030000 MOV DWORD PTR SS:[EBP-2C],3
0040108C  |. C745 D8 781140 MOV DWORD PTR SS:[EBP-28],Splish.0040117
00401093  |. C745 DC 000000 MOV DWORD PTR SS:[EBP-24],0
0040109A  |. C745 E0 000000 MOV DWORD PTR SS:[EBP-20],0
004010A1  |. FF75 08        PUSH DWORD PTR SS:[EBP+8]
004010A4  |. 8F45 E4        POP DWORD PTR SS:[EBP-1C]
004010A7  |. C745 F0 100000 MOV DWORD PTR SS:[EBP-10],10
004010AE  |. C745 F4 000000 MOV DWORD PTR SS:[EBP-C],0
004010B5  |. C745 F8 003040 MOV DWORD PTR SS:[EBP-8],Splish.00403000  ASCII "OurWindow"
004010BC  |. 68 F4010000    PUSH 1F4                                 ; /RsrcName = 500.

Splish.<ModuleEntryPoint>
```

```
Registers (FPU)
EAX 75C8F12A kernel32.BaseThreadInitThunk
ECX 00000000
EDX 00401000 Splish.<ModuleEntryPoint>
EBX 7EFDE000
ESP 0017FF8C
EBP 0017FF94
ESI 00000000
EDI 00000000

EIP 00401000 Splish.<ModuleEntryPoint>

C 0  ES 002B 32bit 0(FFFFFFFF)
P 1  CS 0023 32bit 0(FFFFFFFF)
A 0  SS 002B 32bit 0(FFFFFFFF)
Z 1  DS 002B 32bit 0(FFFFFFFF)
S 0  FS 0053 32bit 7EFDD000(FFF)
T 0  GS 002B 32bit 0(FFFFFFFF)
D 0
O 0  LastErr ERROR_ACCESS_DENIED (00000005)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address   Hex dump                                         ASCII
00403000  4F 75 72 57 69 6E 64 6F 77 00 53 70 6C 69 73 68  OurWindow.Splish
00403010  2C 20 53 70 6C 61 73 68 00 62 75 74 74 6F 6E 00  , Splash.button.
00403020  43 68 65 63 6B 20 48 61 72 64 63 6F 64 65 64 00  Check Hardcoded.
00403030  4E 61 6D 65 2F 53 65 72 69 61 6C 20 43 68 65 63  Name/Serial Chec
00403040  6B 00 47 6F 6F 64 20 6A 6F 62 2C 20 6E 6F 77 20  k.Good job, now
00403050  6B 65 79 67 65 6E 20 69 74 2E 00 65 64 69 74 00  keygen it..edit.
00403060  73 74 61 74 69 63 00 53 6F 72 72 79 2C 20 70 6C  static.Sorry, pl
00403070  65 61 73 65 20 74 72 79 20 61 67 61 69 6E 2E 00  ease try again..
00403080  4D 79 42 6F 78 00 48 61 72 64 20 43 6F 64 65 64  MyBox.Hard Coded
00403090  3A 00 4E 61 6D 65 3A 00 53 65 72 69 61 6C 3A 00  :.Name:.Serial:.
004030A0  50 6C 65 61 73 65 20 65 6E 74 65 72 20 79 6F 75  Please enter you
004030B0  72 20 6E 61 6D 65 2E 00 50 6C 65 61 73 65 20 65  r name..Please e
004030C0  6E 74 65 72 20 79 6F 75 72 20 73 65 72 69 61 6C  nter your serial
004030D0  20 6E 75 6D 62 65 72 2E 00 59 6F 75 72 20 6D 69   number..Your mi
004030E0  73 73 69 6F 6E 20 69 73 20 74 6F 20 64 69 73 61  ssion is to disa
004030F0  62 6C 65 20 74 68 65 20 53 70 6C 61 73 68 20 53  ble the Splash S
00403100  63 72 65 65 6E 2C 20 66 69 6E 64 20 74 68 65 20  creen, find the
00403110  68 61 72 64 63 6F 64 65 64 20 73 65 72 69 61 6C  hardcoded serial
00403120  0D 0A 61 6E 64 20 74 6F 20 6B 65 79 67 65 6E 20  ..and to keygen
00403130  74 68 65 20 4E 61 6D 65 2F 53 65 72 69 61 6C 20  the Name/Serial
00403140  70 61 72 74 2E 20 20 54 68 69 73 20 6C 65 76 65  part.  This leve
00403150  6C 20 69 73 20 74 6F 20 6D 61 6B 65 20 73 75 72  l is to make sur
00403160  65 20 79 6F 75 20 75 6E 64 65 72 73 74 61 6E 64  e you understand
00403170  0D 0A 68 6F 77 20 74 6F 20 75 73 65 20 79 6F 75  ..how to use you
00403180  72 20 74 6F 6F 6C 73 20 61 6E 64 20 77 68 61 74  r tools and what
00403190  20 69 73 20 67 6F 69 6E 67 20 6F 6E 20 69 6E 20   is going on in
004031A0  74 68 65 20 70 72 6F 67 72 61 6D 2E 20 20 54 68  the program.  Th
004031B0  65 73 65 20 61 72 65 20 74 68 72 65 65 20 76 65  ese are three ve
004031C0  72 79 0D 0A 62 61 73 69 63 20 61 6E 64 20 65 61  ry..basic and ea
004031D0  73 79 20 70 72 6F 74 65 63 74 69 6F 6E 73 2E 20  sy protections.
004031E0  20 47 6F 6F 64 20 6C 75 63 6B 20 61 6E 64 20 73   Good luck and s
004031F0  65 65 20 79 6F 75 20 6F 6E 20 74 68 65 20 6E 65  ee you on the ne
00403200  78 74 20 6C 65 76 65 6C 2E 00 0D 43 72 75 64 64  xt level...Crudd
00403210  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0017FF8C  75C8F13C  RETURN to kernel32.75C8F13C
0017FF90  7EFDE000
0017FF94  0017FFD4
0017FF98  77A1D0BD  RETURN to ntdll.77A1D0BD
0017FF9C  7EFDE000
0017FFA0  7E948146
0017FFA4  00000000
0017FFA8  00000000
0017FFAC  7EFDE000
0017FFB0  00000000
0017FFB4  00000000
0017FFB8  00000000
0017FFBC  0017FFA0
0017FFC0  00000000
0017FFC4  FFFFFFFF  End of SEH chain
0017FFC8  77A63[illegible]  SE handler
0017FFCC  0911F5[illegible]
0017FFD0  00000000
0017FFD4  0017FFEC
0017FFD8  77A1DA1F  RETURN to ntdll.77A1DA1F from ntdll.77A1D7EA
0017FFDC  00401000  Splish.<ModuleEntryPoint>
0017FFE0  7EFDE000
0017FFE4  00000000
0017FFE8  00000000
0017FFEC  00000000
0017FFF0  00000000
0017FFF4  00401000  Splish.<ModuleEntryPoint>
0017FFF8  7EFDE000
0017FFFC  00000000
```

Program entry point

Paused

Figure 9   Information Presented by OllyDbg.

pane. Typical status messages include the calculated address offset for the current instruction or the ASCII translation of a selected hexadecimal value. The status messages vary depending on the type of instruction (i.e., `JMP`, `PUSH`, or `CALL`) that the current instruction contains. Since each pane is re-sizable, the window can contain as much or as little information as can fit on a person's computer display (Figure 9).

4.2.2 Disassembly Pane. Depending on screen resolution and the size of the pane, the disassembly pane can display many rows of assembly instructions (41 rows in Figure 9). Each row has an address field, a byte field, an instruction field and a comment field. The address field contains a hexadecimal value representing an address in memory such as 0x00403C3D or 0x760F081C. An address space in the disassembly pane beginning with 0x0040xxxx tells the user that the program is likely executing an instruction in the program's code rather than code in a system library. If the addresses in the disassembly pane begin with 0x7Fxxxxxx, the program is likely executing a dynamic link library (DLL) such as NTDLL.dll or USER32.dll that the program has mapped into its address space.

The byte field displays a variable number of hexadecimal bytes that correspond to the instruction being executed. The bytes displayed represent the binary string the processor takes as input during an instruction execution cycle. For example, the instruction pointed at in Figure 9 is:

```
MOV DWORD PTR DS:[403478], EAX
```

The opcode for this instruction is 0xA37B344000, which the processor reads as five bytes worth of binary values (0xA3 ,0x7B, 0x34, 0x40, 0x00) encoded in little-endian format.

The instruction field contains the assembly language representation of the instruction bytes. Most instructions consist of an mnemonic and zero or more operands to create an instruction opcode. The operands refer to registers, addresses in memory, or immediate values [97]. The disassembler parses each of these bytes and determines from them the instruction type (MOV), the source (EAX), and the destination. The destination address is computed as the memory address in the segment pointed to by the DS register with the specified offset (0x403478).

A light gray horizontal line in the disassembly pane indicates the currently executing instruction. As instructions execute in the debugger, the gray line moves down one row at a time until the program hits an instruction that changes control flow, such as a JMP, CALL or RETN instruction. A person reads information from the disassembly pane to understand the current state of the program, the instruction at the current state and as one input for making inferences about the next state of the program. The instructions are primarily what the user of the debugger reads in order to understand and make sense of a program.

4.2.3 Register Pane. The register pane displays the labels and values of several sets of general purpose registers (`EAX`, `ECX`, `EDX`, `EBX`, `ESI` and `EDI`), the stack base pointer (`EBP`), the stack pointer (`ESP`), the instruction pointer (`EIP`), the `EFLAGS` register, the offset pointed to by each of the section registers, and the eight floating point registers. When a person pauses the debugger at an instruction, the register pane provides the person with the value of the registers at that point in program execution. The instruction pointer (`EIP`) displays a text label with the name of the program module being executed, which gives the user an indication of whether the program's code is executing or whether code from a DLL is executing.

Information in the register pane changes very rapidly while the program executes. The instruction pointer changes with every instruction and the stack pointer (`ESP`) changes with every operation that modifies the program stack. The general purpose registers change at nearly each instruction and the `EFLAGS` register changes often to show status messages for instructions, exceptions, traps, tests, and conditions. The changes happen so rapidly, that people are not able to pay attention to how the registers' values change when quickly stepping through a program's instructions. If a person is paying attention to the values of particular registers, it requires them to slow down their debugging and mentally keep track of how the data flows to and from CPU registers.

4.2.4 Program Stack. The stack pane has three columns, one for hexadecimal values representing the memory address of a stack entry, a second for the stack value itself, and the third column for text comments. The stack pane also contains a large number of rows of stack contents (27 rows in Figure 9).

The stack and the stack pane change with each stack operation that the debugger executes. When program execution traces into a function, the processor places the instruction that follows the function call (the return address) onto the stack, and when the function executes the `RETN` instruction the execution returns to that address. When a value is pushed onto the stack, OllyDbg shifts down all of the contents of the stack pane so that the new value pushed to the logical stack is at the top of the stack pane. While a person steps through a program's instructions in the debugger, the program stack moves up and down rapidly, representing new values being pushed to the top of the stack and old values being popped off the top of the stack.

Although the top of the stack pane usually corresponds to the top of the program stack, the stack can me scrolled up or down, which can make it visually difficult to indicate where the stack begins and ends. A person would have to look at the address of the base pointer, find that address in the stack pane, and interpret the location of that address as the bottom of the stack. If the person scrolls up or down, the stack pointer is no longer at the top of the stack pane, so the person would need to calculate the location of the stack pane as well. Since the stack moves at every stack-based instruction this information constantly changes.

4.2.5 Memory Dump. The memory dump contains several rows of memory data (32 rows in Figure 9). Each row in the dump pane has a four-byte memory address and four columns of four-byte hexadecimal values corresponding to those locations in memory. To the right of the four-byte memory dump, an additional column in the pane provides an ASCII or Unicode text representation of the byte values.

The bytes can correspond to any parts of a program's data, but are typically used to view data or resource sections (`.data`, `.rdata`, and `.rsrc`) mapped into a program's address space in memory. The program's data sections are where strings, global variables, and other less-frequently-changing items are stored in a program. In a typical task, the top address could be `0x403000`, which would normally correspond to the offset of the `.data` section pointed to by the program's file header.

Through the process of reverse engineering most programs, the values in the memory dump remain relatively stable compared with the registers and the stack. Memory values change when instructions move data directly into memory addresses such as the `MOV` instruction above. However, when stepping through a program, the debugger does not show which memory values are being read or modified. There is no mechanism to have the screen scroll where memory is read or changed, so many memory related changes go without being noticed. The reverse engineers usually ignore the memory values that are accessed and changed by instructions unless they noticed and mentally linked the address value with some other meaning in the program.

4.2.6 Primary Affordances. The OllyDbg software has a number of affordances related to file operations in the operating system (like most Win32 programs) as well as a number of debugging-specific functions. The affordances that are most commonly used are those corresponding to the "Step in," "Step over," "Rewind," and "Execute" functions.

Stepping into a program moves the instruction pointer one instruction forward, and simulates the output of the instruction by modifying the registers, memory, and any external files or data. If the program comes to a control flow instruction like an unconditional jump (`JMP`) or a conditional jump instruction in which the test condition has been met, then the instruction pointer will follow the jump to its target address. Stepping over is similar to stepping in with the primary exception that when the instruction pointer is pointing to a `CALL` instruction, the program does not follow the `CALL`, but instead executes the code called by the `CALL` instruction and moves the instruction pointer to the instruction directly after the `CALL` is returned.

4.2.7 Task Environment Summary. This section discussed the information elements present in the OllyDbg debugger in order to talk about the various forms of information and affordances used in the task environments. The information comes in the forms of ASCII text, hexadecimal, instructions, decimals, and other values. Each component of information presented by the task environment has a different meaning, and it is up to the reverse engineer to use his or her knowledge to interpret the contextual meanings of these informational symbols.

The next section walks through a reverse engineering task performed by the researcher to motivate the discussion of other situational factors involved in understanding an executable program. Following that, verbal data from the task is analyzed to look at the conceptual and procedural knowledge related to task performance.

4.3 Walk-Through of Performance in the Splish Task

In order to set the stage for investigating the role that the situation plays on conceptual and procedural knowledge in a reverse engineering task, this section presents a "walk through" of the situation for one portion of a reverse engineering task.

Performance in the Splish task was observed from video data of the researcher performing the task and notes were taken to indicate the actions that are taken in the task environment. Since the researcher performed the task, the researcher is referred to as "the participant" when discussing performance in the task in order to disambiguate the roles of researcher and reverse engineer. Since eye tracking was not used, the focus of the participant's visual attention is indirectly inferred by elements like the current instruction indicator when stepping through code and mouse movements hovering over text, buttons, or other elements of the task environment, combined with verbalizations that take place at the same moment indicating reference to the location. Thinking, planning, and reasoning are inferred when the verbalizations do not refer to objects that are visible in the task environment, but instead refer to other elements, goals, plans, or things outside of the current focus indicated by the participant's mouse movements or other focus items in the task environment. *Objects* as used in this sense refer to a thing or an entity, rather than the use of the term in computer programming to mean data structures grouped with their functions or methods. The term "object" is used throughout the dissertation to refer to things in a program that do not seem like formal objects, but which are used as such by reverse engineers (such as a sequence of instructions or code location).

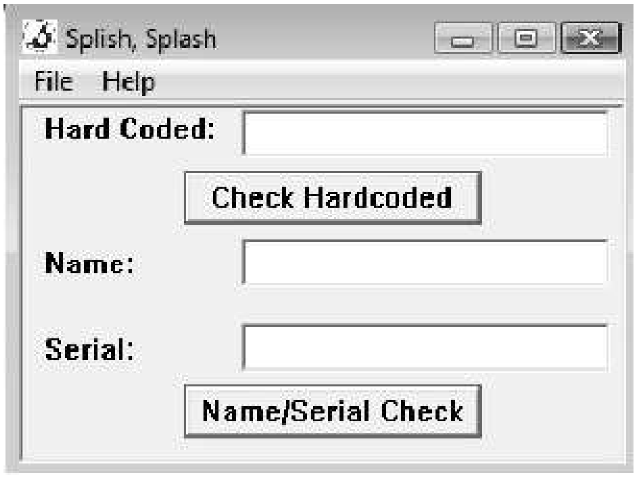


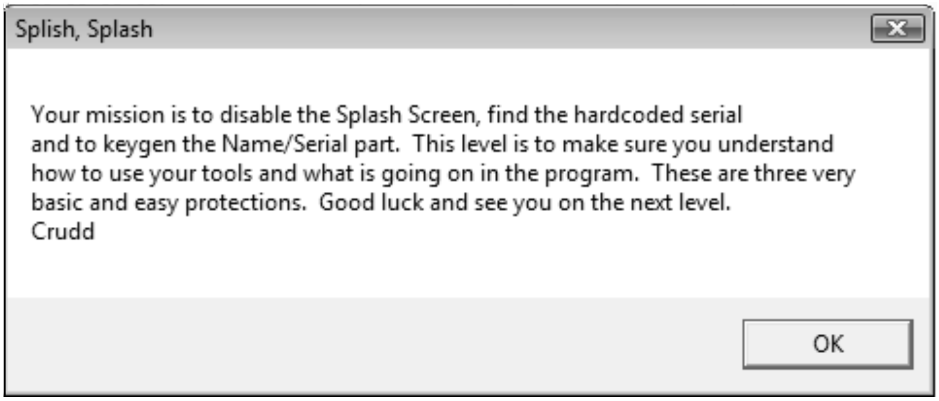
Figure 10 The Splish Crackme Program.

4.3.1 Exploring the Program. At the beginning of solving the crackme, the objectives of the challenge are not known. The participant starts the process by loading the program in the debugger environment, which presents him with the program's assembly instructions. The participant looks at the code, and then decides to run the program in the debugger to see what it presents to the user.

The participant sees a number of text labels, fields, and buttons. The first label is "Hard Coded:" next to a text field, with a button "Check Hardcoded" beneath it. The second two labels are "Name:" and "Serial:", both of which have text fields to the right of them, and with a button labeled "Name/Serial Check" beneath the two text fields. The program has a menu ribbon with "File" and "Help" menus available.

The participant tries entering text in the fields and pressing the buttons, and sees a dialog box that says "Sorry, please try again." The participant continues this to realize that entering anything but the correct serial number or a valid name and serial number combination will result in the "try again" dialog box. The participant explores the "File" and "Help" menus and finds an "About" menu in the "Help" ribbon. Clicking the "Help" ribbon produces a dialog box with the instructions (Figure 10).

4.3.2 Determining the Goal. The participant reads the instructions for the crackme, which make it clear that the goal of the Splish task is to:


Splish, Splash

Your mission is to disable the Splash Screen, find the hardcoded serial
and to keygen the Name/Serial part. This level is to make sure you understand
how to use your tools and what is going on in the program. These are three very
basic and easy protections. Good luck and see you on the next level.
Crudd

Figure 11 Splish's Embedded Instructions.

1. Disable the splash screen that appears at program start,
2. Find the serial number embedded in the application and , and
3. Reverse engineer the application's serial number generation algorithm and write a program that generates an acceptable serial number for any name (Figure 11).

The first task, disabling the splash screen, involves searching the program to find a memory location in the program that creates and displays the splash screen and then modifying the instructions at that location so the splash screen does not appear. This requires navigating through assembly code to find a particular location and mentally linking observable events in the program to memory locations in the program where instructions carry out that functionality.

The second task is to search for a "hardcoded" serial number in the assembly instructions and data of the program. This involves running or debugging the program to determine its functionality at a high level, navigating through the program's assembly code to determine the different code structures, integrating high-level and low-level information to determine the location of the algorithm that handles the serial number, and tracing through the data flow of the serial number processing algorithm to locate the serial number in the program's memory.

The third task, making a "keygen" for the name and serial algorithm, involves locating the name and serial number processing algorithm, tracing data flow through the algorithm to determine how the algorithm works at a high level, and generating a program which performs the functionality as the algorithm, but in reverse. The program should be able to take any given name and produce a serial number for that name which will be accepted by the program's serial number processing algorithm.

All three of these tasks involve the combination of navigation, reasoning, identification and matching, and data flow tracing. The first task includes modifying the bytes of the program, and the third task involves understanding and recasting an algorithm by writing a program to perform the functionality. The first of these tasks is examined in detail.

4.3.3 Determining an Approach. When the program is run, a splash screen window appears with the title "The Reverse Engineering Academy" (Figure 12). In order to disable the splash screen so the program starts without showing it, a person has to:

- Locate the program's code that presents the splash screen,
- Change the program so that the splash screen code has no effect, and
- Write the changes at the byte level to the program's on-disk image so that all modifications to the program will be in effect when the program is restarted.

If someone is able to successfully perform all three of these subtasks, the program will not display the splash screen when it is started. The participant does not have the knowledge of these subtasks from the beginning of the task. Instead, the participant has to construct the task strategy while working in the task and gaining more information from the task environment and from trying out different approaches in the task.

When the participant runs the program from the debugger (or outside the debugger) the first thing visible is the splash screen, followed by the main window of the program. The participant presses buttons and manipulates the affordances of the program in order to determine what the main goal of the task is. Once the goal is

The Reverse Engineering Academy

Figure 12 Disabling the Splash Screen.

discovered, the participant has to develop a plan of action to solve the goal while performing activities to gain more information from the task environment.

For the first of these goals (disabling the splash screen), it is unclear from the start what the participant is to do, so the participant runs the program in the debugger to see what happens. While running the program in the debugger, a person gains information that helps connect the code from the assembly instructions with the information presented by the events from the program. When an event from the program corresponds to the debugger executing code at a particular instruction, the participant can infer that the code at that instruction location caused the program to perform that behavior.

4.3.4 Localizing the Splash Screen Behavior. The participant steps through the code using the F8 keyboard shortcut or by pressing the “Step over” button. Stepping over means the program will advance the instruction pointer, but not step into the code referenced by `CALL` instructions. The participant encounters a `CALL` instruction at the beginning of the program and presses F7 to step into the program. The instruction execution jumps just a few instructions down, so the participant continues to step over the assembly instructions in the program.

Stepping over the instructions, the participant is reading and interpreting information from the code. To the right of the assembly instructions, the debugger presents information about system calls it has identified to the participant performing the task. The participant sees red text labeled `USER32.LoadBitmapA`, and `GDI32.CreatePatternBrush`, black text that says `ASCII: Splish_Class`, and a red label reading `USER32.CreateWindowExA`. The participant is pressing F8 and sees a call to another red-labeled USER32 function when the splash screen suddenly appears.

4.3.5 Inferring the Cause of the Splash Screen Behavior. The participant infers from the appearance of the splash screen that code has executed which controls the presentation of the splash screen window. The participant has stopped pressing F8 and looks to the instruction currently pointed to by the debugger. Looking around the area, the participant sees a `CALL` instruction labeled `CALL <JMP.&USER32.ShowWindow>` at address `0x00401534`. The splash screen is still being displayed with the instruction at that location. Thinking about it for a moment, the participant sets a breakpoint on the ShowWindow instruction, and presses rewind to restart the program.

When the participant restarts the program, the program execution lands on the ShowWindow instruction. The participant looks at surrounding instruction for a moment and presses F8 to move execution forward. As expected, the splash screen appears.

4.3.6 Patching a Jump. The participant presses F8 a few times to get below the ShowWindow instruction and thinks for a moment about what to do. The participant scrolls back up to an instruction labeled `PUSH 1` and double clicks on the instruction which opens up a dialog box.

The dialog box is labeled “Assemble” and `0x0040153E`, which is the address of the `PUSH 1` instruction. In the text field of the dialog box, the participant types in `JMP` and looks for an appropriate address to jump to. The participant finds an address, and reads off `0x00401588`, which the participant then types into the text field after JMP. The participant clicks the button labeled "Assemble" which modifies the bytes of the instruction and then "Close" to close the dialog box.

The participant restarts the program in the debugger using the "Rewind" button, presses "Run" and then again hits the breakpoint at the ShowWindow instruction. The participant sees the instruction labeled PUSH 1 and notices that the changes to the program were not persistent. After thinking for a moment, the participant clears the breakpoint from the ShowWindow function and sets a new breakpoint a few bytes earlier at an instruction labeled Call <JMP.&USER32.CreateWindowExA>.

The participant restarts the program and runs the program to the newly set breakpoint. The participant again sees the PUSH 1 instruction, double clicks the instruction to open the "Assemble" dialog, looks for an address to jump to and types JMP 401583 into the text field, which assembles the bytes EB 48 into that instruction's address, replacing the PUSH 1 instruction with a JMP instruction targeted to a few bytes below the ShowWindow system call. The participant closes the "Assemble" dialog and presses F8 through a number of instructions and can see that the splash screen does not appear. The participant presses F9 to execute the program, and the main window of the program appears without the splash screen appearing.

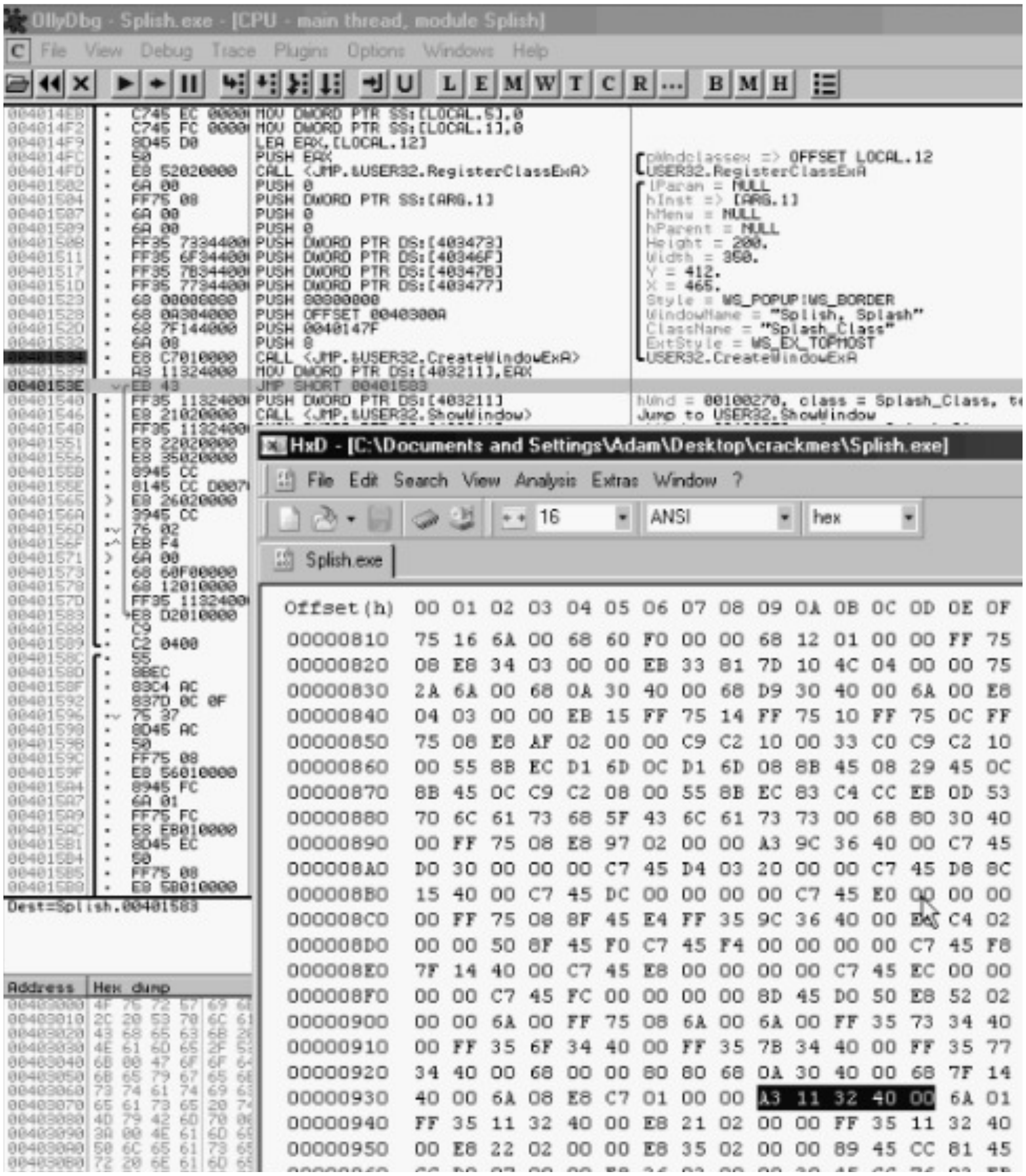
4.3.7 Saving the Changes to Disk. Now the participant knows that patching the jump successfully disables the splash screen, the participant starts to persist the changes into the executable by opening the program in a hex editor, finding the appropriate bytes, and changing them. The participant opens the HxD hexadecimal editor, and opens the Splish.exe file within the hex editor. In the hex editor, the participant types Ctrl+F to open a "Find" dialog. In the "Find" dialog, the participant sees a "Datatype:" label and an associated drop-down menu and selects "Hex-values" from the drop down. With the OllyDbg window in view, the participant looks back up to the instruction that was patched and reads off the byte string a311324000 of the bytes previous to the bytes which were modified, while typing the byte values in the text field labeled "Search for:" in the hex editor's "Find" dialog (Figure 13) The participant presses enter and the hex editor jumps to the a location in the program where the bytes were found.

The participant reads off the bytes, identifies the bytes `EB 43` which are to be modified, and attempts to type in the newly modified bytes.

4.3.8 Handling an Error. When the participant types in `EB 43`, the hex editor generates an error message box labeled "Error: Cannot open file C:\Documents and Settings\adam\Desktop\crackmes\Splish.exe for write access. The process cannot access the file because it is being used by another process." The participant reads the error message, tries to click the "Retry" button, and gets the same message. The participant clicks the "Cancel" button then thinks for a few moments to figure out what to do.

The participant tries to resolve this by closing the file in OllyDbg and switching to the hex editor. When the participant tries to insert the new bytes, the same message appears again from the hex editor. The participant thinks for a moment and then closes the file in the hex editor, and goes to open the hex editor again, but realizes that he does not remember the byte string to search for. The participant starts to open OllyDbg to find the bytes again, but then decides to check to see if the hex editor retained the information.

The participant opens the hex editor again, and presses Ctrl+F and sees the bytes `a311324000` in the text field. The participant then enters opens the text editor and types "search for: a31132400" so it will be available. The participant finds the bytes in the hex editor, but does not remember the bytes that need to be modified to. The participant opens OllyDbg, opens the Splish file, follows the same procedure as before to find the instruction that was patched above the ShowWindow instruction by pressing F8 and pressing F7 when landing on (`CALL` instructions that jump to another location within the program rather than to a system function). Upon finally arriving to the ShowWindow instruction, the participant sees that the breakpoint is



OllyDbg - Splish.exe - [CPU - main thread, module Splish]
C File View Debug Trace Plugins Options Windows Help
L E M W T C R ... B M H
HxD - [C:\Documents and Settings\Adam\Desktop\crackmes\Splish.exe]
File Edit Search View Analysis Extras Window ?
16
ANSI
hex
Splish.exe

```
004014EB  .  C745 EC 0000  MOV DWORD PTR SS:[LOCAL.5],0
004014F2  .  C745 FC 0000  MOV DWORD PTR SS:[LOCAL.1],0
004014F9  .  8D45 D0       LEA EAX,[LOCAL.12]
004014FC  .  50            PUSH EAX                                   [pWndclassex => OFFSET LOCAL.12
004014FD  .  E8 52020000   CALL <JMP.&USER32.RegisterClassExA>        [USER32.RegisterClassExA
00401502  .  6A 00         PUSH 0                                     [lParam = NULL
00401504  .  FF75 08       PUSH DWORD PTR SS:[ARG.1]                  |hInst => [ARG.1]
00401507  .  6A 00         PUSH 0                                     |hMenu = NULL
00401509  .  6A 00         PUSH 0                                     |hParent = NULL
0040150B  .  FF35 7334400  PUSH DWORD PTR DS:[403473]                 |Height = 200.
00401511  .  FF35 6F34400  PUSH DWORD PTR DS:[40346F]                 |Width = 350.
00401517  .  FF35 7B34400  PUSH DWORD PTR DS:[40347B]                 |Y = 412.
0040151D  .  FF35 7734400  PUSH DWORD PTR DS:[403477]                 |X = 465.
00401523  .  68 00000000   PUSH 80000000                              |Style = WS_POPUP|WS_BORDER
00401528  .  68 0A304000   PUSH OFFSET 0040300A                       |WindowName = "Splish, Splash"
0040152D  .  68 7F144000   PUSH 0040147F                              |ClassName = "Splash_Class"
00401532  .  6A 08         PUSH 8                                     |ExtStyle = WS_EX_TOPMOST
00401534  .  E8 C7010000   CALL <JMP.&USER32.CreateWindowExA>         [USER32.CreateWindowExA
00401539  .  A3 11324000   MOV DWORD PTR DS:[403211],EAX
0040153E   v-EB 43         JMP SHORT 00401583
00401540  .  FF35 1132400  PUSH DWORD PTR DS:[403211]                 hWnd = 00100270, class = Splash_Class, te
00401546  .  E8 21020000   CALL <JMP.&USER32.ShowWindow>              Jump to USER32.ShowWindow
0040154B  .  FF35 1132400
00401551  .  E8 22020000
00401556  .  E8 35020000
0040155B  .  8945 CC
0040155E  .  8145 CC D007
00401565  >  E8 26020000
0040156A  .  3945 CC
0040156D  .v 76 02
0040156F  .^ EB F4
00401571  >  6A 00
00401573  .  68 60F00000
00401578  .  68 12010000
0040157D  .  FF35 1132400
00401583  . >E8 D2010000
00401588  .  C9
00401589  .  C2 0400
0040158C  .  55
0040158D  .  8BEC
0040158F  .  83C4 AC
00401592  .  837D 0C 0F
00401596  .v 75 37
00401598  .  8D45 AC
0040159B  .  50
0040159C  .  FF75 08
0040159F  .  E8 56010000
004015A4  .  8945 FC
004015A7  .  6A 01
004015A9  .  FF75 FC
004015AC  .  E8 EB010000
004015B1  .  8D45 EC
004015B4  .  50
004015B5  .  FF75 08
004015B8  .  E8 5B010000
Dest=Splish.00401583
```

```
Address   Hex dump
00403000  4F 75 72 57 69 6E
00403010  2C 20 53 70 6C 61
00403020  43 68 65 63 6B 20
00403030  4E 61 6D 65 2F 53
00403040  6B 00 47 6F 6F 64
00403050  6B 65 79 67 65 6E
00403060  73 74 61 74 69 63
00403070  65 61 73 65 20 74
00403080  4D 79 42 6D 70 00
00403090  3A 00 4E 61 6D 65
004030A0  50 6C 65 61 73 65
004030B0  72 20 6E 61 6D 65
```

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000810  75 16 6A 00 68 60 F0 00 00 68 12 01 00 00 FF 75
00000820  08 E8 34 03 00 00 EB 33 81 7D 10 4C 04 00 00 75
00000830  2A 6A 00 68 0A 30 40 00 68 D9 30 40 00 6A 00 E8
00000840  04 03 00 00 EB 15 FF 75 14 FF 75 10 FF 75 0C FF
00000850  75 08 E8 AF 02 00 00 C9 C2 10 00 33 C0 C9 C2 10
00000860  00 55 8B EC D1 6D 0C D1 6D 08 8B 45 08 29 45 0C
00000870  8B 45 0C C9 C2 08 00 55 8B EC 83 C4 CC EB 0D 53
00000880  70 6C 61 73 68 5F 43 6C 61 73 73 00 68 80 30 40
00000890  00 FF 75 08 E8 97 02 00 00 A3 9C 36 40 00 C7 45
000008A0  D0 30 00 00 00 C7 45 D4 03 20 00 00 C7 45 D8 8C
000008B0  15 40 00 C7 45 DC 00 00 00 00 C7 45 E0 00 00 00
000008C0  00 FF 75 08 8F 45 E4 FF 35 9C 36 40 00 E8 C4 02
000008D0  00 00 50 8F 45 F0 C7 45 F4 00 00 00 00 C7 45 F8
000008E0  7F 14 40 00 C7 45 E8 00 00 00 00 C7 45 EC 00 00
000008F0  00 00 C7 45 FC 00 00 00 00 8D 45 D0 50 E8 52 02
00000900  00 00 6A 00 FF 75 08 6A 00 6A 00 FF 35 73 34 40
00000910  00 FF 35 6F 34 40 00 FF 35 7B 34 40 00 FF 35 77
00000920  34 40 00 68 00 00 80 80 68 0A 30 40 00 68 7F 14
00000930  40 00 6A 08 E8 C7 01 00 00 A3 11 32 40 00 6A 01
00000940  FF 35 11 32 40 00 E8 21 02 00 00 FF 35 11 32 40
00000950  00 E8 22 02 00 00 E8 35 02 00 00 89 45 CC 81 45
```

Figure 13 Finding the Bytes to Modify.

still set, which means the participant could have just pressed F9 and got to the desired instruction directly rather than having to navigate to the instruction.

The participant patches the jump over the `PUSH 1` instruction as before and looks at the bytes that are generated by the assemble command. The participant switches to the text editor and types "change to EB 43," then closes the program in OllyDbg, opens the hex editor, opens the Splish.exe program within the hex editor, finds the location where the bytes are to be modified (as before), then types `EB 43` over the bytes that read `6A 01`. The participant saves the program by selecting "Save as:" and entering a new name for the patched program: "splish_nosplash.exe."

4.3.9 Testing the Changes. The participant closes the hex editor, then finds the splish_nosplash.exe executable in a folder on the Desktop. The participant double clicks the program, notices that the main window appears without the splash screen first appearing. This tells the participant the first goal of the reverse engineering task has been completed. The participant took 13 minutes and 4 seconds to perform this portion of the task.

4.3.10 Summary of the Task Walk-Through. The reverse engineering task used for this study was intentionally very easy, but the task still involved exploring the program, determining the goal, determining an approach to accomplish the goal, localizing an observed behavior, inferring the cause of a behavior, modifying the program, saving changes to disk, handling errors, and testing the changes. Each of these different steps involves a number of different planning, reasoning, problem-solving, and information-seeking processes, which will be described in more detail throughout the dissertation.

The next section describes the data collection the think-aloud protocol which was used to systematically analyze the data from the participant's trial to answer how people make sense of programs. Next, the analysis of the information and affordances of the task environment are described. After that, the following section presents a framework to understand how elements in the reverse engineering situation interact with conceptual and procedural knowledge.

4.4 Think-Aloud Protocol

A think-aloud protocol [72] was used to collect and analyze concurrent verbalizations during the task. The researcher transcribed the verbal data into a text document and simultaneously segmented the data by breaking it into one segment per line of text. Segmenting the data at the same time as transcription allowed the use of contextual cues such as tone of voice, time between utterances, and current actions observed in the video data to aid in discriminating segments. Verbalizations were segmented to represent individual thoughts as is described in Trickett and Trafton [194] and Chi, et al. [41] Thoughts were demarcated by the presence of natural time breaks between utterances and in locations in which the focus of the verbalization switched to represent a different idea.

After segmentation, the researcher coded the verbal segments. The categories used to code the data were developed previously from a review of the literature on sensemaking (Section 2.3). As in Table 1, segments were classified as to whether they represented:

- Goals or plans ("G"),
- Hypotheses ("H"),
- Information seeking behaviors ("I"),
- Attentional focus ("F"), or
- Spurious verbalizations ("D" for "delete").

Each segment must contain a code, but a segment could be coded with more than one code if multiple rules applied to a single segment.

4.4.1 Categorization. Once all the verbal data from the task was coded, segments representing each of the different categories of verbalizations (goals/plans,

Table 1 Coding Rules.

Code	**Rule**
Goal / Plan	Reference to a goal or plan that indicates past, present, or future action taken by the participant: “I’m going to jump into this value,” “I can change this here”, “I put the 7 in”
Hypothesis	Reference to a belief, which could be a guess, reasoned inference, or a summarized or generalized observation: “This isn’t going to work”, “The next value is going to be 4”, or “All the values are probably empty”
Information-Seeking	Reference where the participant is searching for information in the task environment: “Where does this value go?”, “What does this jump do?”, “What happens after the call returns?”
Attentional Focus	Reference to some aspect of the situation, the person’s mental state, problem solving state, or state of the program / task environment: “The value is 3”, “I’m lost”, “EAX has a 4”
Del	Non-related verbalization, for instance “umm” or “okay” by themselves, or “. . . ” which indicates time passing
Notes	Enter concepts you are not sure about and comments.

state-information, hypotheses, and information-seeking behaviors) were reviewed and elaborated to detect emergent categories (more abstract classifications) with which to characterize the coded segments (as described in Ryan and Bernard [168]).

For the verbal segments included in the goals/plans category, labels were assigned to the verbalization which represented the function that the goal or plan appeared to serve in the task (such as "plan-approach," "construct-goal," "olly-close-file," or "set-breakpoint"). For the verbal segments in the attentional focus category, the researcher provided labels that classified the type of information being monitored in performance of the task (for instance "text," "causal behavior," "recognized-structure," and "surprise"). For segments in the hypothesis category, segments were given labels that represented the types of hypothesis stated (for instance, "program property," "behavior," "situation," and "identification of object"). For segments in the information-seeking behavior category, the segments were given labels describing the type of information that was sought (for instance "recognized object," "behavior," "location," "text," and "effect of action"). These labels represent organizational categories which emerged from several reviews of the data to organize the data into groups.

The numbers of segments from the different categories is presented in Table 2. Individual segments were allowed to have multiple codes, so the total segment count does not represent the sum of the segment category counts. Of the segments that provided meaningful information, attentional focus segments comprised the largest number of verbal segments (272 segments), followed by goal and plan-related segments (138 segments), hypothesis-related segments (121 segments), and information-seeking segments (50 segments).

4.4.2 Goals and Plans. A number of goals and plans were referred to in the task (Table 3) The majority of the goals (47.1 percent) described goals that were aimed at planning an actionable approach to achieve the current goal. Some examples of these segments include the phrases "I could just patch a jump right here," "I could

Table 2 Verbal Segment Properties.

Segment Category	**Number of Segments**
Goal / plan segments	138
Hypothesis segments	121
Information seeking segments	50
Attentional focus segments	272
Deleted segments	123
Total number of segments	680

just *nop* the whole thing," "I'll set a breakpoint," and "let's go to the next one." The next most numerous category of goal-related segments represented the construction of a goal representation (22 segments). In these segments, verbalizations such as "like E4 or something," "I want to change the PUSH 1 to jump," "I need to let it call this" represented the goal verbally by elaborating on the features or attributes of the goal state.

Operating system-related goals and plans were those involving the common functions of the operating system's user interface, such as "open it up" (referring to a file), "copy this out," and "close the debugger." Debugger-related goals and plans referred to as "let's close this and back up to the very start," "F9 to get there again," "step over," and "start stepping in." These segments all referred to actions in the debugger. Hex editor-related goals and plans involved elements in the hex editor in a similar manner. The remaining three segments represented goals or plans of action, but they did not fit into neat categories.

4.4.3 Hypotheses. There were also many hypotheses in the verbal data (Table 4). Hypotheses were classified as relating to the properties of objects, properties of behaviors of the program, the identification of an object, a simulated property of an object, a property of the situation, a property of the code and others. Hypotheses dealing with object properties involved the elaboration of one of the attributes of some element in the situation that was treated as an object. Examples of an object

Table 3 Goal- and Plan-Related Segments.

Segment Category	**Number of Segments**
Plan an approach	65 (47.1%)
Construct a goal representation	22 (15.9%)
Operating system-related	21 (15.2%)
Debugger-related	17 (12.3%)
Hex editor-related	9 (6.5%)
Finding a memory address	2 (1.4%)
Detecting a problem	1 (0.7%)
Remember a forgotten goal	1 (0.7%)
Total goal segments	138

property hypothesis are: "should be 74 or 75," "not any check on this time like in the other one," "there's a data section there," and "this is what's done to the name."

Hypotheses dealing with properties of behavior treated the behavior of the code as its own entity, and expressed properties in verbalizations such as "this is processing the initial thing," "`EDX` is going to get `ECX`," "So that will be mod," "it's going to paint the bitmap," and "it's going to tick once." Hypotheses identifying objects involved the participant recognizing the presence of something in the task environment as some mentally-held concept. Hypotheses also talked directly about the situation, in which verbalizations referenced the current state of elements in the task environment: "it's still open in memory," "oh, I had the breakpoints set still," and "must not have saved." Hypotheses related to the code's structure involved statements about the spatial layout of the code or recognized constructs in the code. Hypotheses dealing with properties of the task environment involved expectations about how the task environment worked or what affordances it had available. The other hypotheses in the table only contained a few instances, but did not fit well into any other categories.

4.4.4 Information Seeking. Segments relating to information-seeking behaviors involved a number of different types of information. The two most prevalent types of information sought were text and data values or top-down structure, which makes sense as OllyDbg presents primarily text information to the user and the user

Table 4 Hypothesis-Related Segments.

Segment Category	Number of Segments
Object property	50 (41.3%)
Property of behavior	18 (14.9%)
Identification of object	18 (14.9%)
Property of the situation	11 (9.1%)
Property of the code's structure	9 (6.6%)
Property of the task environment	4 (3.3%)
Relevance of information	3 (2.5%)
Feasibility of an approach	2 (1.7%)
Truth of a statement	2 (1.7%)
Evaluation of progress in the task	2 (1.7%)
Relation between objects	2 (1.7%)

Table 5 Segments Expressing Information Seeking.

Segment Category	Number of Segments.
Text or data value	14 (29.8%)
Top-down structure	14 (29.8%)
Mechanism of a program behavior	9 (19.1%)
Affordance in the task environment	5 (10.6%)
Behavior property	5 (10.6%)
Predicate on a value property	3 (6.4%)

must recognize program constructs from the text values. Segments describes as representing mechanisms of a behavior involved how something in the program worked, such as "it pushes the base pointer on the stack," "to get out of the message is 401," and "does it hit this one at all." Segments representing affordances in the task environment were concerned with whether the task environment had a particular function or information display available. Segments grouped as expressing predicates on behavior and value properties involved true or false statements about a behavior of the program or about a recognized construct treated like an object.

4.4.5 Attentional Focus. A number of verbalizations were categorized as representing the current focus of attention in the task (Table 6). Attention segments referred to anchors that the participant attended to in performance of the task. Again, the majority of these represent text or data values being read from the task environ-

Table 6 Types of Information Capturing Attentional Focus.

Segment Category	**Number of Segments**
Text and data values	133 (48.9%)
Reasoning or inference	66 (24.3%)
Recognized structure	30 (11.0%)
Observed program events	16 (5.9%)
Evaluation of task progress	14 (5.1%)
Unexpected event (surprise)	7 (2.6%)
Expected information	6 (2.2%)

ment: "name and hardcoded serial," "jump short jump 43," "translate message," and "sorry please try again." The second most numerous category represents mental inferences or current explanations in the task: "it didn't even stop on that breakpoint," "now it's doing a nothing message," and "that closes it." The third most common category represents identified structures which are recognized from the data: "that's a call," "that's the start of the program," "that's cancel." The "observed program events" category represents segments that indicated attention focusing on events that the participant observes the program performing. The category "evaluation of task progress" represented verbal segments where the participant discussed how the task was proceeding: "alright, I found the serial first" and "where was I."

4.4.6 Concepts and Emergent Conceptual Themes. Aside from goals and plans, hypotheses, information, and the attentional focus, the conceptual components from each verbalization were also analyzed and categorized. A category was added to each of the rows of the spreadsheet containing the coded and elaborated verbal segments. If a segment contained a reference to a concept, the concept was added in an additional column labeled "CONCEPT." The concept references consisted of any noun or verb (aside from a particular number or data value) that were uttered during the problem-solving task (for instance "doing," "check," "call," "step in," and "push").

After the concept references were elicited, a second method of gathering concept information was undertaken to improve the concepts elicited. All of the transcribed verbalizations except for “DEL” segments were put into a text file, in which numbers, stop words (such as “is,” “to,” “for”) and ancillary phrases (“I think,” “maybe”) were removed. The final list was then compared with the original list of concept references to ensure that the original list did not miss essential concepts. The original list was kept as the concept list from which other analysis would be made.

Once all of the concept references from the verbal data were recorded, each concept was elaborated to determine the emergent categories of concepts that it belonged to (for instance “push is-a instruction,” “push is-a program action,” “call is-a programming concept,” and “remember is-a cognitive action.”). After that, concept references were grouped according their categories (for instance “location,” “debugger action,” and “situation reference,”). Where only one or two concept references existed in a category, they were added to the closest category in which they fit best. Finally, the concept types were organized into a natural grouping based on concerns. This grouping separated the concept types as to whether they represent:

- System concepts,
- Task environment concepts,
- Situational concepts,
- Cognitive concepts, or
- Background knowledge concepts.

System concepts (Table 7) relate to references to objects, data, or other elements belonging to the underlying system being used and examined. This involves statements about the program and its behavior, statement about the operating system's interaction with the program, statement about how things work in the program, and so on.

Task environment concepts (Table 8) relate to those which are involved with the user's interaction with the task environment. These concepts involve static properties

Table 7 System Concept Types.

Concept Type	**Number**
Program behaviors	55
Debugger actions	31
System functions	17
Program concepts	12
Properties of structures treated like "objects" in the code	11
Mechanisms in the program	3
Processor object	1

Table 8 Task Environment Concept Types.

Concept Type	**Number**
Data properties	58
Object references	57
Individual instructions	38
Locations	35
User interaction concepts	25
Value references	21
Task environment actions	14
Text labels	8
Reverse engineering tool concepts	4
Affordances in the task environment	4

of the elements in the environment such as data values, structures in the code which have been encountered, and locations of the program.

Situational concepts (Table 9) involve elements that are from both the task environment and a person's performance in the task at the same time. These concepts include spatial and temporal properties of objects, the recognition of an object from memory, reference to an action in the task environment, the status of an action, or the judgment of whether something is relevant.

Cognitive concept types (Table 10) involve those in which the main objects are mental objects. Cognitive concepts involve properties of the actor's cognitive state, strategy, evaluation of the current problem state, and goal references. These are self-referential in nature and refer to the participant's underlying processes of thinking. Verbalized cognitive concepts are not nearly as numerous as the more visible concept

Table 9 Situational Concept Types.

Concept Type	**Number**
Reference to individual action	111
Spatial property of an object	23
Temporal property of an object	23
Direct reference to an aspect of the situation	22
Recognition of an object	11
Status of an action	8
Reference to the task	5
Judgment of whether something is relevant	3

Table 10 Cognitive Concept Types.

Concept Type	**Number**
Goal reference	14
Cognitive action	11
Problem state evaluation	9
Actor cognitive property	9
Mental simulation evaluation	6
Strategy	4
Problem solving action	2

groups, possibly because of the guidance to not attempt to explain thoughts when making verbalizations. This group of concepts would be the most helpful in developing models of procedural knowledge.

The final concept group, background knowledge concepts (Table 11) relates to those which are cognitive in nature, but which refer to stored knowledge or "background knowledge" that is used in the task. This group consists of background knowledge about debugging, programming constructs and concepts, mathematical and logic operations, and aspects of troubleshooting through experimentation. Concepts in these groups would be helpful in developing models of declarative knowledge in a task. They would also be helpful in developing procedural knowledge models in that many of these concepts involve patterns which could be expressed with a "condition, action" rule.

Table 11 Background Knowledge Concept Types.

Concept Type	**Number**
Debugging concept	110
Programming construct	69
Math concept	66
Comparison concept	27
Experimentation concept	22
Logic concept	12

4.4.7 Concept Summary. All of these concepts were grouped together to determine the elements that are related to reverse engineers' conceptual knowledge. From this, five primary areas emerged which can be collapsed further into three:

- The task environment (the system and the tools to manipulate it),
- The task situation, and
- The agent (the person and the person's knowledge).

The task environment, and the agent have properties which can be represented in a particular situation context. The next section discusses the role that each of these areas play in making sense of an executable program.

4.5 Discussion

The previous discussions presented information and affordances in a common reverse engineering task environment, a walk-through of a simple reverse engineering task, and an analysis of the procedural and conceptual aspects of knowledge from a think-aloud study with a single participant. Each of these areas of discussion provide a viewpoint from which to examine the reverse engineering process. These areas relate to each other constantly throughout the performance of a reverse engineering task. For an example of how the task environment (the tools and the program), the task situation, and the agent's knowledge (background knowledge and contents of short-term memory) interact, the process of finding the location of the ShowWindow instruction is considered.

4.5.1 Finding the ShowWindow Call. When the participant is looking for the instruction that calls the ShowWindow function in the task environment, there are a number of different ways a person can perform this subtask. The strategies a person can use to find a location in memory depend on the amount of "background knowledge" the person has. In any task, a person may have all, some, or none of the knowledge required to solve the problem. In addition, the task environment may provide affordances which make the task easy, but the person has to have knowledge that the affordance exists, and that the affordance should be used in this particular situation. Several features provided by the OllyDbg environment allow a person to find a function call in the assembly code. Some of these features include a list of imported functions and highlighted (red) text to the right of the disassembly pane displaying system call information.

In the reverse engineering session from the Splish task, the call to the `ShowWindow` function was mapped into the program's memory at address `0x00401546`. Someone could find the function based on knowledge of:

1. The name of the system call,
2. The address of the `ShowWindow` function in memory, and
3. The address where `ShowWindow` is called by the program.

Combinations of these three pieces of knowledge would enable someone to find the function by a number of different strategies.

4.5.1.1 Knowledge of the System Call Address. It is not likely that someone without experience in the particular program would know the exact memory address in the program where the call to `ShowWindow` is found. However, a person might have access to a list of where the different operating system function calls are mapped into the program's address space. However, if someone does have explicit knowledge of the address where the `ShowWindow` function from the USER32.DLL file is mapped into memory, the person can use the search features of the debugger or disassembler tool to search for a `CALL` instruction in the program which has this address as an operand.

4.5.1.2 Knowledge of the System Call. If that address is not known, but the person knows that the `ShowWindow` function is what causes the splash screen to appear, the person can look for calls to this function. Since the person does not know the address of ShowWindow in the DLL or the address where it is called to create the splash screen, the reference to the call must be found in another way. The participant could search for a reference to where system calls are mapped into memory, or the person could use the reverse engineering tool to look for a list of calls made by the program, find the appropriate call to the `ShowWindow` function in the list, and then determine if the reference to that function is the reference which creates the splash screen.

4.5.1.3 Knowledge-Free or Learning-Based Strategies. If the reverse engineer does not know that `ShowWindow` is the function of interest, the person can still find the correct function by exploiting knowledge that some function related to "windowing" is involved to learn what the correct system function is and where the call to that function is located. In this case, the person can look through the system calls imported by the program, look for functions that seem like they might involve windowing such as calls which contain the words "window," "screen," "dialog," "image", "bitmap," and so on. The participant then can investigate the references to each, until the target function call is found. This strategy requires the person to sift through much more data than a knowledge-enabled strategy would require, but it enables the person to interactively acquire background knowledge which may be helpful in other reverse engineering tasks.

A less-experienced reverse engineer might try to look for the `ShowWindow` function but may not have the knowledge that an operating system call is what produces the splash screen window. In this case, it is still possible for the person to find the function, but the approach to doing so is completely learning-based. The novice reverse engineer with no knowledge of system calls can use the debugger to step into each instruction until the instruction is executed which causes the splash screen to appear. The participant can then infer that the instruction that was executed when the splash screen appeared caused the program to execute the splash screen. If the instruction that executed is a `CALL` instruction, the person can infer that one of the instructions in the function that was called is responsible for showing the splash screen.

Even this strategy requires background knowledge about how programs are mapped into memory. When the program makes a call to code from a dynamic link library (DLL) file that implements part of the operating system's API, the program steps through instructions that are not technically part of the program. It requires background knowledge to detect if the debugger is stepping through program code or through code from an operating system DLL. To detect whether the debugger has stepped into DLL code, a person has to recognize information cues from the environment, which in this case are provided by addresses in the disassembly pane of the debugger's window which indicate a addresses starting with `0x7XXXXXXX` instead of `0x4XXXXXXX`. There is also a text label in the debugger's register pane which can provide the reverse engineer an indication the debugger is executing code in USER32.DLL, NT.DLL, or another DLL.

The brute force process of stepping through each instruction can be very tedious and can take a long time. Another strategy to find the function call is to step over instructions until the splash screen appears and then recursively narrow down into the functions until the correct `CALL` instruction is found. This strategy involves stepping through the instructions in the program until the program displays the behavior of interest. When the program behavior of interest is displayed, the person can set a break point at one of the instructions directly before the instruction that generated the behavior. When the program is run again, it will stop before the input is read, which allows the user to step into the function where the user's input is taken.

In this way, a person can use knowledge about troubleshooting to acquire knowledge about the particular system call of interest. When the person finally arrives at the system call, he or she will be able to step over that instruction and see the splash screen appear as the system call is executed. This enables the person to mentally connect the event of the splash screen and the system call code as its most plausible cause.

4.6 Conclusions

Reverse engineering tasks require a great deal of domain knowledge, but they also involve the ability to leverage that knowledge to find elements of interest, to make predictions about the program, to abstract instruction segments of the program, and to develop higher-level explanations about the program's properties. All of these activities rely on the ability to make sense of elements in the environment to construct a coherent model of the situation.

This chapter presented information and affordances involved with a reverse engineering task, a walk-through of a reverse engineering task, and analysis of a think-aloud protocol to examine procedural and conceptual knowledge involved in a reverse engineering situation. After that, an example of finding the call to an operating system function was described in terms of the knowledge required and the affordances provided to the person performing the task.

The next chapter presents a semi-structured interview study which elicits elements of procedural and conceptual knowledge in reverse engineering from subject matter experts. In Chapter 6, an observational study is presented to develop a theory of the sensemaking process in understanding executable programs. Finally in Chapter 7, the overall findings of the research are discussed and directions for future research work are presented

5. Semi-Structured Interviews with Subject Matter Experts

5.1 Introduction

The previous chapter discussed a case study of a reverse engineering task and outlined a conceptual framework for understanding situational aspects of how reverse engineers make sense of executable programs. This chapter describes a study undertaken to connect the low-level details involved with cognitive aspects of reverse engineering with concepts and processes that reverse engineers refer to when talking about reverse engineering work. To achieve this goal and to gain a broader picture of cognitive aspects of making sense of executable programs in real-world reverse engineering work, subject matter expert (SME) reverse engineers were interviewed, and the data from the interviews was captured and analyzed. From the analysis of interview data the procedural aspects (goals and decisions) and conceptual aspects (information cues, concepts, and knowledge) of reverse engineering work were elicited and represented.

First, the analysis of the interview data is presented. After that, the different domains in reverse engineering are described, followed by an organization of goals used in reverse engineering tasks. After that, conceptual aspects of reverse engineering are presented including the different ways information cues are used, specialized knowledge requirements, and the role of tacit knowledge in solving reverse engineering problems.

5.2 Segmentation and Coding of Interview Data

After the interviews were transcribed, they were analyzed for conceptual content, organization of concepts, and to answer the research questions of the dissertation.

The researcher read through the printed transcriptions several times and took notes to record how each of the SMEs responded to the interview questions, to notice patterns and themes, and to relate themes across the different interviews and questions. The themes from the post-interview notes were compared with the notes taken during the interviews to ensure no themes or important concepts were missed from impressions captured during the interview.

The text documents containing the interview transcripts were segmented and coded in two separate ways for analysis. First, the text was segmented by sentences and analyzed to elicit concepts from the interviews. Second, a copy of the interview data were segmented by ideas, which spanned from a portion of one sentence to several sentences in length, and coded according to which question group they addressed.

5.2.1 Sentence-Level Concept Analysis. To analyze the document at the sentence level, the transcriptions were divided up into individual sentence-sized segments. The rule that guided segmentation was: "segments should be sentences, and are described by punctuation in the text file (like periods or question marks) that normally indicate the end of a sentence." To segment discourse that was difficult to segment using that rule, the rule "a segment should represent a single idea" was used as a secondary way to demarcate segments from the text. Afterward, all of the segments were reviewed by the researcher to ensure they represented legitimate sentences and phrases.

Once the transcriptions were segmented, each of the segments was coded by the researcher to annotate conceptual content. Each segment within each of the documents was annotated with one or more tags to represent the concepts discussed within that segment. For instance, a single line of text would read something like: "PROTECTIONS, BREAK, APPROACH, INTUITION, (the sentence text)" if the concepts "protection," "break," "approach," and "intuition" were talked about in the sentence. During coding, the segments were kept in their original ordering to ensure referring expressions within the segments maintained their original contexts in the SME's responses to interview questions.

Once all of the sentences were coded, the researcher wrote an automated script to extract the concepts from the text documents, to count their frequencies of occurrence, and to determine the co-occurrences in which one concept appeared with another concept throughout all of the text documents. The script is included as Attachment F.

5.2.2 Idea-Level Concept Analysis. After the sentence-based segmentation, copies of the original, un-coded documents were reviewed again and the text was segmented based on the following rule alone: "segments should represent a single idea." This rule divided the segments much differently, with smaller ideas taking only a part of a sentence and larger or more complex ideas taking sometimes several sentences to express. In cases where the SME participant used storytelling to elaborate on an idea, the segments relating to the annotation of the idea were longer.

Once the text was broken up into idea-sized segments, the segments were coded based on their relation to one of the question groups from the questionnaire. The following codes represented the different groups of questions in the interview questionnaire:

- APPROACH - Statements related to the approach taken to solve a reverse engineering task.
- CUES - Statements related to using information cues in the course of a task.
- DECISIONS - Statements related to decisions in a task and how they are made.
- DOMAIN - Statements referring to the organization of the reverse engineering problem domain.
- GOALS - Statements relating to the underlying goals used in performing a task.
- KNOWLEDGE - Statements related to concepts a reverse engineer needs to know.
- SKILLS - Statements related to abilities or procedures a reverse engineer needs to have.
- TACIT - Statements referring to knowledge that has become proceduralized or automatized with experience.

- TOOLS - Statements relating to reverse engineering tools.

During coding, the answers were abstracted by reviewing each of the original documents several times and annotating a more abstract and concise description of the SME's response. Each annotation had a marking to identify the SME and a code which described the category to which the response applied. For instance, a long sentence describing the goal of knowing the purpose for performing reverse engineering would be coded: "SME1, GOAL, Find the purpose." Another sentence describing a necessary piece of knowledge for a reverse engineer might be coded: "SME4, KNOWLEDGE, Manual function name resolution." The annotations were to be self-contained so that they would not rely on a reading of the text or referring expressions within the text, but instead represent abstracted and captured answers to the research questions.

Once the annotations were created and coded, they were combined into a single file for analysis. This file provides a concise list of the interview questions and the SMEs' answers to the questions. Each of the SMEs were later asked to verify the organization of the goals, knowledge requirements, and tool needs and to provide comments on the structure of the overall responses. The abstracted interview responses are found in Appendix G. The representation of a goal-directed task analysis [68] of reverse engineering as constructed from the interviews is found in Appendix H.

5.3 Recommendations for Observational Study

After the interviews were conducted, the SMEs were asked for their advice on designing an observational study that would capture the essential elements of building a mental model of an executable program. Common elements of their input were incorporated into the design of the observational study discussed in Chapter 6.

5.4 Results and Discussion

The rest of this chapter describes overall results of the study from the SMEs' answers to the interview questions. First, in Section 5.5, the domain of reverse engineering is described in the ways used by the SMEs to decompose the concerns of reverse engineering. Next, Section 5.6 provides an organization of the procedural knowledge aspects of performing reverse engineering tasks including the goals and approaches the SMEs described using when reverse engineering programs.

Following that, the SMEs' interpretation of conceptual knowledge aspects of reverse engineering are presented. This discussion includes the information reverse engineers use from the task environment and the areas of knowledge that are required to perform reverse engineering tasks (Section 5.7).

5.5 Reverse Engineering Domains

There are several different ways that the SMEs differentiated tasks in reverse engineering. Reverse engineers described reverse engineering tasks as involving the following arenas:

- Software,
- Hardware, and
- Firmware.

Within software reverse engineering, the participants discussed several different arenas of software where reverse engineering is performed:

- Web and network applications,
- Desktop applications,
- Documents containing software,
- Libraries and DLLs,
- Embedded systems, and
- System-level software.

The SMEs also differentiated reverse engineering according to the purposes for which reverse engineering is being conducted. This line of differentiation was the most prominent in the interviews, and breaks down into four major categories:

- Vulnerability discovery,
- Malicious software analysis (including looking for rootkits and backdoors),
- Software protection analysis, and
- Reverse engineering unprotected software.

Across all of the interviews, the salient feature that separated software reverse engineering from another activity was that software reverse engineering involves reading programs from assembly code rather than source code. For instance, the SMEs explicitly excluded network penetration testing (a related cyber security domain) and looking for vulnerabilities in source code from consideration as reverse engineering, because while they involved similar types of problem-solving, they do not involve reading assembly language code.

The interviews elicited knowledge that is specific to vulnerability discovery, malicious software analysis, and software protection analysis. The goals and knowledge that are specific to each of those domains are important to understanding how those tasks are performed, but they are not central to the goal of the dissertation, which is understanding the conceptual and procedural aspects of how people make sense of executable programs. Instead, this section focuses on the procedural and conceptual aspects that are shared between the four domains.

5.6 Procedural Aspects of Understanding Programs

In the interviews, the SMEs discussed their approaches to problem-solving in reverse engineering, their goals, and how their goals affected their understanding of the programs they reverse engineered. All of the reverse engineers described a number of particular problems they remembered from experiences they had reverse engineering

Table 12 Goals in Making Sense of Programs.

Goal
Understand the purpose of analysis
Finish the analysis quickly
Discover general properties of the program
Understand how the program uses the system interface
Understand, abstract, and label instruction-level information
Understand, abstract, and label the program's functions
Understand how the program uses data
Construct a complete "picture" of the program

programs. Many of the descriptions involved difficult challenges in breaking software protections, deobfuscating program code, or getting access to instructions of encrypted or packed programs to enable them to begin to understand the program.

The SMEs also discussed the approaches used to understand the program and to make sense of "what the program does" in the context of their goals. Some of the described activities were at a higher level of abstraction, such as "discover the properties of the program" and "understand the purpose of analysis." Others applied to all of the categories of software reverse engineering: "understand system calls" and "understand code inside the function." The goals that applied to understanding unprotected code also applied to all other categories, so this domain is used to understand how reverse engineers make sense of executable programs. The organization of the goals elicited from the interviews is shown in Table 12.

5.6.1 Understand the Purpose of Analysis. Since the properties that are important depend upon the purpose for which analysis is conducted, the SMEs expressed that determining that purpose is itself an important goal. The need for an explicit purpose is consistent with prior findings in which programmers maintaining source code in controlled studies had difficulty understanding programs when they were not given goals to constrain their analysis activities [146, 202] . One of the SMEs commented:

> “You need to find the purpose for reverse engineering the code. What is the specific question that the reverse engineering work will be answering? Start with a specific question. If you don’t start with a specific question your goals will be aimless. The question you have also drives other questions you have to answer as you go through the process.”

For example, if the output of the reverse engineering effort is a report describing the behaviors of a program, the goals of analysis are constrained to those which will help a reverse engineer gain information that relates to that goal. When goals are constrained in this way, the reverse engineer can focus efforts on those activities which will help provide information about the program’s behaviors rather than other less relevant information.

The SMEs indicated that they commonly ignore large parts of programs that are not directly related to their analysis objectives. In order to save time in the analysis task, the entire program cannot be investigated and analyzed, so they focus on those parts of the program which will provide them the most benefit. In this respect, the desired output of the task drives the goals of analysis, which in turn drives the overall direction in which analysis proceeds.

5.6.2 Finish the Analysis Quickly. Though it seems like more of a constraint than a goal, all of the SMEs explicitly described the constant need to complete the analysis tasks as quickly as possible. Reverse engineering a program is manpower intensive, so it can be an expensive way for an organization to find out information about a program. The SMEs expressed that since reverse engineering is so expensive, they have a strong motivation to stay focused on achieving the overall goal of finishing the reverse engineering task and to avoid distractions. In fact, finishing the task quickly was considered by the SMEs to be more important than understanding the program in extensive detail. They described making decisions about the value trade-off where more time spent in analysis may not provide better value to the sponsor that is paying for them to reverse engineer the target program.

The goal of finishing the task quickly leads to the selection of strategies which can accomplish the task quickly rather than those which are slower but provide richer information or better understanding of the program. Finishing quickly also means that in practice reverse engineers constantly try to find faster ways of performing effective analysis, breaking protections, automating repetitive tasks, and generating value for their customers.

5.6.3 Discover General Properties of the Program. The SMEs mentioned the main goal for each type of reverse engineering task was to discover as much as possible about the program. For a small program, this means identifying all of the program's behaviors for all possible inputs. However, for large programs, the state space of the programs grows exponentially with every decision procedure in the program's code. This mean that the goal is to understand the most important aspects of the program's behavior given the most relevant inputs to the program.

One of the ways to quickly gather information about a program is by looking at its general observable properties, such as its file size, the size of the sections of the program that are mapped into memory, the names of the sections, whether or not the file's header is well-formed, and any text strings in the program. This information provides "quick and dirty" approaches to quickly narrow down what needs to be investigated in the program.

5.6.4 Understand How the Program Uses the System Interface. Other properties of the program are more abstract, such as how a program uses the system's interface. In order for programs to perform any tasks on a system they typically have to make programmatic requests through the operating system's system call interface. Functionality extended through the system call interface includes I/O functionality like video buffer write operations or file system read and write operations.

The SMEs described looking at the library calls that a program imports and the system functions that it uses to form a mental model about what the program does, sometimes before ever stepping through the code or watching it execute. System calls can provide information that allow a person to explore the behaviors of a program, or they can be used to enable the person to generate a hypothetical explanation of what behaviors the program might perform, which the reverse engineer can then look for in the program's code.

The SMEs described getting information about how the program uses the system interface by examining the import tables, scrolling through the program looking for system calls that the debugger or disassembler identifies, and by hooking the system APIs and letting the program run in order to discover sequences of system calls used by the program.

5.6.5 *Understand, Abstract, and Label Instruction-Level Information.*

The SMEs indicated that another important process in understanding programs is examining the instructions inside functions to be able to assign meaning to patterns of instruction sequences. Groups of assembly language instructions perform the algorithmic operations and data manipulation processes in a program. Sometimes these *local* code fragments perform computations to construct a memory address to which the program will transfer control. In other times they reconstruct variables that are used later in the program. Still in other cases local code may be sequences of instructions which represent algorithms that the reverse engineer is trying to uncover and understand.

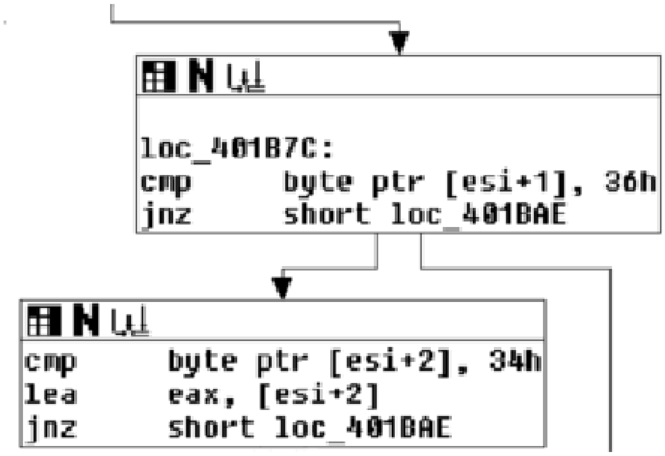
The SMEs described understanding sequences of assembly instructions by tracing data values as they moved through a program's execution, or through translating assembly instructions into a higher-level programming language syntax or into pseudocode, either mentally, on paper, or in a text editor. Analyzing instruction sequences can help reverse engineers better understand how the code inside a function works in order to help them better understand the function. Other times, different sequences of instructions within a single function have their own roles and the reverse engineer might consider them separately and label or comment on them independent of the function they are considered to be a part of.

Once the behavior of a sequence of instructions is specified and understood, the details of that sequence can be abstracted away and replaced with a meaningful symbol or label to represent the behavior of the instruction sequence in the person's memory. Symbols or labels for sequences of instructions make it so that sequence does not have to be processed each time it is encountered. Instead, a reverse engineer can group a sequence of 100 or so instructions with the label "decryption routine," and then that label represents the sequence almost as if it were an object with its own properties and behaviors.

5.6.6 Understand, Abstract, and Label the Program's Functions.

Another theme from the interviews was that reverse engineers analyze a program's functions and subroutines to determine the behaviors of the program. Many programs are written using functions and subroutines and much of the functional structure of a program is preserved when programs are compiled from source code into machine code. For instance, when a program module performs a `CALL` instruction to another area of the program, the program executes until it comes to a `RETN` instruction and return control flow to the originating program module, typically with a return value stored in the `EAX` register (depending on the calling convention).

Reverse engineers can form mental models of how program control flow works by dividing the instructions of a program into meaningful basic blocks by hand or using tools that display graphical representations of the code's control flow. To divide a program into meaningful basic blocks, a program can be grouped into sets of linear sequences of instructions which are demarcated by control flow instructions like `JMP`, `CALL` or `RETN` instructions. Dividing sets of assembly language instructions into subroutines is a natural process for reverse engineering programs since it abstracts away many of the details of assembly language. Many analysis tools such as IDA Pro



```
loc_401B7C:
cmp     byte ptr [esi+1], 36h
jnz     short loc_401BAE
```

```
cmp     byte ptr [esi+2], 34h
lea     eax, [esi+2]
jnz     short loc_401BAE
```

Figure 14 Basic Blocks in IDA Pro.

[88] perform this analysis automatically to present graph-based views of code to the person using the tool.

When a reverse engineer understands what a function does, it becomes meaningful to look at patterns of function calls in the program. Patterns in the ordering of function calls can make analysis tasks move from concerns about syntax issues to concerns over functional and behavioral-level aspects of programs. Additionally, reverse engineers can gain information about how functions interact with each other, such as in how functions pass arguments and return values back and forth.

In understanding the relationships in how functions call each other in a program, a reverse engineer can get a better understanding of the roles that the different modules of the program perform. SMEs indicated that it often requires using top-down knowledge about the problem domain (such as malicious software analysis or vulnerability discovery) in order to make sense of how functions work in the context of the domain.

5.6.7 Understand How the Program Uses Data. SMEs indicated that understanding how instructions interact with program data is also important. The relationships between instructions and the data contained in memory can help a reverse engineer understand the functionality a code segment provides and can provide insight about data structures that might be used in the program. For instance, if several instructions read and write memory to a small group of values in the program's heap, it could indicate that the functions belong to what in the source code was a dynamically allocated object instance of a C++ class.

A reverse engineer can understand local code by following the flow of data, registers, and memory values forward from a starting point or by tracing the flow back from an ending point. Tracing data forward and backward from points in the program's instruction sequence is a filtering process which helps isolate important instructions from less important ones.

Tracing the flow forward from a particular point in the code might be useful in order to discover how a value in memory changes or to determine which of the subsequent local instructions are relevant to that value. This can serve as a way to filter the instructions to only those which are relevant. Tracing the flow backwards from an end point allows seeing where a value came from and how it was constructed. Tracing backward allows determining which instructions are important to a register or memory address having the value that it does.

Reverse engineers can also use information about how a program uses data to determine how the program interacts with the outside world. Programs take input from the world in the form of data, which is processed by functions and instruction sequences in the program. This information can be used to determine the control flow of a program, for instance if malicious software "senses" whether it is being run in a virtual machine, or if it is a bot which looks for a certain type of input or set of commands before transferring control to parts of the program involved in carrying out its behaviors.

Also, if a reverse engineer is looking for exploitable vulnerabilities, understanding the location and safety of how the program handles data that comes from outside the program can help isolate bugs that can be manipulated by an adversary. Knowing that data is from outside of the program involves being able to trace the data from where it was generated to where it is used. Understanding whether or not the program handles the data safely involves understanding how the program uses data as well as how the program "should have" used the data.

5.6.8 Construct a Complete "Picture" of the Program. The SMEs discussed "building a complete picture of the program." This "picture" of the program is the "situation model" discussed in Chapter 2 and its contents and means of construction are the primary interest of the dissertation.

The SMEs discussed the complete picture of the program as understanding "what the program does," "how the program works," "what the parts of the program are," and "where" the different parts of the program are located in memory. From these descriptions and the other procedural aspects of understanging programs outlined above, the main properties of a situation model or "complete picture" of a program involve:

- Program components (functions, subroutines, or sequences of instructions),
- Program behaviors (things the program does), and
- Program functionality (the mechanism of how the parts work).

These categories are consistent with the organization of concept concerns from the case study (Chapter 4).

The SMEs described the activity of switching back and forth between top-down activities (like understanding functions) and bottom-up activities (like tracing data through the program) until they come to the complete picture of the program. The process of switching between top-down and bottom-up activities has also been documented in similar studies with programmers working with source code [146].

The complete picture of a program might also involve questions of intent, such as "why the program's developer would have written the program to perform the behaviors it exhibits in the way it exhibits them." The interpretation of program intent requires a person to be knowledgeable about abstract behaviors programs can exhibit, goals and incentives the developer might have, and scenarios where the developer's intentions could be achieved through writing the program. Understanding how people interpret the intent of programs (like malicious programs) from assembly language representations is an additional area for future research.

This chapter has so far discussed the different work domains involved in software reverse engineering and the procedural aspects of how reverse engineers make sense of executable programs from subject matter expert interviews. The next section discusses the conceptual elements involved in understanding programs.

5.7 Conceptual Aspects of Reverse Engineering

Other aspects of understanding executable programs include the information used in the course of reverse engineering, and background knowledge which helps enable the top-down recognition of patterns in the code. The SMEs related the same theme several times about their approach, decisions, goals, and activities: "it depends." Their decisions were informed by both the information from their reverse engineering tools, and background knowledge from their training, education, and experience. First, this section describes how reverse engineers use information cues in the process of reverse engineering. After that, it describes the knowledge areas which the SMEs indicated are important to being able to reverse engineer programs.

5.7.1 Information Cues. Reverse engineering tools provide information cues which can shape the course of a reverse engineering task. The SMEs reported several examples of where their decisions in the task depended on factors from information presented by their tools. For instance, in vulnerability discovery and unprotected program analysis tasks where applications are usually not protected, the reverse engineer is able to assume the program has been compiled with normal program compilation techniques. Since code structures will likely be normal and there will be no protections to deal with, the person does not have to look for abnormal code structures or latent functionality in the program. Additionally, the information provided to a reverse engineer by a disassembler, debugger or other instrumentation tools can be considered reliable when working on those types of applications.

With malware and software protection analysis tasks, reverse engineers cannot rely on any of these assumptions. The SMEs described that these operating assumptions are violated as a rule rather than as an exception. In malicious or protected code, the code structure will likely be abnormal. The program could contain packed or encrypted code or have code that is heavily obfuscated. The program might also rely on dynamically loaded libraries or function bindings which change throughout the course of the program's execution.

The SMEs described information that they came across passively, that they actively sought out, and that they had to continuously monitoring for. They also related being careful to verify the trustworthiness of the information they are presented in the context of the overall program.

5.7.1.1 Passively Discovered Information. Some of the information cues described by the SMEs were passive information cues. With these types of cues, the SMEs saw something that "looked weird" in the tools, or they came across an error message or some other information which led the task in a different direction. These cues can indicate areas that need to be looked into further and can verify or disconfirm a previously-held assumption the reverse engineer might have had.

An example given of a passively-discovered information cue is when a reverse engineer debugs a program and it terminates right after it is started in the debugger. In this situation, the information perceived about the program stopping suggests that the current operating assumptions (that the program did not have anti-debugging code) have changed. Once that shift takes place, the reverse engineer has to change his or her approach to defeat the anti-debugging code before being able to understand the program.

5.7.1.2 Actively Sought Information.

In some cases, like the terminating program mentioned above, the task environment readily presents the essential information cue to the reverse engineer. In other cases, the reverse engineer must actively seek out information. The SMEs indicated that the information that is sought is based on a background hypothesis or set of assumptions, whether the person realizes it or not.

One SME mentioned looking for indicators in a program, such as calls to library functions that were known to be unsafe, which could help indicate that a program might have vulnerabilities which can be exploited. If the code only uses safe string handling functions and proper code handling practices it might generally be safer code and less vulnerable to attack. However, if the program displays evidence of unsafe code practices, uses unsafe string handling functions, and does not check arguments in a subroutine, it indicates that inputs may also not be handled correctly and that exploitable vulnerabilities might be easier to find.

In this case, the task environment does not readily present this information to the person performing the task. Instead, the person has an assumption that leads him or her to look for the information, determine when it has been found, and interpret what it means in the context of the person's mental model of the program.

The SMEs indicated that expert reverse engineers develop their own instrumentation tools to determine information that is not readily accessible with their current tool setup. This kind of information might include information about whether malware writes instructions into memory or tries to communicate via a network port. A less experienced reverse engineer might not know to look for these cues, might not know how that information can be gathered, or might not have the background in programming and systems to be able to create instrumentation tools to gather the information. The SMEs referred to this as "knowing the right things to look for" and described it as one of the things that reverse engineers gain with experience over time.

5.7.1.3 Continuously Monitored Information. Finding the information may not be enough when a program's information can change between observations. In these cases, actively monitoring elements in the task environment is required. For example, an experienced reverse engineer can determine whether a malware program writes files to the file system by actively monitoring for whether file system changes are made, but if this is not monitored while the program executes, it will not be seen. Other types of monitoring activities described include monitoring system processes to see if new processes are created or monitoring to see if a program tries to execute code that is marked non-executable. In cases in which this type of monitoring cannot be performed visually, the SMEs described that they write tools to automate their monitoring and to notify them when the event of interest takes place.

5.7.1.4 Verifying the Trustworthiness of Information. Another complication with information cues is that reverse engineers have to be able to trust that the information provided by the tools is correct. One of the SMEs indicated that novice reverse engineers will tend to trust their tools more than experts will. All of the SMEs described setting up small experiments to test their assumptions so they can make more confident statements about what the program does.

Sometimes the information is inaccurate because tools do not portray accurate information or cannot understand the program being disassembled. The SMEs referred to not using the decompilers that convert assembly instructions back into source code, or at least verifying the information from the decompiler before trusting it. Decompilers have to make a number of assumptions which might not be the correct for that reverse engineering situation.

Other times information is inaccurate because malicious programs may be involved which change the information that is being reported or present a "garden path" for the reverse engineer. This is a problem because it can take the reverse engineer down a path of analysis which will not reveal the true behaviors of the program. In these cases, the reverse engineer thinks the analysis is going well when it is not. A

SME described it as: “being taken down a rabbit hole… You can potentially waste a lot of precious time with nothing to gain.”

5.7.1.5 Goal-Related Cues. Other information cues provide indications about the problem-solving state rather than the state of the program. These cues indicate that the goal has been achieved or is close to being achieved. The particular goal-related information cues depend on the purpose for which reverse engineering is to be performed and the output expected from the reverse engineering task.

If the purpose of reverse engineering is to understand the overall functionality of a program, a cue one of the SMEs described is that most of the functions in a program have been discovered and understood. If the purpose is to determine whether a program exhibits a specific type of behavior, a SME described using the cue that all of the system calls known to relate to that behavior have already been looked at. In that case, the reverse engineer only has to rule out the possibility that the program dynamically loads the functionality to perform that behavior later.

During vulnerability discovery, it is possible that the entire reverse engineering effort could be wasted if, in fact, the code does not have any vulnerabilities. When this happens, the time spent looking for vulnerabilities in one program could be better served looking for a different vulnerability in another program. In these cases, the SMEs referred to using their rate of progress and their progress given their effort as two indicators. A reverse engineer might determine analysis is going nowhere if there has not been progress for a long amount of time, or if a lot of effort has been expended where usually it only takes a small amount of effort.

This section has described how information cues are used in reverse engineering executable programs. The next section describes the conceptual aspects of knowledge that help reverse engineers connect the information cues in the environment with their performance in the task.

Table 13 General Knowledge Areas in Reverse Engineering.

Knowledge Area
Assembly language
Computer programming
Debugging and troubleshooting
Processor structure and function
Program execution process
Operating system calls
File header formats
Operating system internals
Anti-reverse engineering techniques
Firmware and hardware
Network communication protocols
Compilers and interpretation

5.7.2 Specialized Knowledge. Reverse engineers make sense of programs by connecting the information they encounter from the task environment with their background knowledge. One of the things many of the SMEs pressed upon was the vast amount of knowledge that is required to be good at reverse engineering programs. The SMEs reported that reverse engineers require knowledge from most of the areas involved with computer science. The primary knowledge areas identified from analysis of the interview responses are presented in Table 13.

Apart from the general knowledge involved in reverse engineering, the SMEs also indicated specialized knowledge which they believe separates experts from novices. These areas of domain-specific expertise are presented in Table 14 and the findings from the SME interviews related to these knowledge areas are discussed in the rest of this section.

5.7.2.1 Translating from Assembly Into Higher-Level Languages. The SMEs identified the knowledge of and facility with assembly language as one of the most important components of a reverse engineer's practical knowledge. Sequences of assembly language instructions comprise the major data representation that reverse engineers deal with. One of the most helpful capabilities is that expertise in assembly language allows reverse engineers the ability to see common patterns in program code and quickly translate these patterns to higher-level representations.

The SMEs reported having a fluidity with this process which they gained through experience reverse engineering code. Experts have a built-up mental repository of patterns or “plans” which them connect a sequence of assembly language instructions in the task environment to a representative representation in a higher-level programming language.

Understanding how to translate from assembly language to a higher-level representation also requires understanding the target computer processing unit (CPU) architecture in depth. One must know the instruction set, understand the common uses of different instructions and opcodes, be able to notice when a pattern represents a compiler optimization or something anomalous in the code.

Knowledge about computer architecture theory and basics can be gained through advanced undergraduate and graduate computer science courses. Sometimes the courses include hands-on coursework (often with simpler fixed-length reduced instruction set architectures). More detailed knowledge is specific to a particular processor, so many reverse engineers learn this by studying reference manuals for the processor of interest such as the Intel Architecture Manuals [97] while reverse engineering programs. The SMEs mentioned that another way to gain this pattern recognition capability was to write small programs in a higher-level language, compile them, dis-

Table 14 Specialized Knowledge Areas.

Knowledge Area
Translating from assembly language into higher-level languages
System API functionality
System internals knowledge (processes, I/O, synchronization, etc.)
How compilers generate machine code
Classes of vulnerabilities and exploits
Knowledge of and recognition of malware
Knowledge of software protection techniques and how they work

assemble the compiled code, and read through the assembly-level translations while comparing it to what was written.

5.7.2.2 System API Functionality. Knowledge of a system's application programming interface (API) is essential to understanding the behaviors of a program. Nearly all programs use the operating system's API at some level to access the input and output (I/O) functionality of the system. It is the system's API that allows graphics and message box windows to be displayed to the screen, file operations, security functions, process creation, and more.

An operating system's API is specific to that operating system architecture, and college courses in computer science or computer engineering do not usually prepare a person with this knowledge. This type of knowledge is also gained through experience, or by reading specialized texts in software development and performing the exercises found in those texts, such as Petzold [149]. The SMEs reported gaining knowledge of the operating system APIs through reverse engineering programs that use system calls, or reverse engineering the operating system functions themselves to verify what operations they perform.

5.7.2.3 System Knowledge. System knowledge consists of knowledge about the operating system internals and software architecture of a system. It includes an understanding of how the entire ecosystem surrounding the target program works.

This knowledge encompasses an understanding of the internal structures and functions of the operating system, and how the heap, stack and individual stack frames are laid out. It also includes knowledge of the location of different kernel data structures in memory and how to access their contents. System knowledge includes an understanding of how the processor fetches and executes instructions, how the processor implements its functionality and how the program loader works to read the program into memory.

The SMEs reported that expert reverse engineers should understand how function callbacks, asynchronous events, and thread execution work "under the hood" rather than just the name of the system function that implements them. SMEs also outlined that skilled reverse engineers would have a detailed knowledge about how processes and threads work in the operating system, as well as the user and kernel levels in the operating system and how the protection rings provided by the processor are implemented.

The theoretical component of this knowledge can be acquired through upper-level undergraduate or graduate computer science courses in operating systems and computer architecture. However, more detailed knowledge is specific to a processor or operating system and is gained through experience working in or reverse engineering the operating system's kernel. The SMEs also mentioned studying books like Russinovich and Solomon [167] to understand the design and architecture of the operating system the programs run in.

5.7.2.4 How Compilers Generate Machine Code. Another important aspects of domain-specific knowledge is how programs are compiled into assembly instructions. The assembly instructions investigated by reverse engineers have been through the process of compilation from source code into machine code, and then for analysis have been converted back into assembly instructions by a disassembler. The SMEs said that knowledge about how compilers generate machine instructions can help someone recognize the difference between a compiler optimization and an anomalous or malicious code segment.

Additionally, compilers manipulate, parse, and arrange instructions differently, which makes the layout of assembly instructions from one program to another different. It can also change other assumptions like the function calling convention that is applicable to the program. The SMEs expressed learning the theoretical component of compiler knowledge from compiler textbooks like Aho et al. [1] and from college computer science or computer engineering courses. They described gaining more applied knowledge of how each compiler works from experience compiling their own programs with one or more compilers and reading the assembly code that each compiler generates.

5.7.2.5 Classes of Vulnerabilities and Exploits. Vulnerability knowledge includes understanding the different types vulnerabilities that can exist in each piece of the computing infrastructure. This knowledge consists of knowledge about vulnerability classes, knowledge about how to develop exploits, and knowledge of the ways that different exploits can be leveraged on a system.

Understanding vulnerability classes can mean understanding the different phases in which vulnerabilities are generated in system development, what types of systems they affect, what types of errors lead to vulnerabilities, what attack scenarios use them, how they are exploited, and several other facets [122]. In particular, the SMEs identified that reverse engineers must understand in great detail how memory corruption vulnerabilities (like buffer overflows, integer overflows and underflows, null pointer dereferences, heap corruption, format string vulnerabilities, and so on) occur and how to prevent them.

The SMEs mentioned that expert reverse engineers working in vulnerability discovery or malicious software analysis should how to craft an exploit which takes advantage of a vulnerability. This can be as simple as the ability to generate a malicious input from a user prompt or as complicated as crafting a document which allows an attacker to gain elevated remote access when a user opens it in a document reader. The knowledge of how to exploit a system is important for both developing proof-of-concept exploits, and for knowing what constitutes an *exploitable* vulnerability rather than just a bug. The SMEs mentioned that it often takes developing a proof-of-concept exploit before the sponsor will accept that the system is, in fact, vulnerable to attack.

Knowledge of vulnerability classes also includes understanding the ways that attacks are carried out on different types of systems. This involves understanding how attackers identify systems to attack, how they use vulnerabilities to craft exploits, and how they use exploits to attack the systems. It also involves understanding what type of advantage each type of attack gains an attacker.

Finally, understanding vulnerability classes involves understanding the systems in which the vulnerabilities are found. Hardware vulnerabilities often involve misconfigurations and improper assumptions made during the design of a hardware component which allow attackers to gain access to write to or read from protected devices or segments of memory. Operating system vulnerabilities involve misplaced assumptions in the design of the operating system software or any of the software that the operating system puts trust in. Application vulnerabilities involve ways in which applications can be made to perform operations that violate the interests of users of these applications or system administrators. Web-based vulnerabilities involves understanding software implementation flaws where web-exposed code with logic errors can allow a person to access information and gain unauthorized privileges on a web server. Though all of these vulnerability types involve unauthorized access and control, each requires its own extensive domain knowledge to for a person to be an expert at finding and analyzing these vulnerabilities.

5.7.2.6 Knowledge and Recognition of Malware. Reverse engineers working in malware analysis rely on a wide range of knowledge about the functionality malware can exhibit. SMEs described knowing about what malware does at a high-level, and also an ability to recognize and interpret malicious behaviors when they are seen in a program.

Malware knowledge involves understanding the behaviors, mechanisms, and manifestations of how rootkits, worms, viruses, Trojan horses, botnets, and other types of malicious software work. It also involves understanding how different classes of malware are implemented on the target operating system and processor.

SMEs also made reference to knowledge and use of good “lab practices.” Best practices in analyzing malware involve knowing and being able to apply memory forensics to extract malicious software from a computer system without tainting the trail of evidence or losing essential data. It also means understanding the effects malware can have on a host system and what precautions must be taken in to protect the reverse engineers' systems and networks from the effects of the malicious software.

To the SMEs, understanding malware also means understanding how the malware is protected, and being able to get around those protections to analyze the program.

5.7.2.7 Knowledge of Software Protection Techniques and How They Work.

Programs that employ software protections employ them to prevent reverse engineers from achieving their analysis goals. The SMEs referenced several instances of encountering software protections while analyzing malicious software or when facing an industrial protection employed to prevent piracy, tampering, or reverse engineering. An important element of reverse engineers' specialized knowledge is in understanding software protections, how they work, how they can be defeated, and in understanding other ways to perform the same tasks when they cannot be defeated.

Protection knowledge referenced in the interviews involved understanding the different types of protections, which among others can include:

- Static analysis protections,
- Dynamic analysis protections,
- Obfuscations,
- System hardening protections,
- Virtualization-based protections,
- Packing, and
- Encryption.

Reverse engineers that analyze malware or software protections need to know about these areas and how to break or circumvent these protections when they stand in the way of analysis. If breaking a protection is not feasible, the reverse engineer must know what kinds of actions these protections inhibit so they can generate alternate actions to accomplish roughly the same things.

5.7.3 Automaticity and Tacit Knowledge.

The SMEs were prompted for information about their tacit knowledge in the interviews. First, they were asked directly about processes which had become automatic for them. Afterwards, they were asked to explain what elements of reverse engineering they thought would be difficult for a novice to perform, provided the novice was given detailed instructions on how to complete the task.

Tacit knowledge is typically thought of as compiled experience that allows an expert to act and make decisions in a task more quickly and more effectively than a novice [59, 137]. The SMEs described the ability to recognize high-level programming constructs, recognize anomalies in assembly language, and to recognize unique solutions to problems as the primary tacit components of expert knowledge.

5.7.3.1 Recognizing High-Level Programming Constructs.

SMEs talked about the ability to recognize different programming constructs in assembly language as one of those things that takes a lot of experience to be able to do. An expert might be able to easily recognize a certain configuration of data as a data structure in memory, a two-dimensional array, an object instance of a class, or a grouping of functions. The SMEs described being able to retain the ability to think about the low-level machine code as the higher-level code structures that they were compiled from. This knowledge allowed them the capability of concerning themselves about program behaviors rather than syntax-level details. It also helps them more quickly think about what elements in the program are important to their goals and which ones are not.

5.7.3.2 Recognizing Anomalies.

The SMEs also discussed their abilities to recognize when the code looked anomalous. Each SME mentioned having an ability to notice if a section of assembly code looks “weird” or if something in the program “just looks off.” These cues can be an indication that the program is performing unexpected (and possibly malicious) functionality or at least that the reverse engineer has to do more investigation to learn what is happening.

The SMEs described that beginners might not be able to tell the difference between code that is doing something unusual or tricky and normal compiler optimizations. However, experienced reverse engineers can look at the same code and notice the same compiler optimizations they have seen many times. They credited these recognition capabilities for giving them an intuitive *feel* for what is normal code and what could be abnormal protected or malicious code.

5.7.3.3 Recognizing the Approach is Wrong. The SMEs also referred to having “a sixth sense” that the problem-solving effort was going in the wrong direction or “down the wrong path.” One SME compared approaches in the past to the approaches of other reverse engineers who “beat down a path until it’s dead.” This element of tacit knowledge involves being able to recognize the cues that progress is or is not being made in the task, and being able to compare that against an expectation about how the task should be progressing.

5.7.3.4 Recognizing Unique Solutions to Problems. Each of the SMEs discussed the component of “out of the box” thinking that they perform in reverse engineering. They mentioned how some reverse engineers are able to think about a problem and come up with creative solutions that are not directly apparent from the way the problem presents itself. This ability to recognize a problem as an isomorph or analogy to another problem is part of what one SME called the “dark art” of reverse engineering.

The SMEs related stories of getting what seemed like a crazy idea out of nowhere about how to approach a difficult problem. One SME mentioned going through a process of clearing out all thoughts and not thinking about the problem directly in order to help allow creative solutions to come. In their relations of stories, the ultimate solution to their problems incorporated bringing in elements of knowledge that were outside of the scope of the reverse engineering task, but which made sense in some metaphorical way to a separate and seemingly unrelated approach, often within a different problem domain. Regardless of the approach to creativity, each SME mentioned having an intuitive feel for different ways that a complex problem could be approached which ultimately provided them the ability to perform the task more quickly than other reverse engineers.

The SMEs independently referred to reverse engineering as "putting together a puzzle." This implies that there may be abstract problem-solving and reasoning activities which are involved in solving puzzles performing reverse engineering work. These patterns or activities in the puzzle domain and the reverse engineering domain seem to be similar enough to each other to make the metaphor resonate independently with each of the SMEs.

5.8 Conclusions

In this chapter, a semi-structured interview study with subject matter expert reverse engineers was described to uncover the procedural and conceptual components involved with reverse engineering executable programs. From the study, four primary work domains were defined in software reverse engineering:

- Vulnerability discovery,
- Malicious software analysis (including looking for rootkits and backdoors),
- Software protection analysis, and
- Reverse engineering unprotected software.

Across those four domain areas, the study uncovered eight primary goals that are involved with reverse engineering software. These goals are:

- Understand the purpose of analysis,

- Finish the analysis quickly,
- Discover general properties of the program,
- Understand how the program uses the system interface,
- Understand, abstract, and label instruction-level information,
- Understand, abstract, and label the program's functions,
- Understand how the program uses data, and
- Construct a complete "picture" of the program.

After discussing the procedural aspects of reverse engineering, the conceptual aspects were described, including how information is used in reverse engineering tasks and what knowledge is required. The SMEs described information in reverse engineering tasks as providing the means by which they determine and manage their approach to reverse engineering. Information-seeking activities were characterized as passive, active, monitoring, and trustworthiness-related activities.

The conceptual knowledge areas from reverse engineering were described in Section 5.7.2. First, the general knowledge areas were presented and then the following specialized areas of knowledge were presented and described:

- Translating from assembly language into higher-level languages,
- System API functionality,
- System internals knowledge,
- How compilers generate machine code,
- Classes of vulnerabilities and exploits,
- Knowledge of and recognition of malware, and
- Knowledge of software protection techniques and how they work.

The next chapter focuses in on the procedural aspects of reverse engineering through the analysis of an observational study to elicit the sensemaking process from reverse engineers performing a crackme task. The observational study was designed using information from the study in this chapter about the goals involved in reverse engineering and the processes that are shared across the four different reverse engineering problem domains.

6. Observational Study

6.1 Introduction

As part of the overall methodology in this dissertation, a case study (Chapter 4) explored conceptual and procedural aspects of a situation involved in the task of reverse engineering a simple executable program and a semi-structured interview study (Chapter 5) produced conceptual and procedural aspects that subject matter expert reverse engineers consider important in understanding an executable program. This chapter presents an observational study aimed at characterizing the behaviors involved in reverse engineering a program and eliciting the cognitive process of sense-making involved in carrying out those behaviors.

First, an outline of the requirements of the Angler task is presented. After that, the strategies and task performance of each of the reverse engineers participating in the study is discussed in detail. Next, the verbal protocol analysis to extract the process involved in making sense of a program in reverse engineering, followed by a description of each of the major steps in the process. The last section brings the various pieces together in a theory of sensemaking and presents two examples from the observations of reverse engineers to describe this theory in the context of the sensemaking behaviors outlined in this chapter.

6.2 Overview of the "Angler" Task

For the reverse engineering task, a crackme program called "Angler.exe" from crackme.de [172] was used. IDA divides the Angler program into 15 major subroutines, including those that control the `WinMain` function that starts the windowing process and subroutines to present dialog boxes. The program runs within a single thread of execution in memory. Angler's file header contains pointers to four program sections which are mapped into memory at run time: the `.text`, `.rdata`, `.data`, and `.rsrc` sections, which are typical for portable executable programs running on Win32 operating systems [65]. The program does not show any indications of having hidden sections, encrypted code, or code obfuscations.

Though there are many strategies to approaching the challenge, in general participants had to:

1. Read and understand the goal of the task,
2. Determine that it is a Win32 system function that handles input data,
3. Isolate the function that manipulates the text from the user and serial number text boxes,
4. Determine that the serial string must be in a particular format,
5. Determine how to "catch" the user-provided text as it flows through the function,
6. Understand what data inputs lead to the success message in the function,
7. Translate the behavior of the function into pseudocode, and
8. Write pseudocode for a key generator which produces a key for any given user. name

Reverse engineering the serial number processing algorithm in the Angler crackme is an involved task which was expected to take longer than an hour to complete. The program code handling the input string was a lengthy subroutine composed of a series of 27 basic blocks of assembly instructions, each of which had checks for the correct value at the end of each block.

The algorithm worked by taking the first four characters of the person's name and performing a cyclic redundancy check (CRC) to produce an even-numbered value from that character. After that, the function finds four pairs of prime factors, each of which sum to one of the even-numbered CRC values. Once the factors are found, they are assembled into a format string representation separated by dashes [57]. If the reverse engineer does not understand these behaviors, the program appears to be making a large number of arbitrary numerical checks in a very long sequence of assembly instructions.

Table 15 Participant Years of Experience ($n = 4$).

	Mean	**Standard Deviation**
Computer Programming	7.25	1.8
Reverse Engineering	1.5	0.8

6.3 Observations of Task Performance

Four participants were observed performing the Angler task. The task performers were different individuals than the subject matter experts consulted in the semi-structured interview study in Chapter 5. The group was composed of one Air Force service member, two government civilian employees, and one contractor. The group had an average of 1.5 years of experience in reverse engineering and just over 7 years of programming experience (Table 15). Each participant had at least six months experience reverse engineering software. None of the participants had prior experience with this particular reverse engineering task although three had solved similar crackme challenges before.

All of the participants chose to use OllyDbg, IDA, and Immunity to complete the task. Through the process of reverse engineering, the participants switched between IDA and OllyDbg to get different representations of "what the program is doing."

As anticipated, none of the participants were able to finish all eight steps within the time frame. At the point where the participants' task periods ended, Participant A was working on the second step, Participant B was on the third step, Participant C was working on step 6 and Participant D had just begun step 7. In the next four sections, each participants' overall problem-solving in the task is described. After that, the analysis of verbal data is presented which characterizes the sensemaking processes in terms of goal-directed planning.

6.3.1 Participant A. Participant A spent the majority of the session trying to find a representation for the program that would be understandable. The person looked for ways to translate what the program was doing into higher-level programming language representations, but was not comfortable enough with the tool set and with assembly language to be able to get into the advanced stages of analyzing the program. The participant began by opening the program, then opening Immunity Debugger to attach it to the process. The person generated the input "AAAAAAAA" and looked through the assembly code to find where the text was located. The person also generated an MD5 hash of the program using IDA and pasted the hash value into the Angler text window to see what would happen. When these strategies provided no usable information, the person was not able to get traction with the program understanding task. Participant A did not show familiarity with the IDA tool and spent a large portion of the time trying to understand how to show the assembly code within IDA's window.

6.3.2 Participant B. Participant B started the task by dragging the icon of the program into OllyDbg and looking through the program's assembly code. When the assembly code representing a structured exception handler became visible, the person stopped and read through the structure, but it was not clear if the person recognized the structure. Later, Participant B opened up a window within IDA to examine the text strings in the program to gather information about the program that might be useful. During this process, the person mentioned looking for a password or a serial number hardcoded in memory.

Participant B used information from IDA to go back into OllyDbg with the goal of looking for a specific address (`0x0040708C`). After not finding anything usable at that address, the person stated a hypothesis that the characters representing the correct serial number were not stored anywhere, but were instead constructed through a number of data manipulations.

Participant B stepped into the program's function calls until the program's execution landed in imported USER32.DLL code, which was evidenced by the address range `0x7C8XXXXX` and the DLL name referenced in OllyDbg's register pane. The person saw the string "allusersprofile=C:\Documents and Settings\. . . " and interpreted this to mean that a file was being created that reads a serial number from a user's profile. The person then spent the majority of the time parsing a string processing routine in the DLL. This participant did not appear to recognize that the instruction pointer had dropped down into a DLL function, and thus spent a significant portion of the task debugging a file name processing algorithm in a Windows system function.

Participant B also showed difficulty determining if the task was going in the right direction. The person started down the first path that made sense, but it was not the most effective path to solve the problem. The person had difficulty recognizing pieces of information provided by the task environment which could have led to a hypothesis to drive activities in the task. Participant B became caught up in syntactic issues with the assembly language and did not show the development of a "whole picture" of the program. When the session time expired, the participant had not yet found the function that processes the user and serial text. It was not clear that this had become a goal.

Participant B cited the most mentally difficult thing as understanding the purpose of one of the string processing instructions which was encountered and how it worked. This further demonstrates that the person was working strictly at the syntactic level of the assembly code rather than on putting together a complete picture of the program.

6.3.3 Participant C. Participant C started the task by looking at the properties of the program. This participant monitored the program with Regmon [166] to gather information about the program from the registry keys the program used. The person then used this information to open the program in IDA and to gather more information.

Participant C saw the text "Cyclops" and developed the hypothesis that the name either came from within the program or was read from a file. To test this, the person looked in FileMon to see what file operations the program performed. The person then looked for strings in the program through IDA's interface, then used the Data XREF capability in IDA to see what locations referenced that data. Through that process, the person determined that the function where that information was used was an initialization dialog of some sort, so the person labeled the subroutine "initializeDialog" in IDA.

Later in the task, the person came across references to the bitmap file which was used for the face of the Angler window. The discovery of this reference led the person to a hypothesis as to whether the bitmap was stored in the application or whether it was stored outside of the program. Some searching led to no real results, and Participant C gave up this pursuit to gather different information about the program.

Participant C noticed that a function called `GetDlgItem` was called after the subroutine previously labeled "initializationDialog," so the person looked in the MSDN documentation to find how `GetDlgItem` was structured. The person set breakpoints after the "initializationDialog" subroutine and then executed the program so it ran up to the breakpoint that had been set. In doing this, the participant gathered information that a message box appears, but that the program is not blocking or waiting for input at that point in the program.

The person recognized that an event handler most likely dealt with the button press events on the Angler window, so the person looked for where the event handler was located. The person had a question about how the event handler would work. If the event handler involved buttons, then it would appear one way, and if it involved an area of the screen or an area of the window, it would work differently. This led to the determination that no matter what, an event handler of some type would be necessary.

After finding what appeared to be coordinates, where a button is being placed on the screen, the participant "got lost" in the task and became unsure as to how to approach solving the problem. This led to starting over in the reasoning process in which the person put together a mental model of causes involved in the program. These related the message box to being called by the function, which gets called by an action handler. This also led to a gap in understanding what the `GetProcAddress` function is and how it works. The person sought out documentation from MSDN in understanding this function.

After reading through the MSDN documentation, Participant C determined that the function in question might be the function that handles the text strings from the user. The participant traced through the function and decided that it looked logical algorithm, but that it was probably not related to the text from the dialog. Further down in the program, Participant C noticed the format string and decided the program was looking for the serial number in a particular format. The participant planned out an approach to seeing if the function is ever called in the program which involved setting breakpoints and then putting data in the text box to see if it is ever called. In the approach, the person planned to track the data as it goes through the function.

The person came across the `GetDialogItemText` function call, but after recognizing a lack of knowledge concerning the values the function returns to the program, the person consulted the MSDN documentation. This led to learning that the `GetDialogItemText` function returns a pointer to the string on the stack. Participant C was not sure how items got stored on the program stack and had to restart the program with the goal of determining if the stack contained a pointer to a string when the program executed past the call to `GetDialogItemText`. The person restarted the program and determined the program did return a pointer to a string on the program stack. This information led to the hypothesis that the string represented the user-input serial number, which enabled the participant to locate the code for the serial number processing algorithm.

Upon finding the function that handled the entered serial number, the person started to determine the properties of the function. These included the property that the subroutine checks for the proper number of characters, that the function checks for 15 characters, and that if there are less than 15 characters, the program jumps to another location. The person determined that the location does not have code that leads to the success message. Participant C used incrementally gained information in order to gather new information about the function, such as the observation that the function splits the input string, and then that the function splits the serial number into eight-bit chunks.

Participant C elaborated on this strategy as well. The person made a number of comments that tied address locations in the code to future plans, such as: "that's the spot I want to break on the next time I run the executable." The person used information gathered to incrementally piece together a plan to see what the value was on the stack when the function returns and then trace forward to see how the program uses the value. The participant was working on discovering what data values led to the function working correctly when the time for the session expired.

6.3.4 Participant D. Participant D also started the reverse engineering task by taking note of the program's properties before running the program. The person quickly looked at the program using a number of reverse engineering tools to see if the program was protected or whether it appeared to be packed.

After looking at the program's properties, Participant D ran the program and immediately formed a background hypothesis that a system call was involved. This person used the OllyDbg tool to list all of the program's intermodular calls and saw the `WriteFile` and `ReadFile` system calls. The person specifically looked in the intermodular call display to find the `GetDialogItemText` function and found a version of the function in the list of system calls as `USER32.GetDialogItemTextA`.

Once Participant D had this knowledge, the person navigated to the function directly from within the tool and started to investigate how the program handled user-input data. The person saw the format string comment to the right of the disassembly and set the tool to highlight "interesting" code branches. The highlighting led the person to notice four calls to the same function, which identified that function as a potential point of interest. Participant D used this information to determine further information about the function.

At the end of the function, the person saw an offset address and wanted to "see if it is important." The person used OllyDbg to show all of the jumps that the program takes, then looked for the success message at the end of the jumps. The person identified this message in the assembly code, possibly from the decoded ASCII in the comment section, and then set out to determine how the program's execution got to that location. The person tried a few different approaches and ultimately gave up that path to gather more information.

The person went back to the current instruction pointer and began stepping over code. The person wrote down the jumps within the serial processing function to see which one of them led to the success message. The person restarted the program and generated an input and had the hypothesis "EAX should have the value . . . ."

The person developed several approaches to solving the problem:

- Setting a breakpoint on the stack variables to see where they are changed, then restarting,
- Going to the target address and isolating instructions that write to them, and
- Setting a hardware breakpoint and evaluating writes to the address.

The person chose the third option and started running the program until it hit each of the breakpoints. This approach produced too many instructions that wrote to that address, so it became too overwhelming and the person gave up that approach and cleared the hardware breakpoints.

Participant D came to understand the function as one large processing algorithm, and showed how it could easily be patched. After asking for direction, instruction was given to follow the instructions that came with the crackme program. The participant began translating the function into higher-level representations and instantly became very bogged down. The translation portion of the task was very difficult to keep track of and appeared to be time consuming and tedious. The person got lost more than once in the translation and had to backtrack to keep up with current progress. The person made guesses about the meaning of a jump, followed the jump to gather information, and built criteria to evaluate what the information would mean while following the jump.

Time expired as Participant D was translating the algorithm into pseudocode. The person spent a lot of time making sense of the serial processing algorithm and got bogged down in the conversion of characters to hexadecimal values. This translation process progressed much more slowly than finding the location of interest and determining how the function worked at a high level.

Overall, the participant was able to quickly locate the subroutine of interest and could spend time understanding the format that the serial number was expected to be in, how the name and serial number data flowed through the program, and how the serial number processing function worked with other parts of the code.

6.4 Verbal Protocol Analysis

The researcher reviewed the video and verbal data from each participant and transcribed it into a spreadsheet, broken into one verbal segment per row as discussed in Trickett and Trafton [194]. Participant B's video and audio data recordings were accidentally destroyed during a problem with saving the video file to disk and were not able to be recovered or transcribed. Participant A's problem-solving was transcribed, but was determined to not provide value to understanding the sensemaking process and was not included in the coding.

For the two remaining participants, their verbalizations were segmented to represent a single idea during transcription in order to take advantage of other contextual clues from the audio and video. When a segment contained a shift from one idea to another, the second idea was recorded in its own row as its own new segment. Where significant verbal breaks occurred, the subsequent verbalization would also be recorded on a new row as its own segment. When actions in the video data were observed, a row was created but was not annotated with text.

Table 16 Rules Used to Code Segments.

Verbal Segment	**Coded As**
Describes a desired future state	Create goal representation
Describes activities to accomplish goal	Plan approach
Refers to performing an ongoing activity	Carry out plan
Refers to status of ongoing activity	Carry out plan
Verbalizes noticing information	Sense information
Refers to recognizing relevance of information	Interpret information
Makes statement from relevant information	Update knowledge
Offers an assumption	Generate hypothesis
Asks question about an object	Generate hypothesis

After all of the available data was transcribed, it was coded according to the following coding taxonomy from the literature review in Chapter 2:

- Create goal representation,
- Plan approach,
- Carry out plan,
- Sense information,
- Interpret information,
- Update knowledge, and
- Generate hypothesis.

The coding taxonomy was developed from categories described in the literature review on sensemaking in understanding programs (Section 2.5) and from the portion of the case study in Chapter 4 which related to the procedural aspects of program understanding. The taxonomy contains functions involved in the standard information-processing loop used in programming artificial agents to interface with the environment [165], with the addition of involving the generation of goals and hypotheses.

The data from the two participants were coded according to the coding rules in Table 16.

The researcher coded all of the data (592 segments) and afterwards a second coder independently coded 29.2 percent of the segments (173 sequential segments) from a starting point randomly selected by the second coder. Cohen's Kappa [47] was computed to measure interrater reliability for the 173 segments coded by both coders. Cohen's Kappa measures the agreement between coders on positive and negative instances while taking into account the likelihood of agreement based on chance. Cohen's Kappa is computed as:

$$\kappa = (P_o - P_c)/(1 - P_c)$$

P_o is the proportion of agreements between the coders and P_c is the proportion of agreement which would be predicted by chance. Cohen's Kappa of 0.0 to 0.4 indicates little agreement, 0.6 to 0.8 indicates significant agreement, and 0.8 and above represents near perfect agreement [41, 194]. As recommended in Trickett and Trafton [194], after both coders independently coded the data, the interrater reliability was calculated and the coders met to discuss disagreements. If codes had weak interrater reliability (0.4 or below), the categories would be removed or changed and the data was recoded.

Disagreement and re-categorization occurred once during the coding process. The code "Create Goal Representation" initially was divided into two distinct categories: "Create Goal" and "Create Goal Representation," but in the initial coding, there was not enough agreement between the coders to keep both categories. They were merged into the category "Create Goal Representation" and the data was recoded. The final interrater reliability for the dual-coded verbalizations was 0.82, which demonstrates significant to "near perfect" agreement in all categories (Table 17). Following standard practice, the remainder of the verbalizations were coded by the researcher [194].

6.4.1 Computing State Transitions. The state transitions from two of the tasks were computed to determine in what order reverse engineers perform each of

Table 17 Interrater Reliability of Coding Scheme (173 segments).

State	Category	Cohen's Kappa
a	Create goal representation	0.93
b	Plan approach	0.82
c	Carry out plan	0.78
d	Sense information	0.72
e	Interpret information	0.75
f	Update knowledge	0.81
g	Make hypothesis / assumption	0.92
	Average agreement	0.82

these processes. Transitions between the states indicated movement through the problem-solving process

As described in Bakeman and Gottman [8], matrices of state transition probabilities were computed to determine the sensemaking process used in the task. For m states S_j and S_k, and n segments i, the total transitions between each state S_j and state S_k are computed as:

$$Tr(S_j, S_k) = \sum_{i=1}^{n} (S_{i,j} \times S_{i+1,k}) : S \in (0,1) \tag{1}$$

Where $S_j = S_k$, the transition is computed as a self transition and accounted for as such. The total transitions departing a state S_j are computed as:

$$Tr(S_j, out) = \sum_{k=1}^{m} Tr(S_j, S_k) \tag{2}$$

The total transitions entering a state S_k are computed as:

$$Tr(in, S_k) = \sum_{j=1}^{m} Tr(S_j, S_k) \tag{3}$$

The overall transition probabilities for state S_j to S_k are computed as:

$$P(Tr(S_j, S_k)) = \frac{1}{2}(\frac{Tr(S_j, S_k)}{Tr(S_j, out)} + \frac{Tr(S_j, S_k)}{Tr(in, S_k)}) \tag{4}$$

Table 18 Number of Transitions (Participant C).

$Tr(S_j, S_k)$	**a**	**b**	**c**	**d**	**e**	**f**	**g**
a	3	5	2	3	4	0	1
b	3	5	5	8	1	0	4
c	2	3	1	11	3	2	0
d	3	3	4	26	24	3	8
e	0	4	8	14	15	8	10
f	1	0	0	3	3	4	8
g	6	5	2	6	9	2	19

Table 19 Transition Probability Matrix (Participant C).

$P(Tr(S_j, S_k))$	**a**	**b**	**c**	**d**	**e**	**f**	**g**
a	0.17	0.24	0.10	0.10	0.15	0.00	0.04
b	0.14	0.20	0.21	0.21	0.03	0.00	0.12
c	0.10	0.13	0.05	0.33	0.09	0.10	0.00
d	0.10	0.08	0.12	0.37	0.37	0.10	0.14
e	0.00	0.11	0.25	0.22	0.25	0.28	0.18
f	0.05	0.00	0.00	0.10	0.10	0.21	0.29
g	0.23	0.15	0.07	0.10	0.17	0.07	0.38

State transitions for Participant A were not computed because the participant did not complete enough of the task to elicit useful data about state transitions. The record of Participant B's verbal data was lost through a computer error as described above. The state transitions for the two remaining participants are shown in Table 18 and Table 20, respectively. The transition probabilities for the two participants are shown in Table 19 and Table 21, respectively.

Table 20 Number of Transitions (Participant D).

$Tr(S_j, S_k)$	**a**	**b**	**c**	**d**	**e**	**f**	**g**
a	10	12	8	10	7	0	1
b	8	8	9	11	4	0	3
c	3	4	2	10	4	3	3
d	5	13	4	47	19	9	6
e	9	3	3	15	12	8	1
f	5	1	2	6	4	10	4
g	7	2	1	5	1	2	2

Table 21 Transition Probability Matrix (Participant D).

$P(Tr(S_j, S_k))$	**a**	**b**	**c**	**d**	**e**	**f**	**g**
a	0.21	0.26	0.22	0.15	0.14	0.00	0.04
b	0.18	0.19	0.26	0.18	0.09	0.00	0.11
c	0.08	0.12	0.07	0.22	0.11	0.10	0.13
d	0.08	0.21	0.09	0.45	0.28	0.18	0.18
e	0.18	0.06	0.08	0.22	0.24	0.20	0.03
f	0.13	0.03	0.07	0.12	0.10	0.31	0.16
g	0.25	0.07	0.04	0.15	0.03	0.08	0.10

The mean transition probabilities were computed as $1/N \sum_h^N P(Tr(S_j, S_k))$ for participants h through N. The mean transition probability was $\mu = 0.14$ and the standard deviation was $\sigma = 0.09$ and a threshold for the significance of a transition was set at $\mu + \sigma = 0.23$. A list of the significant transitions at the threshold $P(Tr) \geq \mu + \sigma$ is shown in Table 22.

The transition probability values from the two participants were compared using test from Anderson and Goodman [7] to determine whether two samples could have come from the same underlying Markov chain ($H_o : P \neq P^o$). In this notation, $n_{jk} = \sum_j Tr(in, S_k)$. The equation $\hat{p}_{jk}^{(h)} = n_{jk}/n_k$ represents the maximum likelihood estimates in the probability matrix for participant h. The pooled maximum likelihood estimates $\hat{p}_{jk}^{(pooled)}$ represents the maximum likelihood estimates obtained by adding the data from both participants, and $C_{jk}^{-1} = (1/n_{jk}^1) + (1/n_{jk}^2)$. For two participants, the test is:

$$\chi_{jk}^2 = \sum_j C_{jk}(\hat{p}_{jk}^{(1)} - \hat{p}_{jk}^{(2)})^2 / \hat{p}_{jk}^{(pooled)} \tag{5}$$

For this test, $\chi^2 = 201.97$ which is above the critical value of 63.69 which rejects the null hypotheses that the two participants' data do not come from the same underlying first-order Markov chain with significance $p = 0.01$ and $(rows - 1)(columns - 1)$ degrees of freedom.

This test provides some evidence that there exists an underlying process of sensemaking which may be very similar to the processes elicited from these two par-

Table 22 Prominent Transitions ($P(Tr) \geq \mu + \sigma$).

S_j	S_k	**Mean transition probability**
a	b	0.25
b	c	0.24
c	d	0.28
d	d	0.41
d	e	0.33
e	e	0.25
e	f	0.25
f	f	0.26
f	g	0.24
g	a	0.24
g	g	0.23

ticipants. Nevertheless, before establishing its generality, this sensemaking process should be further investigated in studies with greater numbers of participants and in different task domains. The problem-solving processes of these two participants are useful to provide a framework for thinking about and researching how people make sense of situations.

6.5 Sensemaking in Reverse Engineering

Figure 15 shows a process of how the sensemaking behaviors were used by reverse engineers attempting to solve the Angler task. The processes encapsulated within each of the states are described below. A theory of sensemaking is then presented which incorporates these sensemaking behaviors into an overall theory of how people understand executable programs using assembly code representations.

When the participants were working on problems in the task, they continually moved through a sensemaking loop, which included the establishment of a goal representation, a plan to achieve the goal, carrying out the actions of the plan, sensing information from the task environment, interpreting the information, potentially updating knowledge if the information was relevant, and developing hypotheses based on the new knowledge. The sensemaking loop shown in Figure 15 shows this cycle

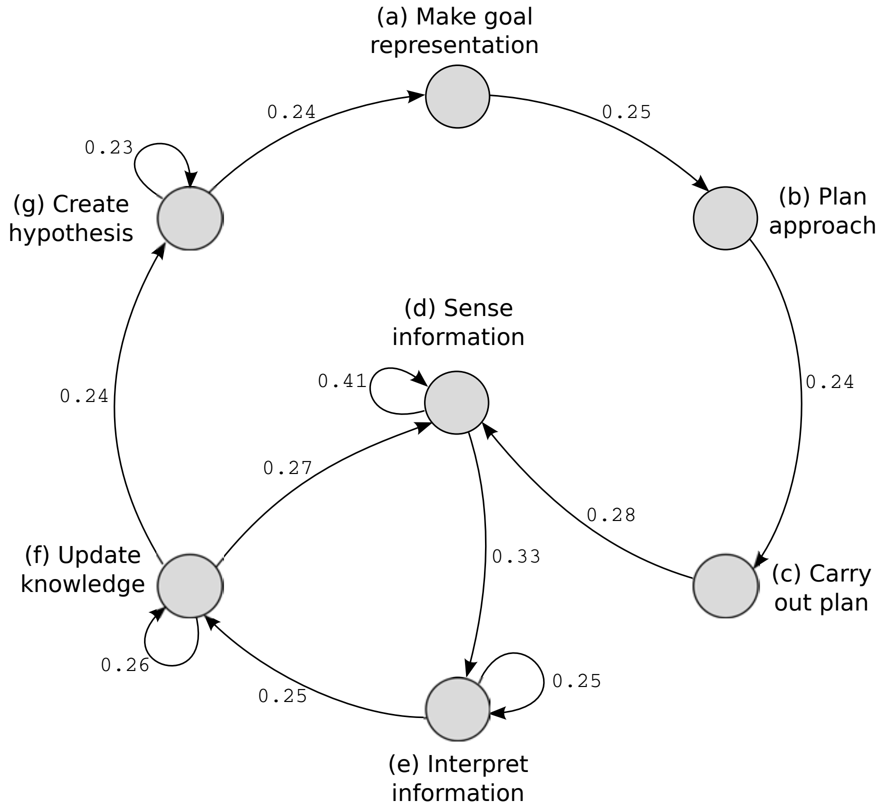


Figure 15 Sensemaking Processes from the Angler Task.

as a Markov model populated with the probabilities that each state transition would occur.

While progressing through this process, the reverse engineer gathers information and constructs or refines a mental model of the program. The program involves different components, such as functions, the code's execution path, data in the program, and sequences of instructions. As a reverse engineer works on the problem, he or she gathers information about these components and relates them to items in the task environment pertaining to elements of the program.

6.5.1 Create Goal Representation. When a reverse engineer generates a goal representation, the person is expressing what they want in terms of features of the desired state. This goal representation can be in the form of a desired configuration of the situation, or as a gain in information about the program. Both types of goals can be represented in the same way.

For example, some of the goals in the data from Participant C include: information about the application as it runs, information about what Angler was doing at a given time, information about whether a file is read into the program, information about where the string "Cyclops" is used, and information about how variables change in the serial number processing function. Goals from the data pertaining to situations involve the desire to be at a particular point in the program's execution, and to "catch" the program after the dialog box handling function is called.

6.5.2 Plan an Approach and Carrying out the Plan. Once a reverse engineer had verbalized a goal, this often was immediately followed with the development of a partially-constructed plan of actions which would enable the attainment of the goal. If the goal was an information goal, the plan involved actions which were believed to help gather the required information. If the goal was a situational goal, the plan involved sequences of actions which were believed to configure the situation in the desired manner.

Table 23 Deliberation on a Plan.

Verbalization
"I'm not sure what happens after the dialog gets initialized"
(7 seconds silence)
"Maybe if I try debugging it and stick a breakpoint after what I think is the initialize function"
"It won't hit the function until it's done initializing"
"And maybe it goes into waiting or something"
"So I'm going to try debugging the application"
"And set a breakpoint after what I think is initialize dialog"

Sometimes a goal readily lent itself to a plan which was formed and followed immediately. An example of this is the goal implied by the segment: "I'm going to set a breakpoint on this instruction." In this case, the actions involved seem straightforward and can be accomplished immediately:

1. Shift attention to the instruction,
2. Encode that is the correct instruction,
3. Press up until the selection indicator is on that instruction,
4. Press the F2 key to enable the breakpoint, and
5. Encode that the color of the instruction has changed to red.

Other times the goal did not directly lend itself to a plan, so there was a deliberation process to construct and evaluate an approach that would generate a usable plan. Sometimes the deliberation was not verbalized, but it was inferred by the presence of long pauses. An example of deliberation over a plan is in Table 23.

Participants determined the best action for their situation by thinking through hypothesized behaviors and inferred future states of the program. A plan as verbalized involves a set of actions in which some of those actions are sequenced. Sometimes the sequencing of actions does not happen until the actions are taking place and conflicts are detected.

Other plans from the participants' data include looking at strings by opening up the strings window, running the program again to see how the stack changes, labeling a function so it can be identified later, inserting a breakpoint after the initialization routine to "catch" the program, writing down a list of function addresses to look up, and tracking data through a function. Each of these plans suggests a number of actions which must be taken. Carrying out a plan simply involves performing the actions specified in the plan.

6.5.3 Sensing Information. Carrying out actions in the task environment often involves changing the configuration of the elements, so information is constantly being sensed. Not all of the information in the task environment is relevant to the reverse engineer's goals, so a large amount of information appeared to be discarded without evidence of having been mentally processed.

A verbal segment was coded as "sensing information" when a participant simply read and stated information from the environment without giving any indications of processing the meaning of the information. Often, this information consisted of hexadecimal values, numbers, and names of labeled items.

Participants sought out information both passively and actively. When participants used passive sensing of information, they were running the program to gather information about the program's behaviors, or looking through the disassembly to gather clues that might be useful later. When participants sought out information actively, they set a goal for what information they wanted, made a plan to acquire that information, and followed the plan by carrying out the task actions. For an example, a participant was observed writing down the addresses of system calls and then using the search features of the debugger to find them. Participants also actively sought out behaviors by isolating phenomena. More than one participant was observed stepping over a system call, then looking to see what had changed in the interval from before the system call to after it had been executed. Participants were also actively sensing information when they made a plan to acquire information about whether strings can be found in memory. The participants entered strings of text characters in the serial text field in the program and then looked for those same values in the program when the program's execution stops at a breakpoint.

6.5.4 Interpreting Information. Instead of merely sensing information content in the environment, participants interpreted the information by applying meaning to the information from their knowledge. Sometimes this dealt with whether or not the information was relevant. Other times it served as a recognition of some structure or component in the program. The following set of verbal segments provides examples to distinguish sensing information from interpreting information:

- "Angler by Cyclops" - sense information.
- "It is the title of the window" - sense information.
- "It doesn't appear anywhere but it's the name of the window" - interpret information.
- "It shows up in the task bar" - interpret information.
- "So, that function gets called." - update knowledge.

When participants did not have the knowledge to understand what they were looking at, their activities were categorized as "sensing information" instead of "interpreting information." Because of a lack of experience with IDA, Participant A was not able to match the affordances in the task environment with concepts about what information representations the IDA tool could provide.

When people explore program behaviors and explore assembly instructions as above, they notice information from the environment and try to connect that information with some knowledge they have. to determine whether the information is relevant and what it means.

6.5.5 Updating Knowledge. When participants in the verbal protocol were interpreting information, they matched an object in the environment to a concept from their knowledge. However, when participants were coded as updating their knowledge, they summarized the information from repeated cycles of sensing and interpretation into a simpler statement which they could potentially use later. After a certain point in reverse engineering the program, as more meaningful information is gathered about the program, the reverse engineer compiles the information into a statement, such as "So, that function gets called" or "that bitmap is stored inside the executable in one of the resources."

Segments coded as "updating knowledge" almost always appeared like summarizations and distillations of the important parts of information which had been encountered, and how it related to existing stored knowledge. After observing the process of sensemaking play out while coding, the second coder referred to being able to detect this process when the person was "compiling" their information into knowledge.

6.5.6 Generating Hypotheses from Compiled Knowledge. Once participants had added new summarized knowledge to their mental model of a program, they often came up with a hypothesis directly afterward. Participants appeared to generate hypotheses in the task after deducing the logical conclusions of new knowledge they had acquired.

Hypotheses and assumptions were generated mainly after participants sensed and interpreted information and updated their knowledge with the implications of this information. The hypotheses that resulted from this process were typically used to generate a new goal, such as to seek out information from the environment to confirm or refute the fact.

Hypotheses often were observed in the form of a statement that can be verified, such as: "It looks like `GetDialogItem` creates a handle to some part of the dialog that's open." In this case, a subsequent sensemaking loop can be started with the goal to verify whether or not the `GetDialogItem` function creates such a Window handle.

Participants gathered information to form hypotheses as well. When participants investigated system calls, they looked up the arguments and return values for the system calls in the operating system's documentation in order to make assumptions about how the system call was being used in the context of the program.

Once a person formed a new hypothesis, it enabled the person to create a new goal to seek information from the task environment about some property of the program, its components, or its behavior. However, when participants did not appear to have the knowledge to develop a hypothesis, the person was not able to generate information-seeking goals and got "stuck" in the task without a way forward. This was seen when Participant A and Participant B followed a set of actions which would not provide them valuable information, and carried those actions out without gaining useful information about the program that could be compiled into knowledge.

When participants reverted to exploring instructions or behaviors of the code in the middle of a problem-solving task, it was often the case that the person had lost sight of any other attainable goal or could not formulate a hypothesis that could be investigated.

6.6 Conclusions

This chapter described an observational study which analyzed data from participants performing reverse engineering tasks. The requirements of the Angler task were described, followed by the details of each of four participants' performance in the task. A verbal protocol was conducted to extract state transition patterns from two of the participants' reverse engineering sessions. From the verbal protocol data, a process of sensemaking was elicited and the steps of that process were described in a theory of how people make sense of executable programs. The next chapter presents the overall conclusions of the dissertation, discusses the implications of the theory developed from the studies in this dissertation, and presents areas for future research investigation.

7. Conclusions

7.1 Overview of the Research

The research problem of the dissertation was to understand how people make sense of programs, and particularly to elicit the conceptual elements and procedural elements of reverse engineering knowledge involved with successful task performance. Chapters 4, 5, and 6 presented three studies that elicited this information from various perspectives.

7.1.1 The Case Study. The first method, presented in Chapter 4, was a case study that investigated the situational factors involved with reverse engineering executable programs. It explored the information and affordances in the reverse engineering tools, the goals, plans, hypotheses, information-seeking behaviors, and attentional focus used in the task. It also examined concepts used in the task and identified the following conceptual themes that can be used to organize conceptual and procedural knowledge content involved with making sense of executable programs:

- System concepts,
- Task environment concepts,
- Situational concepts,
- Cognitive concepts, and
- Background knowledge concepts.

These categories of concepts provided a framework on which the other studies built.

7.1.2 The Semi-Structured Interview Study. Chapter 5 explored conceptual and procedural knowledge further and elicited those elements at a broader level from semi-structured interviews with subject matter expert reverse engineers. The analysis of the interviews produced a number of major goals involved in reverse engineering executable programs:

- Understand the purpose of analysis,

- Finish the analysis quickly,
- Discover general properties of the program,
- Understand how the program uses the system interface,
- Understand, abstract, and label instruction-level information,
- Understand, abstract, and label the program's functions,
- Understand how the program uses data, and
- Construct a complete "picture" of the program.

These goals (and one constraint) are seen by experts as essential to the process of understanding an unprotected program. The information-seeking goals also imply a number of components that are involved in a mental model of the task.

The semi-structured interviews revealed a number of ways that information-seeking behaviors occur and are used in reverse engineering: passive discovery, active information seeking, continuously monitoring information, verifying the trustworthiness of information, and monitoring goal-related cues. Passive discovery involves the goal of "gathering information," while actively seeking information from the environment and continuously monitoring information involve a specific question, a data source, and potentially an assumed hypothesis. Verifying the trustworthiness of information involves goals aimed more particularly at verifying information so it can be relied upon for future goals in the task.

In order for a reverse engineer to successfully make sense of an element of the program, background knowledge is required to interpret the information and to enable the reverse engineer to develop better hypotheses and construct better information goals. The semi-structured interview study produced a group of typical and specialized knowledge requirements involved with reverse engineering executable programs which the SMEs described as essential specialized background knowledge. The specialized knowledge requirements include:

- Translating from assembly into higher-level languages,

- System API functionality,
- System knowledge,
- How compilers generate assembly code,
- Classes of vulnerabilities and exploits,
- Knowledge and recognition of malware, and
- Knowledge of software protection techniques.

Each of these types of specialized knowledge provide a reverse engineer with the ability to interpret different types of information which are required when solving more advanced reverse engineering problems, such as malicious software analysis, software protection analysis, and vulnerability discovery tasks. The organization of goals and concepts from the semi-structured interview study were used to design the observational study.

7.1.3 The Observational Study. In the third study, presented in Chapter 6, an observational study was undertaken to elicit the process of low-level procedural behaviors involved with making sense of a program. The study adapted the taxonomy of sensemaking behaviors constructed in Chapter 2 to categorize the activities of participants working on software reverse engineering tasks.

Observations and notes from four participants were used to represent major goals in the task, and verbal and video data from the two most successful participants was coded and analyzed to build a process model of sensemaking of reverse engineering and the theory about how people use the process to make sense of programs while reverse engineering.

The sensemaking process elicited from the verbal protocol analysis was used to create a theory of reverse engineering which was described in Chapter 6.

7.2 Summary of Research Contributions

This dissertation has resulted in four primary contributions to the scientific body of knowledge:

1. A description of the representational gap in software reverse engineering and conceptualization of reverse engineering as a sensemaking task,
2. The decomposition of situational factors in the process of understanding executable programs,
3. An elicitation of the structure and content of concepts and procedures in reverse engineering from subject matter experts, and
4. A theory that describes the process of sensemaking in reverse engineering.

These contributions provide the way to bridge the representational gap between executable representations of programs and high-level conceptual representations that reverse engineers use in practice.

7.2.1 Conceptualization of Software Reverse Engineering as a Sensemaking Task. The description of the representational gap in software reverse engineering and the process of "sensemaking" in reverse engineering was elaborated in Chapter 2. This description provided a conceptual framework with which to approach how reverse engineers understand executable programs from assembly language representations.

7.2.2 Situational Aspects of Reverse Engineering Executable Software. The decomposition of situational factors involved in understanding executable programs was presented in Chapter 4. This contribution analyzed and described different abstractions used in reverse engineering to represent aspects of the task environment, knowledge, and the task's requirements and how they interact in a reverse engineering situation.

7.2.3 Structure and Content of Concepts and Procedures in Reverse Engineering. The elicitation of the structure and content of concepts and procedures involved in reverse engineering executable programs is found in Chapter 5. These concepts and procedures provide the foundation for higher-level goals, rules, and concepts necessary to automate aspects of program understanding tasks.

7.2.4 Theory of Sensemaking in Reverse Engineering. The culminating contribution of the dissertation was the development of a theory of sensemaking in reverse engineering. The theory involves a cycle of seven sub-processes:

- Goal representation,
- Planning,
- Carrying out a plan,
- Sensing information,
- Interpreting information,
- Updating the mental model, and
- Generating a hypothesis.

Unlike other theories of sensemaking, this theory provides clues to how this process could be computationally realized and makes claims about how people interact with the environment which can and should be empirically tested.

The theory of sensemaking in reverse engineering proposes that reverse engineers make sense of programs through goal-directed information seeking to discover the purpose of the program, properties of the program, the program's system interface, the program's instruction-level and function-level information, and how the program uses data. The process of sensemaking is developed in Chapter 6 and is presented in Figure 16.

Figure 16 presents a conceptual framework to describe the different elements of conceptual and procedural knowledge involved in how people make sense of executable programs. The diagram depicts subgoals required to develop a mental model of a program along with sensemaking loops for each subgoal. The subgoals shown are

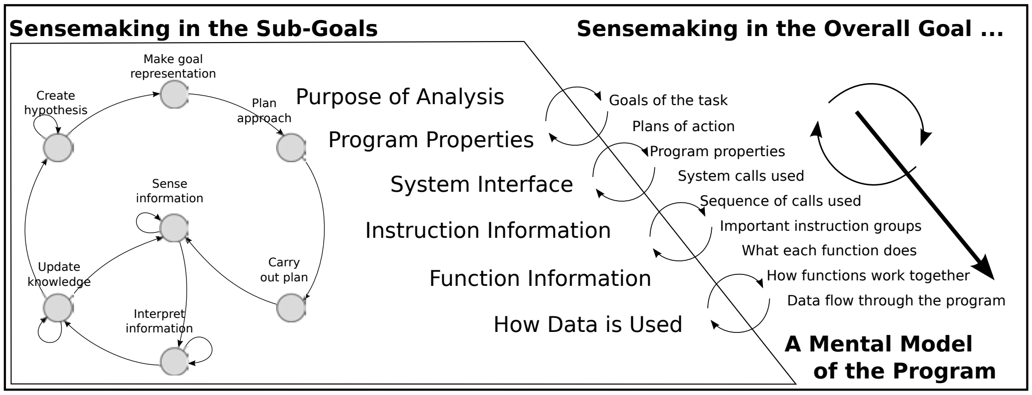


Figure 16 Sensemaking in Reverse Engineering.

those identified from the semi-structured interview study with subject matter expert reverse engineers.

As a reverse engineer moves through the overall sensemaking process, the person must come to understand particular pieces of information about the program. For instance, to understand how the program uses the system interface, the goals of the person involve discovering information about how the system interface is used. This information can be in the form of system calls that the program uses or sequences of system calls. As a person accumulates information about the system interface, it may lead to information about sequences of instructions that are important, general properties of the program, and so on.

The reverse engineer brings background knowledge to the process, which enables items in the task environment to be recognized as relevant and interpreted. The reverse engineer updates knowledge with relevant information and performs actions in the task environment, both to change the state of the program and to gather more information to meet one or more of the other information requirements of the task.

As the reverse engineer acquires new information, he or she updates knowledge about the structure, mechanisms, and behaviors of the program in a mental model of the program. The reverse engineer's mental model of the program contains elements that are related to background knowledge (like programming patterns, assembly language knowledge, and so on), and elements that are related to the current situation (such as the sequence of function calls which were just executed). These items are represented mentally by the reverse engineer and are used to construct a model of the overall "picture" of the program, which specifies the program in terms of relevant mechanisms, behavioral properties, and structural properties.

The sensemaking process involves setting information goals, generating a plan to achieve those goals through actions, carrying out the actions of the plan, sensing information, interpreting relevant information, updating knowledge, and generating hypotheses from the updated knowledge. The theory makes the claim that the hypotheses that drive sensemaking are generated from deductions caused by integrating conceptual information with prior knowledge. These hypotheses create opportunities for the reverse engineer to seek information to verify some piece of stored knowledge about the program.

The process is not necessarily linear, in that a reverse engineer continually moves back and forth between different sub-goals, making sense of these different elements and gaining information in the process. Nevertheless, as more information accumulates about properties of the program, properties of functions, sequences of instructions, system interactions, and how the program uses data, the person is developing a mental model of the program.

Various conceptual components of knowledge feed into sensemaking and enable a person to make sense of some aspect of the program. For instance, in order to understand what sequences of instructions are important in the program, a reverse engineer has to have top-down conceptual knowledge about the behaviors that programs perform, the behaviors that the target program is likely to perform, and how these behaviors can be represented in assembly language.

During the process, the person is performing a macro-level sensemaking process, where the goal is not to understand the individual elements of the program, but instead to understand what the subject matter experts interviewed in Chapter 5 described as the "complete picture of the program," or the mental model of the executable program. This "complete picture" or mental model of the program includes representations of the goals of the task, plans of action, program properties, system calls used, sequences of calls, important instruction groups, the behaviors of functions, how functions work together, and how data flows through a program.

7.3 Implications of the Research

This dissertation has made an essential step toward bridging the representational gap between assembly language representations of programs and the way that reverse engineers think about executable programs in reverse engineering tasks. The results of this research can immediately be used to inform design considerations of reverse engineering tools that take advantage of how people process information in reverse engineering tasks. Understanding the different types of goals and concepts that people pay attention to while reverse engineering is helpful in that representations can be tailored to integrate information goals and documentation of a program into the task.

The results of this research can also be used to improve the education and training of reverse engineers to perform in Air Force cyberspace operator roles. The domains of knowledge requirements described in Chapter 5 can be further decomposed to develop curricula and training aids for helping personnel learn what concepts and procedures they need to know in order to quickly make sense of what a program is doing and how the program is structured. The taxonomy of goals discussed in Chapter 5 can be modified into samples of behavior which can enable one to demonstrate a reverse engineer's performance in a program understanding task.

Another implication of the research includes one of the roles for which this research was directly intended: as a first step to computationally modeling reverse engineer's comprehension of programs from executable representations. Modeling cognition in reverse engineering was not previously possible because there was no indication as to the goals, concepts, state representations, or mental representations which were involved with reverse engineering tasks. This dissertation explored each of these areas through empirical investigation, and now this cognitive modeling work can take place.

7.4 Areas for Future Research

There are a number of areas which have been identified as needing future research effort in order to progress. Areas where future research is needed fall into three categories:

1. Improving capability in reverse engineering executable programs,
2. Further researching the theory and applications of sensemaking, and
3. Improving capabilities in knowledge elicitation and qualitative data analysis

7.4.1 Improving Capability of Reverse Engineering Tools. A number of areas are ripe for research for those intending to develop expertise in reverse engineering, malware detection, and vulnerability analysis. First, is using the sensemaking process to impelement reverse engineering tools that allow reverse engineers to make sense of executable programs more quickly. There are theoretical and technical challenges which underlie each of the cognitive enhancements for tools, such as defining representation commitments and definitions of data standards for disassemblers and debuggers, efficient constraint-based search, and managing the additional data required to support these aspects of the tools with as small a memory footprint as possible.

A second area for reverse engineering tool development is involved in incorporating conceptual and procedural knowledge into reverse engineering tools. Many knowledge domains can be expressed through description logic formalisms. Developing reverse engineering tools which can take advantage of automated inference techniques could help reverse engineers better sift through and sort through the complex information space provided by reverse engineering tools.

A third area for improving reverse engineering capability is through developing and testing training curricula based on the organization of knowledge presented in this dissertation. As organizations continue to put more trust into automated information systems to support their critical missions, criminals and adversaries will use the inevitable weaknesses in software to exploit organizational dependencies and lack of understanding of low-level technologies. Better trained Air Force personnel working in cyberspace operations would have the opportunity to be at the leading edge of the problem in discovering vulnerabilities and threats, instead of continually being at the lagging edge with organizations that find out about their system vulnerabilities from outside security researchers, or after attacks on their systems.

More research needs to be done in mapping high-level program behaviors and features of individual instructions. More work is needed in developing and integrating semantic web technologies into disassembly-based debuggers to take advantage of the inference mechanisms they provide for higher-level reasoning. Various representations discussed need to be worked out, developed, and tested with human users to optimize reasoning and performance in the desired tasks. Additionally, there is still further research needed in the basic questions of how hypotheses are generated and selected and how they should be represented. More work is needed in employing computational approaches to seeking information based on hypotheses, testing the information, and integrating the results of that process into a system knowledge base. Finally, more work is needed in developing these tools so their reasoning is efficient and provides a useful engineering advantage to those that analyze code from assembly language.

7.4.2 Further the Theory and Application of Sensemaking.

Research work in developing a general computational theory of sensemaking is required. This can be performed through the development of computational process models to describe the cognitive activities outlined in this dissertation, as well as describing them at a higher level of abstraction. Computational cognitive models expressed within a cognitive architecture make commitments to the representation of human cognition in such a way that they can be used to predict errors in training applications or cognitive tutors, perform as intelligent teammates or decision aides, and allow testing of theoretical propositions in cognitive science.

Theoretical work is required in developing a smaller set of sensemaking tasks which can be used to develop the aspects of sensemaking theory through experimental trials with greater numbers of untrained participants (for example, college students). Experimental work is required in validating each of the steps in the process and modeling work is in order to understand how people perform each of the cognitive behaviors involved in the sensemaking process.

Additionally, to provide value in studying sensemaking, work is also needed in developing more realistic synthetic task environments. Better task environments can provide ways to study sensemaking in many different task areas. They can also allow data collection from elements of the sensemaking process in other realistic environments, but which still capture the important aspects of moving through a complex data space where computational algorithms are often not able to perform well without incorporating heuristics..

Each of the phases in the sensemaking theory described in this dissertation should be investigated rigorously to verify their applicability to broader populations and to other sensemaking problems in different domains. Additionally, more work is needed in casting the sensemaking problem in mathematical or computational formalisms so that more precise hypotheses can be generated. Finally, the theories of sensemaking should be tested in other areas where sensemaking and analysis are believed to be involved to develop training and technologies that improve human performance in those tasks.

7.4.3 Improving Methodologies in Knowledge Engineering. Researching aspects of cognition presents its own set of challenges in that direct data is not available and inferences must be made throughout. These challenges are compounded by the lack of suitable knowledge representation formalisms at the appropriate level of abstraction to express the results of the research.

One of the major challenges in performing the dissertation research was that knowledge representation formalisms were either too detailed to be tractable, or too broad to be meaningful. When researchers must represent the complex cognitive work of many participants, especially those that are communicating with each other, the task could become overwhelming for any researcher. More methodological work and support engineering is needed in tools, techniques, and technologies which can facilitate knowledge engineering work at a large scale.

Better representation formalisms are needed to express the separate areas of knowledge, tasks, task environments which apply to many domains. These areas are intermingled in many formalisms, which makes it difficult to express complex functionality, reuse models, or grow the scale, robustness, and capabilities of automated or semi-automated knowledge-based work.

There are yet many areas uncovered in understanding how reverse engineers make sense of programs. As modest as are the achievements of the study, the quest for understanding how people solve real-world problems is a small step to incrementally advancing humanity's understanding of the unexplored aspects of the intelligence, knowledge, understanding, and consciousness.

Appendix A. Request for Exemption from Human Experimentation Requirements

14 February 2010
MEMORANDUM FOR AFIT/ENG
AFIT/ENR
AFRL/Wright Site IRB
IN TURN

FROM: AFIT/ENG/DCS-10M

SUBJECT: Request for exemption from human experimentation requirements (32 CFR 219, DoDD 3216.2 and AFI 40-402) for Cognitive Task Analysis of Software Reverse Engineering

1. The purpose of this study is to determine the cognitive information processes underlying how problem solvers construct and reason with abstractions during software reverse engineering. This study should shed light on what guides people in decisions on how to reduce information to make reasoning in complex problems tractable. The research findings should be applicable to many difficult problems in cyber security and decision making. The results are intended to be published in peer-reviewed scholarly journals as well as in the doctoral dissertation.

2. This request is based on the Code of Federal Regulations, title 32, part 219, section 101, paragraph (b) (2) Research activities that involve the use of educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures, or observation of public behavior unless: (i) Information obtained is recorded in such a manner that human subjects can be identified, directly or through identifiers linked to the subjects; and (ii) Any disclosure of the human subjects' responses outside the research could reasonably place the subjects at risk of criminal or civil liability or be damaging to the subjects' financial standing, employability, or reputation.

3. The following information is provided to show cause for such an exemption:

3.1. Equipment and facilities: The equipment and facilities used for the study will be a personal computer running VMWare Workstation, Windows XP operating system, IDA Pro disassembler, OllyDbg debugger, and HEdit hex editor software. The research will also involve taking video captures of the computer screen, audio recordings of a 'think aloud' protocol, while a subject reverse engineers computer software. Audio recordings of the subject's problem solving will be taken, and sketches and handwritten data (boxes and pictures) will be retained for analysis by the researcher. The interviews will take place over the telephone or in person at the subject matter experts' workplace. The task analysis will take place at the Cyber Trust Lab on the third floor of building 620 at the Air Force Research Laboratory's Sensors Directorate at Wright-Patterson AFB, OH.

3.2. Subjects: There are three phases to this research: 1.) the knowledge elicitation phase, 2.) the task analysis phase and 3.) the knowledge representation phase. The concept mapping phase will consist of up to five subject matter expert volunteers solicited by the researcher based on their amount of experience solving problems in reverse engineering software. A typical subject matter expert will have five to seven years of hands-on experience reverse engineering and/or programming in assembly language.

The task analysis phase will include up to 25 volunteer subjects from the population of people able to solve software reverse engineering problems. The population will include a mix of DoD contractors, DoD civilians, and active duty military, as well as potentially personnel with no DoD affiliation (from expected greatest to least numbers). Many of the reverse engineers will likely be DoD contractors since the DoD performs a lot of contracted work in software reverse engineering.

For each category, appropriate measures will be taken to ensure that the participants are able to participate in the study and that their designation does not preclude them from participating (i.e., that there are no contractual or pending contractual relationships that would create an implied pressure to participate). Also measures will be taken to work with limits on their participation based on their designation (i.e., that some civilians might only participate during non-duty time, or that participation is not commander-directed for active duty, etc.).

Participants will be included by self-identifying their ability to reverse engineer software in x86 assembly code. Additionally, lacking the ability to reverse engineer using IDA Pro and OllyDbg will be used as criteria to exclude them from the study. Factors such as age, sex, race, organizational affiliation, or job designation outside of this criteria will not be used to exclude or include candidates. The knowledge representation phase will involve interviews with the subjects and subject matter experts to clarify and provide additional information about data collected during the first two phases.

3.3. Timeframe: The overall study (not including data analysis) will be conducted over the period of three months, or until 25 subjects are collected, whichever occurs sooner. The time for each subject matter expert participant of the concept mapping phase will be approximately two hours during one scheduled interview. The time period for the task analysis participant is expected to be approximately 30 minutes to three hours, depending on the problem solving skill of the participant. After three hours, the task analysis session will be terminated. The time period for an individual knowledge representation session is expected to be 30 minutes to one hour.

3.4. Data collected: Demographic data will be collected only to outline potential sources of bias in the research during the analysis of the data. Such information will be limited to educational background and experience reverse engineering. Contact information such as name, e-mail address, and phone number will be recorded in order to gather follow-up information about data from the sessions. The contact data will be separated from the data from the interviews and sessions once the analysis is completed.

Research data will include audio recordings of each participant's voice as he or she thinks aloud through the problem solving portion of the task, and a video recording of the participant's screen. A sample questionnaire is attached.

3.5. Risks to Subjects: The risk to participants includes the potential accidental release of collected data about their background in reverse engineering. Despite measures taken to protect the subjects' identity, there is the possibility that the collected demographic information can be correlated to uniquely identify a participant based on the sound of their voice. The data should not be able to provide any other purpose besides establishing a person's skill at solving a reverse engineering problem. If the participant inadvertently releases personally identifiable data during the recorded task analysis, it will be sanitized by the researcher.

3.6. Informed consent: All subjects will be self-selected to volunteer to participate in the interview. No adverse action will be taken against those who choose not to participate. Subjects will also be made aware of the nature and purpose of the research, sponsors of the research, and disposition of the results. A copy of the Privacy Act Statement of 1974 will be available for their review.

4. If you have any questions about this request, please contact Mr. Adam Bryant (primary investigator).

//SIGNED//
ROBERT F. MILLS, Ph.D., Associate Professor
Faculty Advisor, AFIT/ENG

//SIGNED//
ADAM R. BRYANT, Civilian
Graduate Student, AFIT/ENG

Attachments:
1. Research description
2. Structured interview outline
3. Task analysis instructions

Appendix B. Structured Interview

Study: Understanding Conceptual and Procedural Knowledge in a Reverse Engineering Task

Investigator: Adam Bryant, AFIT/ENG and AFRL/RYWA
Purpose:
This interview is designed to help better understand the knowledge used in reverse engineering tasks. For this reason, several questions have been designed to guide this study and to help elicit the concepts and procedures used in practice. This information may be useful to designing better reverse engineering tools, and in understanding the process of "sensemaking" when a person interprets and characterizes ambiguous data in reverse engineering.

The results of this study should shed light on what guides people in decisions on how to reduce information. The study is sponsored by the Air Force Research Laboratory's Robust Decision Making Strategic Technology Team and is part of a student's program of doctoral work at the Air Force Institute of Technology's Department of Electrical and Computer Engineering.

Instructions:
This interview will be recorded and is expected to take between one and two hours.

Questions:

Overview and process:
1. Provide an overview of some of the different types of reverse engineering tasks you perform.
2. Can you identify five authentic problems that an expert should be able to solve if they have become a master at reverse engineering?
3. Can you break down the reverse engineering task to between 5 and 7 major steps that you perform in most reverse engineering tasks?
4. What steps tend to vary between the different types of reverse engineering tasks?

Sub-goals:
5. What do you consider your main goals to be when you're reverse engineering?
6. How do you approach carrying out a plan to achieve your goals?
7. If you have to change your approach mid-task, what signals to you that it's time for a different approach?
8. How are your goals in a reverse engineering task different from when you first got started reverse engineering?

Critical decisions and cues:
9. Describe three to five difficult decisions you've had to make in the course of reversing software.

10. When do those types of decisions typically appear, or what are the decision cues?
11. How did you decide what the different decision options were?
12. How did you choose between the different options?

Specialized knowledge:
13. Describe a particularly difficult or complex reverse engineering task you've performed.
14. What aspects made it difficult?
15. What conceptual knowledge did you need in order to be able to tackle it?
16. What are the skills and abilities that you think separate experts from novices?
17. What conceptual knowledge do you think experts have that novices don't have?

Specialized tools and equipment:
18. Describe the standard and specialized tools you use in the performance of reverse engineering, and what abilities they provide?
19. Describe your reverse engineering setup.
20. What are the must-have tools and equipment?
21. What must-have information do the tools provide?
22. What capabilities do these tools and equipment allow you to have?

Automaticity and implicit procedural knowledge:
23. What types of decisions or steps in reverse engineering have become automatic for you?
24. If you explained the steps to complete a problem to a new reverse engineer, and the novice had to complete the task based solely on your instructions, where would he or she likely get hung up?

Appendix C. Task Analysis Instructions

Title: "Understanding Conceptual and Procedural Knowledge in a Reverse Engineering Task"

Investigator: Adam Bryant, AFIT/ENG and AFRL/RYWA

Purpose:
This task analysis is designed to determine the conceptual and procedural knowledge that reverse engineers use in solving a simple reverse engineering problem. This information is useful to designing better reverse engineering tools, and in understanding the processes of sensemaking and information-seeking when a person interprets and characterizes ambiguous data. The results of this study will shed light on how reverse engineers use knowledge to reason about their task environment. The study is sponsored by the Air Force Research Laboratory's Robust Decision Making Strategic Technology Team and is part of a student's program of doctoral work at the Air Force Institute of Technology's Department of Electrical and Computer Engineering.

Time Required:
This task analysis should take between 30 minutes and one hour. If you need a break at any time, please inform the research assistant. If during the task, you feel you are unable to complete the task data about your experience may still be valuable. At any time, if you no longer wish to participate in the task analysis for any reason, please inform the research assistant.

Think-Aloud Protocol:
You should "think aloud" while performing each of the tasks in order to provide empirical data about your thought processes. Cognitive psychology studies suggest that thinking aloud does not interfere with the underlying problem- solving thought processes. Do not attempt to explain your actions to the researcher or assistant, as explaining your behavior does interfere with your thought processes. For this reason, try to simply think aloud while performing the tasks. If you should fall silent, the research assistant will remind you to verbalize while performing the task.

Tasks:
You will be seated at a computer with Windows XP desktop through a virtual machine (unless otherwise arranged). On the desktop and the task bar, there are links to the reverse engineering tools. Several sheets of paper are provided for you to take notes on as you complete the tasks. Since there is variation in the hex editors and configurations used by different reverse engineers, you will be given as much time as you need to familiarize yourself with the tools before starting the tasks. You will be provided a folder with a reverse engineering challenge which you are to complete.

Research Description

This section describes a cognitive task analysis methodology which is adapted to systematically understand intuitive cognitive processes in software reverse engineering. A set of hypotheses are presented to guide the research. Afterwards, empirical work that will test these research hypotheses is laid out into a detailed research plan. Finally, preliminary work in determining how reverse engineers use abstract mental models to solve problems is presented.

Experimental Tasks

The subject will be provided with a binary executable and will be responsible to modify the program's control flow so the program's "Help" menu displays that the program is "registered." The subject will then have to identify the process which controls this and modify instructions and register values in order to redirect the control flow into the desired process. This task represents a simple reverse engineering task that is often one of the first steps in a reverse engineer's training. However, the task requires the reverse engineer to seek information, rely on background knowledge, and act and interact with the task environment.

Coding Experimental Data

A verbal protocol will be performed to capture data from reverse engineers performing reverse engineering tasks. The verbal protocol consists of recording audio and screen captures of reverse engineers working in their own environment (as constraints permit). If the subject does not have or is not able to use their own reverse engineering environment in the task analysis, a working backup setup will be available consisting of a Windows XP operating system in a VMWare Workstation running IDA Pro, OllyDbg, and the HEdit hex editor. If this is the case, the subject will be given time to become familiar with the system setup.

The subject will be instructed to think aloud while reverse engineering while the researcher records their voice and the video capture from the computer monitor. The subject will be instructed only to verbalize, and not to explain their task. Talking aloud in this fashion during task performance has been found to provide useful clues that allow researchers to inference sub-goals and retrieval from short and long-term memory. It has also been found not to significantly interfere with the underlying cognitive processes of many tasks unless the subjects attempt to explain their reasoning while performing the task. Several pieces of paper will also be provided on which the subject may make notes, which will be retained to help integrate the model from the coded empirical data.

From the audio and verbal data, the researcher will analyze each session, analyzing count and frequency of behaviors and concepts that are used. The researcher will qualitatively interpret sequential data to determine higher-level processes and functions from the low-level behavior sequences. The researcher will also infer the sub-task structure from the sequence data. The data from the verbal protocol will be presented back to the interviewee in a post-task interview in order to explain any gaps and verify inferences made about the sub-task structure.

Content Analysis of Verbal Protocol Data

The reverse engineering task will be independently coded by the researcher and by one or more assistants with a background in human factors psychology, computer science, and/or artificial intelligence. The coders will receive advance training from the researcher in how to properly code human factors studies. The verbal dialog from the task will be coded to identify concepts, goals and sub-goals, hypotheses, and data seeking behaviors. In coding the audio the coders will monitor for times where the subject names references related to one of the following:

- Concepts
- Goals and sub-goals
- Plans
- Hypotheses
- Information seeking

Whenever the coders detect a reference to any of the above, they will write down the name in the applicable category column. For each time the same concept is heard after the initial annotation, a tally will be used to indicate the number of times a concept is invoked. For detecting concepts, the coders will annotate each time a concept is used in reasoning by listing those objects or entities verbalized as nouns in reference to and during the task. Ambiguous references such as “this” or “right here” will be given a temporary placeholder name and later correlated with video to infer the referred concept.

Goals and plans will be coded by listening for when a subject expresses a desired state. For instance, when a subject says something like “Now I need to ... ,” “I’m supposed to ...” or something of the sort, they are expressing a goal. Goals will be annotated whenever they are explicitly specified, or when they are alluded to as an effect, such as when the subject describes changing a register value in order to change control flow. In the previous case, changing control flow would be an effect and the register values represent a state variable. When a goal is recognized, the coder can create a text label such as “find window function” and annotate the time. It will be up to the analysis to integrate the labels from different coders based on the time stamps. It will also rely on analysis to describe the goals in terms of problem state attributes.

Hypotheses will be coded by listening for questions or statements that indicate a guess, such as “Maybe,” “What if,” “It’s possible that,” etc. Each of these hypotheses will be coded by providing a label for the hypothesis and annotating the time.

Information-seeking behaviors will be coded by annotating when the subject mentions the needed data as part of a question, as in "I need to find when this memory address changes," or "Looking for the first time the code calls this function." It may also be indicated by descriptions of moving through the spatial environment that the tools create, such as "Going down to this part," "Moving over here to this window," or "Scrolling back up to see where this value came from."

Data Analysis

After coding, an integration phase will take place for each subject's tasks. The tasks will be correlated on concepts, goals, plans, hypotheses, and information-seeking behaviors. The data will be analyzed in order to describe and classify the different types of conceptual components. The concepts will be separated into concepts representing things in the environment and concepts representing functions (usually expressed as verbs). The coded concept data will be analyzed in order to track similarities and differences in conceptualization of structure and function between subjects and between reverse engineering problems for individual subjects. The researcher will describe the level of abstraction of each concept and will seek to infer observable attributes that enabled recognition of a concrete instance of a concept.

Goals, sub-goals, and plans will be characterized based on state variables which represent the goal and the plans made to achieve the goal. The researcher will seek to infer the state which represents completion of the goal, and the state variables that the subject uses to track progress toward completion of the goal. The coded audio, and video data will be integrated to determine and characterize the overall reasoning strategy for each of the subjects and for each subject in each of the reverse engineering problems. Similarities and differences will be analyzed in reasoning strategies, clarity and number of goals and sub-goals, and structure and relationships of goals and sub-goals.

The sequential data will be analyzed to characterize the problem solver's representations of state and state variables for each reverse engineering problem. Like the concept data, it will be compared between subjects and between problems for an individual subject. Since humans often solve problems in dynamic environments, the data will be interpreted to describe how state is dynamically formed and used in reasoning. Static (or always available) state variables will also be characterized and described. The data between subjects and between reverse engineering problems for an individual subject will be analyzed to compare similarities and differences in how state and state variables are used in problem solving in the reverse engineering domain.

Hypotheses and information-seeking behaviors will be characterized by the type of information sought, and the type of hypothesis. Since in the sense making model, information-seeking behaviors occur when a subject is trying to establish or restructure a "frame" or hypotheses, these behaviors will be matched to hypotheses through inference. Information seeking and hypothesis data will be used to infer the structure of the mental models and describe problem-detection activities of the reverse engineers.

Knowledge Representation

The knowledge representation phase is aimed at capturing and depicting the knowledge in a representation formalism, which can take on either a general description or a formal description such as first-order logic, production rules, or a meta-model. The goal of knowledge representation is to provide a description of the reverse engineer's knowledge used in the reverse engineering task and to verify this knowledge with the reverse engineer and possibly the SMEs.

The fully-fleshed out knowledge representation should relate the concepts to one another and express constraints between the relationships. The representation of the procedural knowledge should be able to express the processes in terms of concepts and information requirements from the reverse engineering task. It may take the form of production rules or first-order logic statements, but other representation possibilities will be investigated. The specificity of the knowledge representation will depend on the specificity and quality of the knowledge extracted and interpreted from the interviews and verbal protocol.

```
################################################################
# Script to analyze coded subject matter expert interview data
# @author Adam Bryant
# @email adam.bryant@wpafb.af.mil
# @date 15 Dec 2011
################################################################
import sys
from numpy import *
import operator

class TextAnalyzer:
    def __init__(self):
        pass

    def analyzeFile(self):
        linelist = []
        #put strings in sublists
        f = open(r’interview1.csv’)
        for line in f.readlines():
            linelist.append(line.rsplit(’,’))
        concepts = {}
        for line in linelist:
        # flip it so phrases are first
            line.reverse()
            # chop the first item out (phrases)
            line = line[1:]
            for word in line:
                if word in concepts.keys():
                    concepts[word] += 1
                else:
                    concepts[word] = 1
        print sorted(concepts.items(),
                key=operator.itemgetter(1),
                reverse=True)
        for line in linelist:
            print line
        print len(concepts)

def main():
    ta = TextAnalyzer()
    ta.analyzeFile()

if __name__ == "__main__":
    sys.exit(main())



###############################################################
# Script to count words and occurrences from case study data
# @author Adam Bryant
# @email adam.bryant@wpafb.af.mil
# @date 15 Dec 2011
###############################################################
#!/usr/bin/env python

import sys  # to exit the program:  sys.exit(0)



def main():
    try:  # open the files for writing and reading
        inText = open("input.csv", "r")
        outText = open("output.txt", "w")
        raw = inText.read()
    except IOError:
        print 'Cannot open file %s for reading' % inText
        sys.exit(0)
    dataList = raw.split("\n")  # break into lines


    strippedDict = dict()
    wordDict = dict()
    entryDict = dict()
    for line in dataList:       # put lines into list
        category = line.rstrip(",") # pull comma off the end
        word = category.split(",")[0]
        category = category.split(",")[1] # break into two
        strippedDict[category] = strippedDict.get(category, 0) + 1
        wordDict[word] = wordDict.get(word, 0) + 1
        entryDict[line] = entryDict.get(line, 0) + 1

    tempList = strippedDict.items()
    tempList2 = wordDict.items()
    tempList3 = entryDict.items()
    sortedList = sorted(tempList, key=lambda x: x[1],
        reverse = True)
    sortedWords = sorted(tempList2, key = lambda x: x[1],
        reverse = True)
    sortedEntry = sorted(tempList3, key = lambda x: x[1],
        reverse = True)

    ##### Write the words from the list to an output file
    try:
        outText.write("Frequency of Concept Categories:
            (total categories:")
        outText.write(str(len(sortedList)))
        outText.write(")\n")
        for entry in sortedList:
            outText.write(str(entry))
            outText.write("\n")
        for i in range(0, 10):
            outText.write("\n")
        outText.write("Frequency of Concepts: (total concepts:")
        outText.write(str(len(sortedWords)))
        outText.write(")\n")
        for entry in sortedWords:
            outText.write(str(entry))
            outText.write("\n")
        for i in range(0, 10):
            outText.write("\n")
        outText.write("Frequency of Concept, Category Pairs:
            (total pairs:")
        outText.write(str(len(sortedEntry)))
        outText.write(")\n")
        for entry in sortedEntry:
            outText.write(str(entry))
            outText.write("\n")
        print "completed writing to output file"
    except IOError:
        print "Cannot write to file %s" % outText
    outText.close()

if __name__ == ’__main__’:
    main()
```

	Segment	**D**	**G**	**S**	**H**	**I**
1	okay	1				
2	let's do Splish.exe		1			
3	it's loaded into memory			1		
4	these arguments are nice	1				
5	this is the new OllyDbg			1		
6	okay	1				
7	I'm going to...	1				
8	what am I doing?		1			
9	let's just play it and see what happens		1			
10	f9		1			
11	splash screen			1		
12	Splish splash			1		
13	check hardcoded			1		
14	name and serial check			1		
15	enter serial number			1		
16	that closes it			1		
17	your mission is to disable the splash screen			1		
18	name and hardcoded serial			1		
19	and keygen the name serial part			1		
20	see what's going on the program			1		
21	three very basic and easy protections			1		
22	see you in the next level			1		
23	Crudd			1		
24	okay so	1				
25	check hardcoded			1		
26	sorry please try again			1		
27	okay so	1				1
28	it doesn't tab over				1	
29	test me	1				
30	sorry please try again			1		

	Segment	**D**	**G**	**S**	**H**	**I**
31	alright	1				
32	sooooo...	1				
33	let's close this and back up to the very start		1			
34	and see what it's doing					1
35	start stepping in f9		1			1
36	that 's the start of the program			1		1
37	that's an intermodular call			1		1
38	follow that		1			
39	just a little jump			1		
40	system metrics			1		1
41	intermodular call			1		1
42	okay	1				
43	wonder what that pushes in				1	
44	pushes the base pointer on the stack					1
45	call			1		
46	it's another one			1		
47	function			1		
48	leave			1		
49	return			1		
50	push			1		
51	call			1		
52	jump here			1		
53	splash class			1		
54	bitmap			1		
55	it's pushing something in memory				1	
56	it's gonna have a picture				1	
57	in the resources somewhere				1	
58	a pattern brush			1		
59	it's gonna paint the bitmap				1	
60	pattern brush			1		
61	window's next				1	
62	a window			1		
63	upppp			1		
64	what happened?			1		
65	that must be a call here				1	
66	401546			1		
67	a call to show window			1		
68	yeah of course	1				
69	f9 to get there again		1			
70	so that's a jump to it				1	

	Segment	D	G	S	H	I
71	I need to patch that jump somehow		1			
72	show window			1		
73	I need to have that serious		1			
74	it's gonna tick once				1	
75	it's telling me SendMessageA			1		
76	I guess that's the message				1	
77		1				
78	I could jump around it completely		1			
79	I cold just patch a jump right here		1			
80	401... 401583					1
81	and then just miss all this stuff		1			
82	I could make the tick count zero that's really fast		1			
83	I could just nop the whole thing		1			
84	that's really messy I wonder if it would screw up the rest of the code		1			
85	I'll just jump it		1			
86	it's really quick					
87	jmp			1		
88	401588			1		
89	should be 74 or 75				1	
90	upp 48.			1		
91	there we go	1				
92	let's rewind it		1			
93	and play		1			
94	oop my stuff isn't there anymore			1		
95	must not have saved				1	
96	loaded from the backup				1	
97	do that again		1			
98	set a breakpoint here		1			
99	upp..			1		
100	the breakpoints stay but the other crap doesn't			1		
101	set a breakpoint here		1			
102	let's reload it from memory		1			
103	jump again		1			
104	401583			1		
105	pshhhw	1				
106	let's f8 it		1			
107	bong	1				
108	okay	1				
109	should just be able to		1			
110	OurWindow			1		

	Segment	D	G	S	H	I
111	looks like my first win32 program			1		
112	CreateWindowExA			1		
113	that must be the other				1	
114	ShowWindow				1	
115	okay this is the message loop				1	
116	all that code in win32 just does that				1	
117	it's pretty sad	1				
118	there we go	1				
119	there's the message loop					1
120	so to get out of the message it's 401					1
121	I'm gonna remember this		1			
122	translate message			1		
123	get message			1		
124	gets the message 401544				1	
125	(type)	1				
126	window			1		
127	save as stuffs			1		
128	okay	1				
129	so	1				
130	(scroll up)	1				
131	if I rewind it it's gonna break that				1	
132	so let's patch that jump again		1			
133	(patch that jump)	1				
134	lets do this in a hex editor		1			
135	HexD			1		
136	so I've got to find it first		1			
137	open it up		1			
138	desktop		1	1		
139	crackmes		1	1		
140	Splish		1	1		
141	can't search for address				1	
142	search for this string		1			1
143	control f to find that string		1			1
144	not a string it's a hex value			1		
145	a31132400			1		
146	alright looks like it	1				
147	right after that it's four things then				1	
148	ff35113240			1		
149	so these are the ones I need to change right here				1	
150	so changing this				1	

	Segment	D	G	S	H	I
151	write access			1		
152	up			1		
153	so	1				
154	it's still open in memory so it must have a memory lock or something				1	
155	close this		1			
156	now let's try	1				
157	aw crap			1		
158	(err)			1		
159	now I got to close this one first and I can close the other one		1			
160	OllyDbg		1	1		
161	alright open		1			
162	where is it					1
163	I don't need to open that one				1	
164	close that one		1			
165	close yes		1	1		
166	close the process		1			
167	Splish splash			1		
168	now in the other one		1			
169	open		1	1		
170	Splish			1		
171	copy this out just in case		1			
172	(type)	1				
173	then I'll run a patch after that		1			
174	like e4 or something		1			
175	close this cos I'm stupid and I forgot		1			
176	Splish		1	1		
177	debug		1	1		
178	open		1	1		
179	f7 to it		1			
180	don't want to f7 through there				1	
181	just intermods			1		1
182	until I see			1		1
183	right here it is			1		1
184	oh I had the breakpoint set still				1	
185	CreateWindow			1		
186	so you push 1			1		
187	I wanna change the push 1 to jump		1			
188	(type)	1				
189	I guess I can do this by bytes why it was only two bytes earlier				1	
190	just a short jump			1		

	Segment	D	G	S	H	I
191	401583			1		
192	so	1				
193	jump short jmp 43			1		
194	so it's like 43 bytes or something				1	
195	eb 43			1		
196	change to eb 43		1			
197	um	1				
198	what am I doing		1			
199	this	1				
200	close that up		1			
201	open my handy dandy hex editor		1			
202	Splishy Splishy	1				
203	control f		1			1
204	did it save it?					1
205	good thing I saved it cause I didn't remember				1	
206	401066			1		
207	eb 43			1		
208	save that		1			
209	save it under a different name		1			
210	because that would be bad		1			
211	splsih no splash			1		
212	uh try this		1			
213	yep	1				
214	runs with no splash screen			1		
215	find hardcoded serial and keygen name serial part			1		
216	find serial		1			
217	serial number is...					1
218	let's open the debugger		1			
219	open says me	1				
220	Splish Splish splash	1				
221	I got to find the		1			
222	401544			1		
223	so it's control b		1			
224	401445 is that what I want		1			1
225	um close		1			
226	40154 so this is the get message window			1	1	
227	(scroll)	1				
228	so there's a serial number there somewhere				1	
229	the serial number's actually gonna be in memory somewhere				1	
230	unless it's actually constructed dynamically				1	

	Segment	D	G	S	H	I
231	so that's the goodboy message			1	1	
232	403043			1		
233	so lets find a reference to that					1
234	that 403143			1		1
235	nope			1		1
236	up	1				
237	(scroll)					1
238	there's a strings thing somewhere				1	1
239	you can see all the breakpoints				1	
240	ump	1				
241	(look for)					1
242	trace			1		
243	step into?			1		
244	mmm	1				
245	help about			1		
246	there used to be a thing where you could see the text strings				1	
247	I haven't done that in so long	1				
248	this is all the resource stuff			1	1	
249	(look at memory)					1
250	let's find the thing in the window	1				
251	ah let's do it	1				
252	set a breakpoint here	1				
253	clear this other one	1				
254	so it's waiting for a message				1	
255	I should leave that out for when it translates ti	1				
256	I want when it gets the message		1			
257	what's it do wth the message					1
258	I don't even know	1				
259	so if iet has	1				
260	or eax eax			1		
261	so if it has any message at all it's gonna break out here				1	
262	it is hardcoded			1		
263	nasty			1		
264	check hardcoded			1		
265	oh			1		
266	that didn't go where I wanted to go			1		
267	get message			1		
268	it didn't jump out of there at all			1		
269	um	1				
270	so lets breakpoint somewhere up here		1			

	Segment	**D**	**G**	**S**	**H**	**I**
271	okay f8		1			
272	now I need to		1			
273	I need to do over here		1			
274	let's play it again		1			
275	nasty			1		
276	check hardcoded is gonna jump right			1		
277	um so where is it in memory					1
278	I don't want to search for strings cause it's stupid		1			
279	where's that freaking message go					1
280	does it hit this one at all					1
281	let's try it see if I can		1			
282	upm	1				
283	it does hit it			1		
284	see if it's equal					1
285	it's not equal			1		
286	so this one will keep going				1	
287	jump			1		
288	short			1		
289	so if I put something in eax like nothing				1	
290	it wil jump out				1	
291	returns			1		
292	op..	1				
293	that's not good			1		
294	that just killed it			1		
295	let's just do this without killing it		1			
296	I'm dumb	1				
297	alright	1				
298	I need to go to		1			
299	translate message			1		
300	I need to let it call this		1			
301	so now it's doing a nothing message			1		
302	I need to jump in here		1			
303	f7		1			
304	dispatch message			1		
305	alright so here's the goodboy message				1	
306	I just happened to see it			1		
307	good job now keygen it			1		
308	back to this function		1			
309	to get here		1			
310	to get the goodboy it needs to get to this jump				1	

	Segment	**D**	**G**	**S**	**H**	**I**
311	so this comparison needs to work out				1	
312	403643			1		
313	so it’s a loop so more in that				1	
314	follow in the dump			1		
315	more address			1		
316	mmm	1				
317	memory address			1		
318	I’ll set a breakpoint		1			
319	rewind it		1			
320	play it up		1			
321	hardcode		1			
322	nasty nasty		1			
323	check					1
324	okay so that’s not where I thought it was			1		
325	it didn’t even stop on that breakpoint			1		
326	I wonder if it’s a fake umm				1	
327	name and serial number				1	
328	so here’s the message box a			1		
329	jump short			1		
330	no					1
331	so I need to get the name and serial		1			
332	pass that one		1			
333	create window create window			1		
334	should see a GetDialogItemText or something			1		
335	translate message			1		
336	there’s a resource thing			1		
337	it’s a small program so I can just like look through it				1	
338	window text			1		
339	Splish splash			1		
340	please enter your name			1		
341	so			1		
342	so that’s not it			1		
343	say sorry please try again			1		
344	wish I could do that and then back it up		1			
345	so these are two separate bad boy messages				1	
346	alright I found the serial first			1		
347	so there’s the stuff that I had			1		
348	scroll through memory		1			1
349	do I see anything?					1
350	well	1				

	Segment	D	G	S	H	I
351	find references			1		
352	I forgot about that			1		
353	let's find references to this		1			
354	push offset			1		
355	im pushing this memory thing		1			
356	so which one comes first					1
357	okay	1				
358	so try that		1			
359	oh you gotta be kidding me			1		
360	it's just hardcoded			1		
361	I found the hardcoded serial it's just hardcoded			1		
362	I forgot about that			1		
363	these are always so stupid	1				
364	I figured these out and forgot em	1				
365	hardcoded it's a hardcoded serial			1		
366	I need to figure it out		1			
367	there's an algorithm behind this				1	
368	enter name			1		
369	Adam		1			
370	and		1			
371	deadbeef		1			
372	okay so please try again			1		
373	so this is where the comparison is				1	
374	so I'll just catch this		1			
375	lets try that		1			
376	let's back it up		1			
377	this is the badboy message			1	1	1
378	it is at jump is not equal			1	1	
379	there's a comparison with eax and ecx			1	1	
380	I wish I could jump back there		1			
381	that's what I really want		1			
382	if that will take me there		1			
383	if not		1			
384	this is the good boy message			1		
385	if ebx is the same as 403563			1		
386	what is that					1
387	uh	1				
388	references to address constant			1	1	
389	ex 403403			1		
390	ebx will have this point				1	

	Segment	D	G	S	H	I
391	edx will have the thing that's supposed to be in memory				1	
392	how's this thing star					1
393	I need to watch it here		1			
394	cdq			1		
395	idiv here			1		
396	I need to look that up		1			
397	cdq a loop			1		
398	xor			1		
399	add edx			1		
400	compare do and oa			1		
401	so that's 10				1	
402	0a is 10			1	1	
403	so compare da			1		
404	alright I gotta see this going through		1			
405	I can't do that		1			
406	it would be awesome if I could		1			
407	let's do adam and deadbeef again		1			
408	deadbeef		1			
409	it jsut loaded			1		
410	this	1				
411	okay so	1				
412	(27:35)	1				
413	so I just found that 65				1	
414	it's um	1				
415	in memory				1	
416	41 b			1		
417	edx = 0				1	
418	this loop stuff is pretty neet	1				
419	checking to see if it's decimal 10		1			1
420	xor			1		
421	edx and ebx			1		
422	so it's			1		
423	eax equals			1		
424	the thing pointed to be ebx			1		
425	what was that?				1	
426	ecx is 10			1	1	
427	ecx is			1		
428	10			1	1	
429	eax equals			1		
430	mmm	1				

	Segment	**D**	**G**	**S**	**H**	**I**
431	pointer to ebx			1	1	
432	plus esi			1		
433	um	1				
434	gonna tries to divide			1		
435	it's the start of a loop or something				1	
436	alright ecx			1		
437	probably 10			1	1	
438	step over		1			
439	add edx into			1		
440	edx equals edx plus 2			1		
441	so edx equals			1		
442	edx is gonna get ecx				1	
443	edx = remainder of			1		
444	so that will be mod				1	
445	ecx mod ecx			1		
446	I don't know	1				
447	smething like that	1				
448	I know the idea is to divide ecx				1	
449	xor's ecx			1		
450	edx = edx xor fn edx			1		
451	edx and edx			1		
452	that's cancel			1		
453	add edx and 2			1		
454	this is more like lisp	1				
455	edx equals			1		
456	edx plus 2			1		
457	compare			1		
458	if umm	1				
459	edx			1		
460	equals			1		
461	10			1	1	
462	this is			1		
463	jump to this part			1		
464	otherwise	1				
465	if edx low does not equal 10 then				1	
466	edx = edx minus 10			1		
467	so it can equal 10			1		
468	we're gonna put		1			
469	eal into that thing		1			
470	so that thing		1			

	Segment	**D**	**G**	**S**	**H**	**I**
471	10			1		
472	so whatever that thing was		1			
473	okay			1		
474	jump short			1		
475	where was i			1		
476	so it's gonna compare				1	
477	...	1				
478	ebx plus 1				1	
479	eax plus plus				1	
480	it will compare if ebx is equal to				1	
481	that value 40346				1	
482	403			1		
483	...	1				
484	...	1				
485	...	1				
486	4034			1		
487	6.. 3			1		
488	says 4			1		
489	...	1				
490	ebx = 4			1		
491	...	1				
492	so that's the same temp				1	
493	let's call it temp2		1			
494	not foo I hate foo		1			
495	...	1				
496	let's call it a number		1			
497	eax = that number		1			
498	and then		1			
499	jump is not equal		1			
500	so this is one		1			
501	and back to the start		1			
502	actually no it goes back to this		1			
503	ohh	1				
504	eax			1		
505	kay	1				
506	feels like a loop				1	
507	so	1				
508	um once it comes out				1	
509	it looks like do while				1	
510	do		1			

	Segment	**D**	**G**	**S**	**H**	**I**
511	...	1				
512	while		1			
513	ebx =		1			
514	that value		1			
515	okay	1				
516	and then once that fails to be true it will xor these things				1	
517	so ebx = 0			1		
518	ecx = 0			1		
519	edx = 0			1		
520	oops					
521	so what's that again					1
522	this is processing the initial thing				1	
523	so the e			1		
524	plus 10			1		
525	ecx plus 10			1		
526	...	1				
527	...	1				
528	um	1				
529	...	1				
530	...	1				
531	edI =			1		
532	403240			1		
533	403			1		
534	2			1		
535	...	1				
536	40324			1		
537	...	1				
538	som			1		
539	this is d			1		
540	so that's gonna be nothing				1	
541	...	1				
542	pop			1		
543	...	1				
544	yeah that's nothing				1	
545	so increment				1	
546	alright so	1				
547	ecx = 10			1		
548	...	1				
549	...	1				
550	temp = eax = temp			1		

	Segment	**D**	**G**	**S**	**H**	**I**
551	whatever that temp value is			1		
552	cdq			1		
553	doing that thing to memory			1		
554	see how it changes			1		
555	alright it's an 8			1		
556	what could I do to ecx to 10 that would make it have an 8		1			1
557	oh a 6 and an 8			1		
558	so eax got a 6 and edx got an 8				1	
559	what's it divided by?					1
560	maybe eax				1	
561	I don't know what was in eax				1	
562	can I minus back?					1
563	ohhh I can't do that				1	
564	ecx = eax			1		
565	...	1				
566	...	1				
567	eax divided by eax mod ecx			1		
568	then eax = eax divided by ecx			1		
569	would that work out					1
570	so how many times does					1
571	oh crap I don't remember here bef				1	
572	re so play again		1			
573	oh serial check			1		
574	okay			1		
575	I need to stop right there		1			
576	deadbeef		1			
577	I did ecx			1		
578	okay so I had 41			1		
579	41 in eax			1		
580	and a in ecx			1		
581	yeah so that has to be like				1	
582	that's gonna be like				1	
583	eax = eax divided by ecx				1	
584	so ecx was				1	
585	so that's probably right				1	
586	then let's see what ecx was					1
587	eax mod ecx			1		
588	so this is getting better				1	
589	lets go to the next one		1			
590	same thing			1		

	Segment	**D**	**G**	**S**	**H**	**I**
591	f8 f8 f8 f8		1			
592	I’ve got 44 in here			1		
593	that looks important				1	
594	okay			1		
595	I divide			1		
596	step over		1			
597	we’ve got stuff into			1		
598	memory			1		
599	...	1				
600	...	1				
601	ebx increment			1		
602	were progressing through our loop			1		
603	see if ebx is equal tot hat area of memory			1		
604	40367			1		
605	4034			1		
606	6			1		
607	7			1		
608	8			1		
609	so that’s the 8 that got pushed in				1	
610	so ebx = to			1		
611	...	1				
612	I’m an idiot	1				
613	I can’t do this math	1				
614	ebx equals			1		
615	43476			1		
616	so what would that be equivalent to a programming language					1
617	ebx = .			1		
618	forgot about that			1		
619	before that was temp equals			1		
620	ebx			1		
621	do	1				
622	sooo	1				
623	not any check on this time like in the other one				1	
624	temp is					
625	compare ebx to dword pointer			1		
626	if ebx			1		
627	...	1				
628	ebx			1		
629	ebx =			1		
630	403467			1		

	Segment	**D**	**G**	**S**	**H**	**I**
631	is the hardcoded address			1		
632	does not equal			1		
633	...	1				
634	there's a data section				1	
635	it's loaded somewhere				1	
636	it's an offset				1	
637	jump back here			1		
638	eax = whatever			1		
639	doooo	1				
640	...	1				
641	...	1				
642	...	1				
643	let's do while		1			
644	...	1				
645	now this jumps to				1	
646	somethings going on				1	
647	it has to jump out of here				1	
648	okay	1				
649	uh	1				
650	...	1				
651	...	1				
652	...	1				
653	was I looking at the wrong thing					1
654	a zero name?					1
655	...	1				
656	oh this is if I don't have a name			1		
657	...	1				
658	alright	1				
659	to get here				1	
660	we need to get here.				1	
661	and to get here				1	
662	we need to get here				1	
663	these should be equal				1	
664	...	1				
665	xor ebx ebx			1		
666	...	1				
667	compare ebx			1		
668	to 4063			1		
669	so this transforms				1	
670	this use to be the serial				1	

	Segment	D	G	S	H	I
671	this is what's done to the name				1	
672	it seems straightforward to put this in				1	
673	I'm not going to do it because I don't have programming tools installed here		1			
674	so far it makes sense		1			
675	where the 42 and 44 come from				1	
676	I'm not sure				1	
677	that concludes my CamStudio	1				
678	have a great day.	1				
679	(end of transcript)	1				
		123	143	278	122	52

```
Participant,Category,Response
SME4,ANALOGY,Build a picture of the functions and how they manipulate
data
SME4,ANALOGY,Puzzle
SME3,ANALOGY,Puzzle
SME3,ANALOGY,Put together a picture
SME2,ANALOGY,Like a dark art
SME2,ANALOGY,Creative or out of the box thinking
SME1,APPROACH,Approach depends on the domain
SME3,APPROACH,Approach depends on the domain
SME4,APPROACH,Approach depends on the domain
SME1,APPROACH,Approach depends on current assumptions which depend on
domain
SME3,APPROACH,Approach depends on current assumptions which depend on
domain
SME1,APPROACH,Get general properties of the binary
SME3,APPROACH,Get general properties of the binary
SME2,APPROACH,Get general properties of the binary
SME4,APPROACH,Get general properties of the binary
SME1,APPROACH,Some assumptions always hold (PE and import and export
directories)
SME1,APPROACH,Finding how program uses system interface / API
SME3,APPROACH,Find how the program uses the system API
SME3,APPROACH,Construct complete picture of the program
SME3,APPROACH,Determine what program is doing
SME1,APPROACH,Examine the instruction-level information
SME1,APPROACH,Examine the function-level information
SME1,APPROACH,Hooking API calls to abstract sequences of system calls
SME1,APPROACH,Simplifying instruction sequences from obfuscated ones
SME4,APPROACH,With malware Static analysis
SME4,APPROACH,Get everything into IDAPro
SME4,APPROACH,Look at structural issues
SME4,APPROACH,Find the entry point of the program
SME4,APPROACH,Assembly only don’t trust decompiler enough
SME4,APPROACH,Read Man pages or API documentation to see how functions
are made
SME4,APPROACH,Infer and label data type information
SME4,APPROACH,Figure out how data is used in the program in functions
SME4,APPROACH,Abstract assembly into mental picture of C code
SME4,APPROACH,For malware learn how it manipulates files
SME4,APPROACH,For malware how does it communicate on the network
SME4,APPROACH,With malware analyzing program for generating signatures
```

```
SME4,APPROACH,With malware find registry keys used to find on the network
SME4,APPROACH,With malware find files it leaves behind
SME4,APPROACH,With malware understand behavior of program
SME4,APPROACH,Write a decoder to decrypt encrypted traffic to learn what
is exfiltrated
SME4,APPROACH,With protected or malware recover instructions
SME4,APPROACH,With malware (or unknown) static analysis
SME4,APPROACH,Log API and library calls
SME4,APPROACH,Identify obfuscator
SME4,APPROACH,Get around the protection
SME4,APPROACH,Work more quickly if possible
SME4,APPROACH,Recover instructions
SME4,APPROACH,Write tools to defeat protections
SME4,APPROACH,Get past obfuscations as quickly as possible
SME4,APPROACH,For obfuscated understand obfuscator so you can understand
the binary
SME4,APPROACH,Avoid fighting with anti-debugging techniques
SME4,APPROACH,Automate analysis process
SME4,APPROACH,Automate difficult tasks
SME5,APPROACH,Resolve IDA warnings by laying out the different options
SME5,APPROACH,Managing personal knowledge
SME5,APPROACH,Reverse engineering involves troubleshooting
SME5,APPROACH,How much time you have left influences the approaches you
take
SME5,APPROACH,Hypothesis is based on your idea of why you’re looking at
something
SME5,APPROACH,Domain knowledge focuses the approach
SME5,APPROACH,Develop and work from a hypothesis
SME5,APPROACH,Analyze attack surface
SME5,APPROACH,Determine what the program is doing
SME5,APPROACH,Determine how the program is doing what it does
SME5,APPROACH,Depends on the goals
SME5,APPROACH,Automate tasks so they are faster next time
SME5,APPROACH,Watch what malware does
SME5,APPROACH,Automating repetitive tasks
SME2,APROACH,Approach depends on the domain
SME1,CUES,Is program obfuscated
SME1,CUES,Does program have antidebugging
SME1,CUES,If it has antidebugging do I need to be stealthy?
SME1,CUES,Is the program packed?
SME1,CUES,function information includes (win32 API info internal win32
functions symbol information)
SME1,CUES,program output information is from functional information
```

```
SME1,CUES,Program’s use of interface tells you about the behavior of
the program
SME1,CUES,Program’s local code tells you about algorithms in the
program
SME3,CUES,When many things look unusual it might mean the program is
malware
SME4,CUES,Is code in the right places
SME4,CUES,Are sections named by standard compiler naming schemes
SME4,CUES,Does address layout match standard compiler layouts
SME4,CUES,Are entry points in usual places
SME4,CUES,Does binary import enough functions
SME4,CUES,If binary doesn’t import many functions it doesn’t do much or
imports dynamicaly
SME4,CUES,Does IDA produce warnings
SME4,CUES,Does IDA think functions are well formed?
SME4,CUES,Can IDA identify the compiler?
SME4,CUES,If not obfuscated skip boilerplate compiler code step through
from winmain or dllmain
SME4,CUES,If obfuscated step from entry points
SME4,CUES,Network functions it performs tells you how it communicates
SME4,CUES,With malware can it download files
SME4,CUES,With malware can it upload files
SME4,CUES,With malware can it encrypt communications
SME5,CUES,Are there vulnerabilities here that I know how to exploit?
SME5,CUES,When hypothesis changes approach changes
SME5,CUES,With vunerabilities if code doesn’t touch attack surface
ignore it
SME5,CUES,If it’s not trusted consider it malware
SME5,CUES,If malicious don’t run on the network
SME5,CUES,Is the provenance of the software known?
SME5,CUES,Is the company trusted?
SME5,CUES,Is the software trusted?
SME5,CUES,Does the code look malicious in nature?
SME5,DECISION,Determining if a program has a vulnerability
SME5,DECISION,What patches will be easiest to exploit?
SME4,CUES,The type of file informs your choice of tools
SME4,CUES,Does the binary look well formed or not?
SME1,DECISION,Whether to re-write the software or not (time to reverse)
SME1,DECISION,how to approach
SME1,DECISION,In vulnerabilities whether to keep analyzing or choose a
different target
SME1,DOMAIN,What does the malware do?
SME1,DOMAIN,Are there vulnerabilities?
```

```
SME1,DOMAIN,With malware are there backdoors in the software?
SME1,DOMAIN,Vulnerability analysis
SME1,DOMAIN,What program does
SME1,DOMAIN,How program talks to system
SME2,DOMAIN,Hardware reverse engineering software reverse engineering
SME1,DOMAIN,Software protections
SME2,DOMAIN,Software protections
SME3,DOMAIN,Software protections
SME2,DOMAIN,Understanding unprotected applications
SME3,DOMAIN,In software protections understanding packers
SME3,DOMAIN,In software protections understanding anti-exploitation protections
SME3,DOMAIN,In software protections understanding how protections are implemented
SME3,DOMAIN,In unprotected how to interface with binary
SME2,DOMAIN,What does malware do?
SME2,DOMAIN,Are there vulnerabilities?
SME3,DOMAIN,Are there vulnerabilities?
SME2,DOMAIN,With malware are ther backdoors in the software?
SME5,DOMAIN,Web and application vulnerabilities
SME5,DOMAIN,System-level vulnerabilities
SME5,DOMAIN,Penetration testing
SME5,DOMAIN,Exploitation
SME5,DOMAIN,Viruses are not necessarily malware
SME5,DOMAIN,Malware implies intent
SME5,DOMAIN,MIPS
SME5,DOMAIN,ARM
SME5,DOMAIN,Windows
SME5,DOMAIN,Linux
SME5,DOMAIN,x86
SME5,DOMAIN,Hardware
SME5,DOMAIN,Software
SME5,DOMAIN,Firmware
SME4,GOAL,Get a picture of communications taking place
SME4,GOAL,Depends on stakeholder requirements
SME4,GOAL,Formal write up
SME4,GOAL,Signatures
SME4,GOAL,Well-documented IDA database
SME4,GOAL,Reverse engineer quickly
SME5,GOAL,Educate programmers on how not to introduce vulnerabilities
SME5,GOAL,Exploiting a vulnerability
SME5,GOAL,Understanding nuances about a target since they might be useful
```

```
SME5,GOAL,Getting information about a program
SME5,GOAL,Sometimes the goal is getting into a system
SME5,GOAL,Have attack and defensive goal for analysis
SME1,GOAL,Find the purpose
SME1,GOAL,Depends on what domain you are working in
SME1,GOAL,Determine the relevant and important properties you want to
find about the program
SME2,GOAL,Goals depend on the domain you’re working in
SME2,GOAL,Finish the task as quickly as possible (reverse engineering
is expensive)
SME3,GOAL,Determine what stakeholders are interested in
SME3,GOAL,Finish the task as quickly as possible
SME5,GOAL,Determine vulnerabilities
SME5,GOAL,Patch vulnerabilities
SME5,GOAL,Exploit development
SME2,KNOWLEDGE,Assembly language
SME2,KNOWLEDGE,How machine code works
SME1,KNOWLEDGE,Assembly language
SME1,KNOWLEDGE,Machine code
SME1,KNOWLEDGE,Have to know the API like the Win32 system interface
SME1,KNOWLEDGE,Have to know how memory is laid out (in a function and
generally)
SME1,KNOWLEDGE,Domain specific knowledge (Instruction executing I/O
instructions data types how program loader works)
SME1,KNOWLEDGE,how synchronization works in a program
SME1,KNOWLEDGE,how antidebugging tricks work
SME1,KNOWLEDGE,Exception handling is an antidebugging trick
SME2,KNOWLEDGE,understanding how hardware works
SME2,KNOWLEDGE,For protections) includes how protections works
SME2,KNOWLEDGE,For protections) includes how to get around protections
SME2,KNOWLEDGE,for protections) how obfuscations work
SME2,KNOWLEDGE,In vulnerability analysis don’t need to understand
protections
SME2,KNOWLEDGE,With malware understand some level of protections
SME2,KNOWLEDGE,With protections understand antidebugging and packing
SME2,KNOWLEDGE,With malware understand vulnerabilities and exploits
SME4,KNOWLEDGE,How TLS callbacks work
SME4,KNOWLEDGE,API functions
SME4,KNOWLEDGE,Assembly language
SME4,KNOWLEDGE,With malware Windows API calls
SME4,KNOWLEDGE,Network communications and I/O system calls
SME4,KNOWLEDGE,Manual function name resolution
SME4,KNOWLEDGE,Cryptography
```

```
SME4,KNOWLEDGE,Knowing how to track data flow
SME4,KNOWLEDGE,Knowing what IDA warnings mean
SME4,KNOWLEDGE,C++
SME4,KNOWLEDGE,How compilers generate code
SME5,KNOWLEDGE,Knowing how to explain what you are looking for
SME5,KNOWLEDGE,There are problem domain (sys admin embedded etc.)
aspects of knowledge as well
SME5,KNOWLEDGE,Large learning curve with reverse engineering
SME5,KNOWLEDGE,Understanding different vulnerabilitites
SME5,KNOWLEDGE,Knowing how to leverage data flow to exploit a program
SME5,KNOWLEDGE,Knowledge of exploits gained by writing them
SME5,KNOWLEDGE,Knowledge of vulnerabilities gained by implementing them
SME5,KNOWLEDGE,Knowledge is domain specific
SME4,SKILLS,How to reconstruct an import table
SME4,SKILLS,How to annotate analysis work better
SME5,SKILLS,Learning things you wouldn't otherwise care about
SME1,SKILLS,Getting around protections
SME2,SKILLS,Getting around protections
SME3,SKILLS,Getting around protections
SME1,SKILLS,Writing reverse engineering tools (loader disassembler
assembler compiler)
SME2,SKILLS,Writing reverse engineering tools (loader disassembler
assembler compiler)
SME3,SKILLS,Writing reverse engineering tools (loader disassembler
assembler compiler)
SME1,SKILLS,Automating workflow processes
SME2,SKILLS,Automating workflow processes
SME3,SKILLS,Automating workflow processes
SME4,SKILLS,Developing reverse engineering tools
SME1,STEPS,Depends on the domain
SME2,STEPS,Depends on the domain
SME1,STEPS,for protected programs and malware first get access to
instructions you are interested in
SME3,STEPS,Understand structure of any overlays
SME2,STEPS,for protected programs and malware get access to
instructions
SME3,STEPS,for protected programs and malware get access to
instructions
SME1,STEPS,Make abstractions from function information
SME3,STEPS,Make abstractions from function information
SME1,STEPS,Apply relations (between instructions and between blocks
and calls between functions and between instructions and data areas)
SME1,STEPS,With vulnerabilities where the input handling code is
```

```
SME1,STEPS,With vulnerabilities how to craft the input to exploit the
vulnerability
SME1,STEPS,With malware what the program’s output is
SME1,STEPS,With malware what are the vulnerabilities
SME3,STEPS,With malware static analysis
SME3,STEPS,Label instructions
SME1,STEPS,Fix up references
SME3,STEPS,Fix up references
SME1,STEPS,Label data
SME3,STEPS,Label data members
SME1,STEPS,Label functions
SME1,STEPS,Get around protections
SME2,STEPS,Get around protections
SME3,STEPS,Get around protections
SME4,STEPS,File identification is the first step
SME4,STEPS,Get properties of the program
SME4,STEPS,With malware get the code embedded in malicious documents
SME4,STEPS,Identifying the right tool for the job
SME4,STEPS,Walk through the binary
SME4,STEPS,Abstract into properties of functions
SME4,STEPS,Naming functions to help paint a picture
SME4,STEPS,Abstract into something you can give a meaningful name to
SME5,STEPS,Determine the goal
SME5,STEPS,Learn how to get past protections
SME5,STEPS,Get past protections
SME1,TACIT,Recognizing higher-level code constructs in assembly code
(struct array object class functions for loop counter while loop
switch statement)
SME2,TACIT,With experience comes speed
SME2,TACIT,Knowing which paths are fruitful and which aren’t
SME2,TACIT,Recognizing when you’re going down the wrong path
SME2,TACIT,Coming up with creative / out of the box approaches to
challenging problems
SME3,TACIT,Recognizing when things don’t look right in the program
SME3,TACIT,Recognizing compiler optimizations from anomalies
SME4,TACIT,Recognizing meaning of system calls in context
SME4,TACIT,Recognition of crypto algorithms
SME4,TACIT,Recognizing when import function names are being resolved
SME4,TACIT,Knowing when to trust IDAPro
SME4,TACIT,Understanding when something looks wrong with a function
SME5,TACIT,Knowing whether there might be a vulnerability by looking
at the code
SME5,TACIT,Experience allows you to focus on your goal
```

```
SME1,TOOLS,My particular custom tool
SME2,TOOLS,My particular custom tool
SME3,TOOLS,My particular custom tool
SME4,TOOLS,My particular custom tool
SME2,TOOLS,Disassembler
SME2,TOOLS,Debugger
SME2,TOOLS,Instrumentation tools (to gather information)
SME1,TOOLS,OllyDbg
SME3,TOOLS,OllyDbg
SME1,TOOLS,IDAPro
SME2,TOOLS,IDAPro
SME3,TOOLS,IDAPro
SME1,TOOLS,Immunity
SME1,TOOLS,Windbg
SME2,TOOLS,Windbg
SME1,TOOLS,CFFExplorer
SME1,TOOLS,LordPE
SME2,TOOLS,LordPE
SME1,TOOLS,Custom special purpose tools
SME1,TOOLS,Custom scripts to solve specific tasks
SME1,TOOLS,PELib
SME1,TOOLS,for malware ImpREC
SME1,TOOLS,PEID
SME1,TOOLS,HexRays decompiler
SME3,TOOLS,HexRays decompiler
SME1,TOOLS,Bindff
SME1,TOOLS,Codesurfer
SME1,TOOLS,Patchdiff
SME1,TOOLS,Binnavi
SME1,TOOLS,Responder
SME1,TOOLS,Universal Import Fixer (UIF)
SME3,TOOLS,Run-time debugging
SME3,TOOLS,Stack parsing while program runs
SME3,TOOLS,PEExplorer
SME4,TOOLS,Hex editor
SME4,TOOLS,Imprec
SME4,TOOLS,IDA
SME4,TOOLS,Debugger
SME4,TOOLS,Emulator
SME4,TOOLS,Hex
SME4,TOOLS,Documentation (PE ELF PDF structures MSDN man pages
Google)
SME4,TOOLS,PEID
```

```
SME4,TOOLS,LordPE
SME4,TOOLS,Binnavi
SME5,TOOLS,WinDbg
SME5,TOOLS,In Linux GCC
SME5,TOOLS,Visual studio
SME5,TOOLS,IDA
SME5,TOOLS,Scripts
SME5,TOOLS,Some tool to document your progress and manage
knowledge
SME5,TOOLS,Tracing data flow is difficult
```

Appendix H. Goal-Directed Task Analysis of Reverse Engineering an Executable Program

```
{topgoal}{Quickly Understand as Much as Possible About the Program}

{goal}{Understand the Purpose of Analysis}
  {subgoal}{Understand the type of reverse engineering task}
    {decision}{Is it a malware analysis task?}
      {information}{Trustworthiness of program}
      {information}{Program's originator}
      {information}{Where the program came from}
      {information}{Trustworthiness of program's originator}
    {decision}{Is it a vulnerability discovery task?}
      {information}{System(s) the program will run on}
      {information}{accessibility of the system (network)}
      {information}{typical users of the program and system}
      {information}{scale of the program (how many users)}
      {information}{how protected the system is}
      {information}{the mission supported by the system}
      {information}{access level of the program}
      {information}{whether source code is available}
      {information}{whether the customer wants an exploit}
    {decision}{Does the task involve breaking a protection?}
      {information}{if the program is known to be protected}
      {information}{if circumventing the protection is needed}
      {information}{encryption in program sections}
      {information}{if assembly instructions aren't available}
      {information}{if functionality relies on outside data}
    {decision}{Is task documenting/re-writing the program?}
      {information}{if customer needs to interface with program}
      {information}{if source code is not available}
      {information}{if customer relies on program's functionality}
      {information}{if program's authors are not available}

  {subgoal}{Understand how analysis should be represented}
    {decision}{Will the customer want a report/tutorial?}
      {information}{if the customer asked for a report}
      {information}{whether novel elements need explained}
      {information}{if the purpose is to inform decisions}
      {information}{if the purpose is to develop security controls}
      {information}{if  the purpose is for training others}
    {decision}{Does the customer want a documented IDA database?}
      {information}{if the analysis will be reused}
      {information}{if novel techniques may be present}
      {information}{if other representations are not sufficient}
    {decision}{Does the customer want a proof of concept exploit?}
      {information}{if analysis involves discovering vulnerabilities}
      {information}{if analysis is to justify expenditures}

{goal}{Discover the General Properties of the Program}
  {subgoal}{2.1 Determine the likely trustworthiness of the program}
    {decision}{Does the program come from a trustworthy source?}
    {decision}{Is the program likely malware? (ref)}
    {decision}{Is the program signed by an organization?}
      {information}{program signature}
    {decision}{Does the program’s hash value match the public hash?}
      {information}{program hash value}
      {information}{publicly-advertised hash value}
    {decision}{Does the program have a normal install process?}
      {information}{install process}
    {decision}{Is there anything else suspicious about the program?}
      {information}{observations about the program}

  {subgoal}{Determine if the program is/has malware}
    {decision}{Does the program fail an anti-virus scan?}
      {information}{antivirus scan results}
    {decision}{Does the program create copies of itself?}
      {information}{presence of code that writes instructions}
      {information}{observations of a new program written to disk}
      {information}{presence of function to generate semi-random text}
    {decision}{Does the program try to download or run other programs?}
      {information}{Presence of hardcoded IP address in program data}
      {information}{presence of hardcoded web address in program data}
      {information}{network communication over IRC}
      {information}{network communication over an unknown protocol}
    {decision}{Does the program try to hide its presence?}
      {information}{if it traverses the operating system’s process list}
      {information}{presence of device driver code}
      {information}{marks files as hidden}
    {decision}{Does the program attempt to persist on the system?}
      {information}{whether it writes a link to a startup folder}
      {information}{presence of registry keys related to startup}
      {information}{creates a task to be scheduled}
    {decision}{Does the program try to escalate its privilige level?}
    {decision}{Does the program try to get a shell?}
    {decision}{Does the program try to capture user data?}
    {decision}{Is the program packed or encrypted?}
```

```
      {information}{parts of the program do not disassemble}
      {information}{the import address table is not readable}
      {information}{the program matches signature of known packer}
    {decision}{Does the program have an unusual-looking header?}
      {information}{values in normally empty header fields}
      {information}{presence of the DOS header}
      {information}{excessive number of sections}
  {information}{zero-length sections}
  {information}{header marked executable}
    {decision}{Does the code show a normal programming style?}
      {information}{if the code seems to be broken into functions}
      {information}{if the code does not jump around too much}
      {information}{if there is hierarchy structure to the program}
    {decision}{Are suspicious-looking strings visible in the program?}
      {information}{values of the strings}
    {decision}{Determine if the program should be monitored}
  {information}{If the program spawns other threads}
      {information}{If the program open up strange windows}
      {information}{If the program tries to talk to the network}
  {information}{If the program changes system files}
      {information}{If the program attempts to execute data}
      {information}{If the program tries to register / load drivers}

  {subgoal}{Determine if the program has vulnerabilities}
    {decision}{Does the program seem likely to have vulnerabilities?}
  {information}{Is the program written in an unmanaged language?}
      {information}{Was the program written by a well-known company?}
      {information}{Is UI text presented professionally?}
      {information}{Is the program’s user interface designed well?}
    {decision}{Does the program show indicators of unsafe coding?}
      {information}{Does the program use unsafe API function calls?}
      {information}{Does the program look to use a lot of pointers?}
      {information}{Does the program check input values?}
      {information}{Does the program show a function-based design?}

{goal}{Localize Effects and Behaviors in the Code}

{goal}{Understand How Program Uses the System Interface}
    {subgoal}{Determine system calls made by the program}
    {subgoal}{Determine behaviors made by sequences of calls}

{goal}{Understand, Abstract, and Label Program’s Functions}
    {subgoal}{Determine program behaviors that seem important}
    {subgoal}{Determine intermodular calls that seem important}
    {subgoal}{Determine how functions call each other}
    {subgoal}{Determine the preconditions of a function}
    {subgoal}{Determine the postconditions of a function}
    {sugboal}{Determine program elements changed by a function}
    {subgoal}{Find important functions in the code}

{goal}{Understand, Abstract, and Label Instruction-Level Information}
    {subgoal}{Determine how data flows through the function}
    {subgoal}{Determine what data elements trigger conditional jumps}
    {subgoal}{Match the behavior of the function to a known pattern}

{goal}{Understand how the Program Uses Data}
  {subgoal}{Trace the Data Forward from an Input}
  {subgoal}{Trace the Data Backward from an Event}
    {decision}{Does the event depend on this code?}
  {information}{Map of data flow}

{goal}{Construct a Complete Picture of the Program}
  {subgoal}{Understand the components of the program}
  {subgoal}{Understand how the components work together}
```

Appendix I. Observational Study Coded Data (Participant C)

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
1		1					
2			1				
3	1						
4				1			
5				1			
6				1			
7					1		
8			1				
9		1					
10				1			
11				1			
12			1				
13				1			
14	1						
15		1					
16	1						
17		1					
18			1				
19			1				
20	1						
21		1					
22			1				
23				1			
24					1		
25							1
26							1
27							
28				1			
29				1			
30					1		

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
31				1			
32					1		
33							1
34							1
35						1	
36							1
37	1						
38				1			
39							1
40	1						
41					1		
42						1	1
43							1
44							1
45	1						
46			1				
47				1			
48					1		
49		1					
50				1			
51				1			
52				1			
53					1		
54						1	
55	1						
56	1						
57		1					
58		1					
59				1			
60				1			
61				1			
62							1
63							1
64				1			
65							1
66							1
67					1		
68					1		
69							1
70							1

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
71							1
72					1		
73				1			
74				1			
75				1			
76					1		
77							1
78			1				
79				1			
80					1		
81				1			
82	1						
83					1		
84			1				
85		1					
86							1
87							1
88		1					
89		1					
90				1			
91			1				
92					1		
93			1				
94				1			
95					1		
96					1		
97			1				
98					1		
99		1					
100		1					
101		1					
102				1			
103				1			
104		1					
105							1
106				1			
107				1			
108					1		
109				1			
110					1		

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
111							1
112					1		
113		1					
114			1				
115				1			
116		1					
117	1						
118	1						
119					1		
120					1		
121				1			
122		1					
123		1					
124				1			
125	1						
126							1
127				1			
128						1	
129						1	
130							1
131							1
132					1		
133			1				
134						1	
135					1		
136							1
137							1
138		1					
139							1
140		1					
141					1		
142			1				
143				1			
144			1				
145				1			
146					1		
147							1
148							1
149			1				
150				1			

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
151				1			
152					1		
153					1		
154					1		
155						1	
156							1
157							1
158						1	
159							1
160					1		
161					1		
162				1			
163					1		
164						1	
165						1	
166				1			
167					1		
168							1
169							1
170							1
171				1			
172					1		
173					1		
174				1			
175					1		
176				1			
177							1
178					1		
179		1					
180	1						
181	1						
182			1				
183						1	
184				1			
185							1
186	1						
187					1		
188			1				
189				1			
190				1			

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
191							1
192					1		
193					1		
194						1	
195							1
196							1
197					1		
198						1	
199					1		
200				1			
201				1			
202					1		
203				1			
204					1		
205							1
206							1
207				1			
208						1	
209						1	
210							1
211	1						
212				1			
213				1			
214					1		
215				1			
216					1		
217				1			
218							1
219		1					
220			1				
221				1			
222				1			
223				1			
224					1		
225						1	
226						1	
227				1			
228						1	
229					1		

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
230					1		
231					1		
232					1		
233					1		
234					1		
235						1	
236							1
237					1		
238				1			
239				1			
240			1				
241					1		
242					1		
243				1			
244				1			
245				1			
246				1			
247				1			
248				1			
249				1			
250					1		
251			1				
252		1					
253				1			
254				1			
255					1		
256					1		
257						1	
258							1
259		1					
260				1			
261							1
262	1						
263		1					
264							1
265							

Appendix J. Observational Study Coded Data (Participant D)

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
1	1						
2		1					
3					1		
4						1	
5	1						
6	1						
7					1		
8					1		
9						1	
10	1						
11	1						
12				1			
13						1	
14							1
15	1						
16				1			
17						1	
18						1	
19	1						
20				1			
21		1					
22					1		
23	1						
24		1					
25	1						
26	1						
27				1			
28				1			
29					1		
30				1			

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
31				1			
32	1						
33							1
34	1						
35			1				
36	1						
37	1						
38					1		
39						1	
40				1			
41				1			
42							1
43	1			1			
44			1				
45				1			
46	0			1			
47				1			
48	1						
49	0			1			
50					1		
51	1						
52	0				1		
53	1						
54	0				1		
55	0			1			
56					1		
57			1				
58					1		
59							1
60			1				
61					1		
62				1			
63					1		
64				1			
65				1			
66				1			
67				1			
68					1		
69				1			
70				1			

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
71				1			
72				1			
73						1	
74	1						
75	1						
76			1				
77							1
78				1			
79					1		
80					1		
81						1	
82				1			
83					1		
84	1						
85				1			
86		1					
87				1			
88				1			
89	1						
90				1			
91	1						
92							1
93	1						
94	1						
95				1			
96				1			
97	1						
98			1				
99				1			
100					1		
101				1			
102	1						
103				1			
104			1				
105							1
106	1						
107	1						
108	1						
109	1						
110				1			

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
111	1						
112				1			
113					1		
114	1						
115			1				
116		1					
117			1				
118				1			
119		1					
120			1				
121						1	
122						1	
123		1					
124							1
125	1						
126	1						
127		1					
128				1			
129		1					
130				1			
131						1	
132						1	
133				1			
134						1	
135				1			
136				1			
137					1		
138					1		
139							1
140		1					
141		1					
142				1			
143		1					
144			1				
145			1				
146				1			
147					1		
148					1		
149					1		
150	1						

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
151		1					
152				1			
153				1			
154				1			
155				1			
156					1		
157						1	
158					1		
159			1				
160							1
161							1
162	1						
163		1					
164				1			
165				1			
166					1		
167		1					
168	1						
169		1					
170	1						
171		1					
172	1						
173		1					
174		1					
175		1					
176		1					
177				1			
178					1		
179				1			
180				1			
181				1			
182				1			
183				1			
184				1			
185					1		
186					1		
187				1			
188		1					
189		1					
190		1					

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
191			1				
192	1						
193			1				
194	1						
195			1				
196				1			
197					1		
198	1						
199			1				
200			1				
201				1			
202		1					
203	1						
204				1			
205		1					
206			1				
207				1			
208					1		
209	1						
210		1					
211		1					
212			1				
213				1			
214			1				
215		1					
216				1			
217						1	
218		1					
219				1			
220		1					
221	1						
222	1						
223	1						
224	1						
225			1				
226		1					
227		1					
228				1			
229				1			
230		1					

	Goal	**Plan**	**Carry**	**Sense**	**Interpret**	**Update**	**Hypothesis**
231				1			
232		1					
233			1				
234		1					
235			1				
236				1			
237		1					
238	1						
239		1					
240					1		
241	1						
242		1					
243			1				
244					1		
245				1			
246				1			
247				1			
248					1		
249				1			
250					1		
251					1		
252				1			
253					1		
254					1		
255					1		
256					1		
257					1		
258				1			
259				1			
260				1			

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
261					1		
262				1			
263			1				
264						1	
265			1				
266				1			
267		1					
268					1		
269					1		
270	1						
271				1			
272					1		
273						1	
274				1			
275				1			
276				1			
277				1			
278				1			
279			1				
280				1			
281				1			
282					1		
283					1		
284				1			
285					1		
286				1			
287				1			
288				1			
289				1			
290				1			
291					1		
292				1			
293					1		
294				1			
295				1			
296					1		
297						1	
298				1			
299				1			
300				1			

	Goal	Plan	Carry	Sense	Interpret	Update	Hypothesis
301				1			
302				1			
303				1			
304				1			
305				1			
306				1			
307				1			
308				1			
309				1			
310				1			
311				1			
312				1			
313				1			
314			1				
315				1			
316				1			
317						1	
318				1			
319				1			
320					1		

Bibliography

1. Aho A., Lam M., Sethi R., and Ullman J. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2007.

2. Allemang D. "Functional representation and program debugging. Introduction: Plans and devices," *Artificial Intelligence*, 136–143 (1991).

3. Anderson C. A., Lepper M. R., and Ross L. "Perseverance of Social Theories: The Role of Explanation in the Persistence of Discredited Information," *Journal of Personality and Social Psychology*, *39*:10371049 (1980).

4. Anderson J. R., Bothell D., Byrne M. D., Douglass S., Lebiere C., and Qin Y. "An integrated theory of the mind," *Psychological Review*, *111*(4):1036–60 (October 2004).

5. Anderson J. "Methodologies for studying human knowledge," *Behavioral and Brain Sciences*, *10*(03):467–505 (February 1987).

6. Anderson J. *How Can the Human Mind Occur in the Physical Universe?*, *3*. Oxford University Press, USA, 2007.

7. Anderson T. W. and Goodman L. A. "Statistical inference about Markov chains," *Annals of Mathematical Statistics*, *28*:89–110 (1957).

8. Bakeman R. and Gottman J. M. *Observing Interaction: An Introduction to Sequential Analysis* (2nd Edition). Cambridge: Cambridge University Press, 1997.

9. Balakrishnan G., Reps T., Melski D., and Teitelbaum T. "WYSINWYX: What you see is not what you execute," *Verified Software: Theories, Tools, Experiments*, 202–213 (2007).

10. Barsalou L. W. "Perceptual symbol systems," *The Behavioral and Brain Sciences*, *22*(4):577–609; discussion 610–60 (August 1999).

11. Barsalou L. "Simulation, situated conceptualization, and prediction," *Philosophical Transactions of the Royal Society B: Biological Sciences*, *364*(1521):1281–1289 (2009).

12. Barsalou L., Yeh W., Luka B., Olseth K., Mix K., and Wu L. "Concepts and meaning." *Chicago Linguistics Society 29: Papers from the parasession on conceptual representations* edited by K. Beals, et al., Chicago Linguistics Society, 1993.

13. Basili V. and Mills H. "Understanding and Documenting Programs," *IEEE Transactions on Software Engineering*, *SE-8*(3):270–283 (May 1982).

14. Baxter G., Monk A., Tan K., Dear P., and Newell S. "Using cognitive task analysis to facilitate the integration of decision support systems into the neonatal intensive care unit," *Artificial Intelligence in Medicine*, *35*(3):243–257 (2005).

15. Bayer U., Kirda E., and Kruegel C. "Improving the efficiency of dynamic malware analysis." *Proceedings of the 2010 ACM Symposium on Applied Computing*. 1871–1878. 2010.

16. Bayer U., Moser A., Kruegel C., and Kirda E. "Dynamic analysis of malicious code," *Journal of Computer Virology*, *2*(1):67–77 (2006).

17. Beklin N. J. "Anomalous states of knowledge as a basis for information retrieval," *Canadian Journal of Information and Library Science*, *5* (1980).

18. Benbasat I., Goldstein D., and Mead M. "The case research strategy in studies of information systems," *MIS quarterly*, 369–386 (1987).

19. Biggerstaff T., Mitbander B., and Webster D. "Program understanding and the concept assignment problem," *Communications of the ACM*, *37*(5):72–82 (1994).

20. Binmore K. "Making decisions in large worlds," *Annales d'Economie et de Statistique*, *86*:4 (2007).

21. Birrer B., Raines R., Baldwin R., Oxley M., and Rogers S. "Using qualia and hierarchical models in malware detection," *Journal of Information Assurance and Security*, *4*:247–255 (2009).

22. Blunden B. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Wordware, 2009.

23. Booker L. *Intelligent Behavior as an Adaptation to the Task Environment*. PhD dissertation, The University of Michigan, 1982.

24. Bozsahin H. C. and Findler N. V. "Memory-based hypothesis formation: heuristic learning of commensense causal relations from text," *Cognitive Science*, *16*:431–454 (1992).

25. Brickner M. and Lipshitz R. *Pilot study: System model of situation awareness:"Sensemaking" and decision making in command and control*. Journal article, Human Effectiveness Directorate, Air Force Research Laboratory, 2004.

26. Bright P., "pwn2own day one: Safari, IE8 fall, Chrome unchallenged," 2011.

27. Brinson A., Robinson A., and Rogers M. "A cyber forensics ontology: Creating a new approach to studying cyber forensics," *Digital Investigation*, *3*:37–43 (September 2006).

28. Brooks R. "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, *18*(6):543–554 (1983).

29. Brooks R. “Elephants don’t play chess,” *Robotics and Autonomous Systems*, *6*(1-2):3–15 (June 1990).

30. Bryant A. “Toward detecting novel software attacks by using constructs from human cognition.” *Proceedings of the 3rd International Conference on Information Warfare and Security, Peter Kiewit Institute, University of Nebraska, Omaha USA, 24-25 April 2008*. 2008.

31. Bryant A. and Grimaila M. “Developing a Framework to Improve Information Assurance Battlespace Knowledge.” *Proceedings of the 2nd International Conference on Information Warfare & Security*. 17. Academic Conferences Limited, 2007.

32. Bryant A. *Evaluating Organizational Information Assurance Metrics Programs (Masters Thesis)*. Technical Report, Department of Systems and Engineering Management, Air Force Institute of Technology, 2007.

33. Bryant A., Mills R., Peterson G., and Grimaila M. “(in press) Software reverse engineering as a sensemaking task,” *Journal of Information Assurance and Security* (2012).

34. Bylander T., Allemang D., Tanner M., and Josephson J. “The computational complexity of abduction* 1,” *Artificial Intelligence*, *49*(1-3):25–60 (1991).

35. Byrne E. “A conceptual foundation for software re-engineering.” *Proceedings of the Conference on Software Maintenance, 1992*. 226–235. 1992.

36. CamStudio Developers , “CamStudio,” 2011.

37. Canzanese , Jr.R., Oyer M., Mancoridis S., and Kam M., “A survey of reverse engineering tools for the 32-bit Microsoft Windows environment,” 2005.

38. Card S., Moran T., and Newell A. *The Psychology of Human-Computer Interaction*. CRC, 1983.

39. Chandrasekaran B. *Design and Diagnosis Problem Solving with Multifunctional Technical Knowledge Bases*. Technical Report, Laboratory for Artificial Intelligence Research, The Ohio State University, 1992.

40. Chi E., Pirolli P., Chen K., and J. P. “Using information scent to model user information needs and actions and the web.” *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 490 – 497. 2001.

41. Chi M. T. H. “Quantifying qualitative analyses of verbal data: A practical guide,” *Journal of the Learning Sciences*, *6*:271–315 (1997).

42. Chikofsky E., Cross J., and Others . “Reverse engineering and design recovery: A taxonomy,” *IEEE Software*, *7*(1):13–17 (1990).

43. Christodorescu M., Jha S., and Kruegel C. “Mining specifications of malicious behavior.” *Proceedings of the the 6th Joint Meeting of the European Software En-*

gineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. 5–14. ACM, 2007.

44. Christodorescu M., Jha S., Seshia S., Song D., and Bryant R. "Semantics-aware malware detection." *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. 32–46. 2005.

45. Clark R. and Estes F. "Cognitive task analysis for training," *International Journal of Educational Research*, *25*(5):403–417 (1996).

46. Cohen F. "Computer viruses: Theory and experiments," *Computers & Security*, *6*(1):22–35 (1987).

47. Cohen J. and others . "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, *20*(1):37–46 (1960).

48. Cohen L., Manion L., Morrison K., and Morrison K. *Research Methods in Education*. Psychology Press, 2007.

49. Collaborative RCE Tool Library , "LordPE," 2011.

50. Collberg C., Thomborson C., and Low D. *A taxonomy of obfuscating transformations*. Technical report, Department of Computer Science, University of Auckland, 1997.

51. Conrad F., Blair J., and Tracy E. "Verbal reports are data! A theoretical approach to cognitive interviews." *Proceedings of the Federal Committee on Statistical Methodology Research Conference*. 11–20. Citeseer, 1999.

52. Cooke N. "Varieties of knowledge elicitation techniques," *International Journal of Human-Computer Studies*, *41*(6):801–849 (1994).

53. Corritore C. "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *International Journal of Human-Computer Studies*, *54*(1):1–23 (January 2001).

54. Crandall B. and Getchell-Reiter K. "Critical decision method: A technique for eliciting concrete assessment indicators from the intuition of NICU nurses," *Advances in Nursing Science* (1993).

55. Crandall B., Klein G., and Hoffman R. *Working Minds: A Practitioner's Guide to Cognitive Task Analysis*. Cambridge, MA: The MIT Press, 2006.

56. Crandall B., "Personal communication," May 2009.

57. Cyclops , "Crackmes by Cyclops," 2011.

58. D'Amico A., Whitley K., Tesone D., O'Brien B., and Roth E. "Achieving cyber defense situational awareness: a cognitive task analysis of information assurance analysts." *Proceedings of the Human Factors and Ergonomics Society Annual Meeting49*. 229–233. 2005.

59. Davenport T. H. and Prusak L. *Working Knowledge: How Organizations Manage What They Know*. Harvard Business Press, 2000.

60. De Groot A. *Thought and Choice in Chess*, *4*. Mouton De Gruyter, 1978.

61. Détienne F. "Expert programming knowledge: a schema-based approach." *Psychology of Programming* 205 – 222, Academic Press, People and Computer Series, 1990.

62. Dixon S., Wickens C., and Chang D. "Mission control of multiple unmanned aerial vehicles: A workload analysis," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *47*(3) (2005).

63. Drepper U., "What every programmer should know about memory (2007)," 2010.

64. Duala-Ekoko E. and Robillard M. "Tracking code clones in evolving software." *Proceedings of the 29th international conference on Software Engineering*. 158–167. 2007.

65. Eilam E. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.

66. Eisenhardt K. M. "Building Theories from Case Study Research," *Academy of Management Review*, *14*(4):532–550 (1989).

67. Endsley M. R. "Situation models: An avenue to the modeling of mental models." *Proceedings of the 14th Triennial Congress of the International Ergonomics Association and the 44th Annual Meeting of the Human Factors and Ergonomics Society*. 2000.

68. Endsley M. "Toward a theory of situation awareness in dynamic systems," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *37*(1):32–64 (1995).

69. Endsley M. and Rodgers M. "Situation awareness information requirements analysis for en route air traffic control." *Proceedings of the Human Factors and Ergonomics Society 38th Annual Meeting38*. 71–75. 1994.

70. Engelmore R., Feigenbaum E., Friedland P., Johnson B., Shrobe H., Schorr H., and Nii H. *Knowledge-based systems in Japan.* Technical Report, Japanese Technology Evaluation Center, 1993.

71. Erickson J. *Hacking: The Art of Exploitation*. No Starch Press, 2008.

72. Ericsson K. and Simon H. "Verbal reports as data," *Psychological review*, *87*(3):215 (1980).

73. Fikes R. and Nilsson N. "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, *2*(3-4):189–208 (1971).

74. Fix V. and Wiedenbeck S. "Mental representations of programs by novices and experts," *Proceedings of INTERCHI*, 74–79 (1993).

75. Fu W. and Pirolli P. "SNIF-ACT: A cognitive model of user navigation on the World Wide Web," *Human-Computer Interaction*, *22*(4):355–412 (2007).

76. Gaddis T. *Starting Out With C++: From Control Structures Through Objects* (6th Edition). MA: Boston: Pearson Education, 2009.

77. Gannod G. and Cheng B. "A formal approach for reverse engineering: a case study," *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, 100–111 (1999).

78. Geertz C. "Thick description: towards an interpretive theory of culture." *The Interpretation of Cultures* edited by C. Geertz, New York: Basic Books, 1973.

79. Geib C. "Plan recognition." *Adversarial Reasoning: Computational Approaches to Reading the Opponent's Mind* edited by A. Kott and W. M McEneaney, 77–95, Chapman & Hall/CRC, 2007.

80. Gentner D. and Stevens A. *Mental Models*. Lawrence Erlbaum, 1983.

81. Gharajedaghi J. *Systems Thinking: Managing Chaos and Complexity: A Platform for Designing Business Architecture*. Elsevier, 1999.

82. Goodall J., Radwan H., and Halseth L. "Visual analysis of code security." *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*. 46–51. 2010.

83. Green T. "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, *7*(2):131–174 (June 1996).

84. Grimaila M. and Fortson L. "Towards an Information Asset-Based Defensive Cyber Damage Assessment." *Proceedings of the IEEE Symposium on Computational Intelligence in Security and Defense Applications (CISDA)*. 2007.

85. Heelan S. "Vulnerability Detection Systems: Think Cyborg, Not Robot," *Security & Privacy, IEEE*, *9*(3):74–77 (2011).

86. Hendry D. "Sketching with conceptual metaphors to explain computational processes." *Visual Languages and Human-Centric Computing (VL/HCC'06)*. 95–102. IEEE, 2006.

87. Hennessy J., Patterson D., and Goldberg D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

88. Hex-Rays , "The IDA Pro Disassembler and Debugger," 2011.

89. Hitchcock G. and Hughes D. *Research and the Teacher: A Qualitative Introduction to School-Based Research*. Burns & Oates, 1995.

90. Hoffman R., "Eliciting knowledge from experts: A methodological analysis," May 1995.

91. Hoglund G. and Butler J. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2006.

92. Hoglund G. and McGraw G. *Exploiting Software: How to Break Code*. Addison-Wesley, 2005.

93. Hopgood A. *Knowledge-Based Systems for Engineers and Scientists*. CRC Press, Inc. Boca Raton, FL, USA, 1993.

94. Huang P. and Sycara K. "Learning from and about the opponent." *Adversarial Reasoning: Computational Approaches to Reading the Opponents Mind* edited by A. Kott and W. McEneaney, Boca Raton: Chapman & Hall/CRC Computer and Information Science Series, 2007.

95. IARPA Office of Safe and Secure Operations , "Solicitations: Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) Program," 2009.

96. Immunity, Inc. , "Immunity Debugger," 2011.

97. Intel, Inc. . *Intel 64 and IA-32 Architectures Software Develper's Manual. Combined Volumes: 1, 2A, 2B, 3A and 3B*. Intel, Inc., 2011.

98. John B. E. and Kieras D. E. "Using GOMS for user interface design and evaluation: Which technique?," *ACM Transactions on Computer-Human Interaction*, *3*(4):287–319 (December 1996).

99. Johnson T. and Krems J. "Use of current explanations in multicausal abductive reasoning," *Cognitive Science*, *25*(6):903–939 (2001).

100. Johnson W. and Soloway E. "PROUST: Knowledge-based program understanding," *IEEE Transactions on Software Engineering*, 267–275 (1985).

101. Johnson-Laird P. *Mental Models*. Cambridge Univ. Press Cambridge, UK, 1983.

102. Johnson-Laird P. *How We Reason*. Oxford University Press, USA, 2009.

103. Josephson J. R., Chandrasekaran B., Smith J. W., and Tanner M. C. "A Mechanism for Forming Composite Explanatory Hypotheses," *IEEE Transactions on Systems, Man, and Cybernetics*, *17*(3):445–454 (1987).

104. Josephson J. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge Univ Pr, 1996.

105. Kahneman D. *Attention and Effort*. Prentice-Hall, Inc., 1973.

106. Kahneman D. *The Simulation Heuristic*. Technical Report, Department of Psychology, Stanford University, 1981.

107. Kahneman D., Slovic P., and Tversky A., editors. *Judgment Under Uncertainty: Heuristics and Biases*. New York: Cambridge University Press, 1982.

108. Kapoor A. *An approach towards disassembly of malicious binary executables*. Master's Thesis, The Center for Advanced Computer Studies, University of Louisiana at Lafayette, 2004.

109. Kemp C., Goodman N., and Tenenbaum J. "Learning causal schemata." *Proceedings of the Twenthy-Ninth Annual Meeting of the Cognitive Science Society*. 389–394. 2007.

110. Kintsch W. "The role of knowledge in discourse comprehension: A construction-integration model," *Psychological review*, *95*(2):163 (1988).

111. Kirwan B. and Ainsworth L. *A Guide to Task Analysis*. Taylor & Francis, 1992.

112. Klein G. *The Power of Intuition: How to Use Your Gut Feelings to Make Better Decisions at Work*. Random House, Inc., 2004.

113. Klein G. "Naturalistic decision making," *Human Factors*, *50*(3) (2008).

114. Klein G., Moon B., and Hoffman R. "Making sense of sensemaking 1: Alternative perspectives," *IEEE Intelligent Systems*, *21*(4):70–73 (2006).

115. Klein G., Phillips J., Rall E., and Peluso D. "A data-frame theory of sensemaking." *Expertise Out of Context: Proceedings of the Sixth International Conference on Naturalistic Decision Making*. 113–155. 2007.

116. Lee N. and Johnson-Laird P. "Synthetic reasoning and the reverse engineering of boolean circuits." *Proceedings of the Twenty-Seventh Annual Conference of the Cognitive Science Society, Stresa, Italy*. 1260–1265. 2005.

117. Leedy P. D. and Ormrod J. E. *Practical Research: Planning and Design* (7th Edition). Upper Saddle River, NJ: Merrill Prentice Hall, 2001.

118. Lethbridge T., Sim S., and Singer J. "Studying software engineers: data collection techniques for software field studies," *Empirical Software Engineering*, *10*:311–341 (2008).

119. Letovsky S. and Soloway E. "Delocalized plans and program comprehension," *IEEE Software*, *3*(3):41–49 (May 1986).

120. Menzies T. "Applications of abduction: Knowledge-level modelling," *International Journal of Human-Computer Studies*, *45*(3):305–335 (September 1996).

121. Merrill M. "Knowledge objects and mental models." *Advanced Learning Technologies, 2000. IWALT 2000. Proceedings. International Workshop on*. 244–246. 2000.

122. Meunier P. "Classes of Vulnerabilities and Attacks." *Wiley Handbook of Science and Technology for Homeland Security* edited by J. G. Voeller, Wiley, 2011.

123. mh nexus , "HxD - Freeware Hex Editor and Disk Editor," 2011.

124. Michalewicz Z. and Fogel D. *How to Solve It: Modern Heuristics*. Springer-Verlag New York Inc, 2004.

125. Microsoft, Inc. , “WinDbg,” 2011.

126. Microsoft, Inc. , “Windows XP Professional,” 2011.

127. Militello L. and Hutton R. “Applied Cognitive Task Analysis (ACTA): A practitioner’s toolkit for understanding cognitive task demands,” *Ergonomics*, *41*(11):1618–1641 (1998).

128. Militello L., Hutton R., Pliske R., Knight B., Klein G., and Randel J. *Applied cognitive task analysis (ACTA) methodology: A detailed description of how to conduct the ACTA method*. Journal article, Navy Personnel Research and Development Center, 1997.

129. Miller T. “A Cognitive approach to developing tools to support planning.” *Linking Expertise and Naturalistic Decision Making* edited by E. Salas and G. A. Klein, 95–111, Lawrence Erlbaum, 2001.

130. Minsky M. “A Framework for Representing Knowledge.” *The Psychology of Computer Vision* edited by P. Winston, McGraw-Hill, 1974.

131. Morik K., Kietz B., Emde W., and Wrobel S. *Knowledge acquisition and machine learning*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA, 1993.

132. Morris N. and Rouse W. “Review and evaluation of empirical research in troubleshooting,” *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *27*(5):503–530 (1985).

133. Nash T. *An Undirected Attack Against Critical Infrastructure*. Technical Report, US-CERT Control Systems Security Center, 2005.

134. Newell A. *Unified Theories of Cognition*. Cambridge, MA: Harvard University Press, 1990.

135. Newell A. and Simon H. *Human Problem Solving*. Prentice-Hall Englewood Cliffs, NJ, 1972.

136. Ngbala A. and Branscombe N. “Mental simulation and causal attribution: When simulating an event does not affect fault assignment,” *Journal of Experimental Social Psychology*, *31*(2):139–162 (1995).

137. Nonaka I. and Takeuchi H. “The knowledge-creating company,” *Harvard Business Review*, *6*:96–104 (1991).

138. Norman D. “Some observations on mental modeIs.” *Mental Models* edited by D. Gentner, Lawrence Erlbaum, 1983.

139. Norman D. *The Design of Everyday Things*. New York: Basic Books, 2002.

140. OllyDbg , “OllyDbg,” 2011.

141. OReilly R. *The Leabra Model of Neural Interactions and Learning in the Neocortex*. PhD dissertation, Carnegie Mellon University, 1996.

142. Patton M. *Qualitative Evaluation and Research Methods*. Sage Publications, 1990.

143. Pearl J. *Heuristics–intelligent search strategies for computer problem solving*. Addison-Wesley Publishing Co., Reading, MA, 1984.

144. Pearl J. *Causality: Models, Reasoning, and Inference*. Cambridge Univ Pr, 2000.

145. PEiD Team , "PEiD," 2011.

146. Pennington N. "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, *19*(3):295–341 (1987).

147. Pennington N. and Hastie R. "Reasoning in explanation-based decision making," *Cognition*, *49*(1-2):123–163 (1993).

148. Petrowski J. and Taillard P. *Metaheuristics for Hard Optimization*. Springer-Verlag, 2006.

149. Petzold C. *Programming Windows* (5 Edition). Microsoft Press, 1999.

150. Pirolli P. and Card S. "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis." *Proceedings of the International Conference on Intelligence Analysis*. 2–4. 2005.

151. Popper K. *All Life is Problem Solving*. London: Routledge, 1999.

152. Popper K. *Conjectures and Refutations: The Growth of Scientific Knowledge*. London: Routledge, 2003.

153. President of the United States O. *The National Strategy to Secure Cyberspace*. Technical Report, President of the United States, Office of, 2009.

154. Quesada J., Kintsch W., and Gomez E. "Complex problem-solving: a field in search of a definition?," *Theoretical Issues in Ergonomics Science*, *6*(1):5–33 (2005).

155. Quilici a. and Woods S. "Applying plan recognition algorithms to program understanding," *Proceedings of the 11th Knowledge-Based Software Engineering Conference*, *372*:96–103 (1998).

156. Quist D. and Liebrock L. "Visualizing compiled executables for malware analysis." *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*. 27–32. 2009.

157. Rajlich V. "Intensions are a key to program comprehension," *17th International Conference on Program Comprehension*, 1–9 (May 2009).

158. Robson C. *Real World Research* (2nd Edition). Blackwell, 2002.

159. Roth E., Malsch N., Multer J., and Coplen M. "Understanding how train dispatchers manage and control trains: A cognitive task analysis of a distributed

team planning task." *Human Factors and Ergonomics Society Annual Meeting Proceedings43*. 218–222. 1999.

160. Roth E., Woods D., and Pople H. "Cognitive simulation as a tool for cognitive task analysis," *Ergonomics* (1992).

161. Rumelhart D., Ortony A., and Others . *The representation of knowledge in memory*. Center for Human Information Processing, Dept. of Psychology, University of California, San Diego, 1976.

162. Rumelhart D. and Ortony A. "The representation of knowledge in memory." *Schooling and the Acquisition of Knowledge* edited by Anderson, R.C. and Spiro, R.J. and Montague, W.E., Hillsdale, N.J.: Erlbaum, 1977.

163. Runeson P. and Höst M. "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, *14*:131–164 (2009).

164. Russell D. M., Stefik M. J., Pirolli P., and Card S. K. "The cost structure of sensemaking," *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '93*, 269–276 (1993).

165. Russell S. and Norvig P. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

166. Russinovich M. and Cogswell B., "Windows Sysinternals," 2011.

167. Russinovich M. and Solomon D. *Microsoft Windows Internals*. Microsoft Press, 2005.

168. Ryan G. W. and Bernard H. R. "Techniques to Identify Themes," *Field Methods*, *15*(1):85–109 (2003).

169. Sarter N. B. and Woods D. D. "Situation awareness: A critical but ill-defined phenomenon," *The International Journal of Aviation Psychology*, *1*(1):45–57 (1991).

170. Sarter N., Woods D., and Billings C. "Automation surprises." *Handbook of Human Factors and Ergonomics* 1926–1943, John Wiley and Sons, 1997.

171. Schank R., Abelson R., and Others . *Scripts, Plans, Goals and Understanding: An Inquiry Into Human Knowledge Structures*. Lawrence Erlbaum Associates, New Jersey, 1977.

172. Schneider, T. , "crackme.de," jan 2011.

173. Schoenfeld A. and Herrmann D. "Problem perception and knowledge structure in expert and novice mathematical problem solvers," *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *8*(5):484 (1982).

174. Schwarz B., Debray S., and Andrews G. "Disassembly of executable code revisited." *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on.* 45–54. 2002.

175. Schwenk C. "Cognitive simplification processes in strategic decision-making," *Strategic Management Journal*, *5*(2):111–128 (1984).

176. Seaman C. "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, *25*:557–572 (1999).

177. Seamster T., Redding R., Cannon J., Ryder J., and Others . "Cognitive task analysis of expertise in air traffic control," *International Journal of Aviation Psychology* (1993).

178. Shneiderman B. *Software Psychology*. Cambridge, MA: Winthrop Publishers, 1980.

179. Shortliffe E. H. *Computer-based medical consultations: MYCIN*. American Elsevier, 1976.

180. Skoudis E. and Zeltser L. *Malware: Fighting Malicious Code*. NJ: Upper Saddle River: Prentice Hall PTR, 2003.

181. Soloway E. and Ehrlich K. "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, *SE-10* (1984).

182. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M., Liang Z., Newsome J., Poosankam P., and Saxena P. "BitBlaze: A new approach to computer security via binary analysis." *In Proceedings of the 4th International Conference on Information Systems Security (ICISS)*. 1–25. Springer, 2008.

183. Sparks S., Embleton S., Cunningham R., and Zou C. "Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting." *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. 477–486. 2007.

184. Storey M. and Fracchia F. "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software* (1999).

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

186. Szor P. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

187. Tanenbaum A. and Woodhull A. *Operating Systems: Design and Implementation*. Prentice Hall, 1997.

188. The United States Air Force Chief Scientist O. *Report on Technology Horizons: A Vision for Air Force Science & Technology During 2010 - 2030, Volume 1*. Technical Report, United States Air Force, 2010.

189. Therriault D., Rinck M., and Zwwan R. “Assessing the influence of dimensional focus during situation model construction,” *Memory and Cognition*, *34*:78–89 (2006).

190. Thompson K. “Reflections on trusting trust,” *Communications of the ACM*, *27* (August 1984).

191. TightVNC Software , “TightVNC,” 2011.

192. Tilley S. *A Reverse-Engineering Environment Framework*. Technical report, Carnegie-Mellon Software Engineering Institute, 1998.

193. Tonella P., Torchiano M., Du Bois B., and Systä T. “Empirical studies in reverse engineering: state of the art and future trends,” *Empirical Software Engineering*, *12*(5):551–571 (March 2007).

194. Trickett S. and Trafton J. “A primer on verbal protocol analysis.” *The PSI Handbook of Virtual Environments for Training and Education: Developments for the Military and Beyond* edited by D. Schmorrow, et al., Westport, CT Praeger Security International, 2007.

195. Tulving E. and Donaldson W. *Organization of Memory*. Academic Press, 1972.

196. Van de Maele F., “Formal concept analysis in software maintenance: State of the art,” 2007.

197. Van Dijk T. “Relevance assignment in discourse comprehension,” *Discourse Processes*, *2*(2):113–126 (1979).

198. Van Dijk T. and Kintsch W. *Strategies of Discourse Comprehension*. New York: Academic Press, 1983.

199. VanLehn K. “Problem solving and cognitive skill acquisition.” *Foundations of Cognitive Science* edited by Posner, M. I., MIT Press, 1989.

200. Vans M. A. “Program understanding behavior during corrective maintenance of large-scale software,” *International Journal of Human-Computer Studies*, *51*(1):31–70 (July 1999).

201. Vessey I. “Expertise in debugging computer programs: A process analysis,” *International Journal of Man-Machine Studies*, *23*(5):459–494 (1985).

202. von Mayrhauser A. and Vans A. M. “Comprehension processes during large scale maintenance.” *Proceedings of the 16th international conference on Software engineering*. 39–48. IEEE Computer Society Press, 1994.

203. von Mayrhauser a. and Vans A. “From code understanding needs to reverse engineering tool capabilities.” *Computer-Aided Software Engineering, 1993. CASE’93., Proceeding of the Sixth International Workshop on*. 230–239. IEEE, 1993.

204. Weick K. *Sensemaking in Organizations*. Sage Publications, Inc, 1995.

205. Weiser M. “Programmers use slices when debugging,” *Communications of the ACM*, *25*(7):446–452 (July 1982).

206. Wilson M. “Six views of embodied cognition,” *Psychonomic Bulletin & Review*, *9*:625–636 (2002).

207. Wilson T. “Human Information Behavior,” *Informing Science*, *3*:49 – 55 (2000).

208. Wood L. E. “Semi-structured interviewing for user-centered design,” *Interactions*, *4*(2):48–61 (March 1997).

209. Wood L. and Ford J. “Structuring interviews with experts during knowledge elicitation,” *International Journal of Intelligent Systems*, *8*(1):71–90 (1993).

210. Woods D., Patterson E., and Roth E. “Can we ever escape from data overload? A cognitive systems diagnosis,” *Cognition, Technology & Work*, *4*(1):22–36 (2002).

211. Yin R. *Case Study Research: Design and Methods* (5th Edition). Sage Publications, Inc, 2009.

212. Zeigler B. and Hammonds P. *Modeling & Simulation-Based Data Engineering: Introducing Pragmatics Into Ontologies for Net-Centric Information Exchange*. Academic Press, 2007.

213. Zhang J. and Norman D. A. “Representations in distributed cognitive tasks,” *Cognitive Science*, *18*:87–122 (1994).

214. Zhang P. “Supporting sense-making with tools for structuring a conceptual space.” *The 3rd Annual i-Conference (Feb 2008, Los Angeles)*. 1–11. 2008.

215. Zhang P., Soergel D., Klavans J. L., and Oard D. W. “Extending sense-making models with ideas from cognition and learning theories,” *Proceedings of the American Society for Information Science and Technology*, *45*(1):23–23 (June 2009).

216. Zwaan R. and Radvansky G. “Situation models in language comprehension and memory,” *Psychological Bulletin*, *123*(2):162–185 (1998).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
03-22-2012

2. REPORT TYPE
Dissertation

3. DATES COVERED *(From - To)*
07-03-2007 - 03-22-2012

4. TITLE AND SUBTITLE
Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations.

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Mr. Bryant, Adam R., Civilian

5d. PROJECT NUMBER
6095WSPT

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
Wright-Patterson AFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/DCS/ENG/12-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Integrated Electronic & Net-Centric Warfare Division, Air Force Research Laboratory
Dr. David Kapp
2241 Avionics Circle, Wright-Patterson AFB, OH 45433
david.kapp@wpafb.af.mil (937) 528-8054

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RYWA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION IS UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This dissertation develops a theory of the conceptual and procedural aspects involved with how reverse engineers make sense of executable programs. Software reverse engineering is a complex set of tasks which require a person to understand the structure and functionality of a program from its assembly language representation, typically without having access to the program's source code. This dissertation describes the reverse engineering process as a type of "sensemaking," in which a person combines reasoning and information foraging behaviors to develop a mental model of the program. The structure of knowledge elements used in making sense of executable programs are elicited from a case study, interviews with subject matter experts, and observational studies with software reverse engineers. The results from this research can be used to improve reverse engineering tools, to develop training requirements for reverse engineers, and to develop robust computational models of human comprehension in complex tasks where sensemaking is required.

15. SUBJECT TERMS
Sensemaking, Reverse engineering, Human-computer interaction, verbal protocol studies

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Robert F. Mills AFIT/ENG
U	U	U	UU	266	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636 x4527 robert.mills@afit.edu

Reset