Bryant, A., Mills, R., Peterson, G. & Grimaila, M. (2012) Software reverse engineering as a sensemaking task. Journal of Information Assurance and Security. 6(6), pp 483--494.

Software Reverse Engineering as a Sensemaking Task

Adam R. Bryant[1, 2], Robert F. Mills[2], Gilbert L. Peterson[2] and Michael R. Grimaila[2]

[1]Human Effectiveness Directorate, Air Force Research Laboratory,
Wright-Patterson Air Force Base, Ohio, USA 45433
adam.bryant@wpafb.af.mil

[2]Center for Cyberspace Research, Air Force Institute of Technology,
Wright-Patterson Air Force Base, Ohio, USA 45433
robert.mills@afit.edu, gilbert.peterson@afit.edu, michael.grimaila@afit.edu

***Abstract*: Software reverse engineering involves analyzing computer program executables to understand their structure, functionality, and behavior. In this paper, common reverse engineering functions are decomposed to isolate the information-processing and sensemaking subtasks involved. This paper reviews the applicable literature on eliciting mental models of software reverse engineers. Based on the literature, a taxonomy of common processes is developed which leads to a methodology to elicit and represent reverse engineers' mental models of the tasks.**

Keywords: reverse engineering, program understanding, sensemaking, situation awareness, knowledge engineering

I. Introduction

Software reverse engineering is a type of complex task which at the surface involves many common features with the cognitive processes of sensemaking and situation awareness. In particular, reverse engineering an executable program requires a person to process and sift through large amounts of data and integrate that data with extensive specialized background knowledge [17].

As a first step to isolating and understanding the elements of sensemaking in software reverse engineering, this paper reviews existing models of situation awareness and sensemaking in the context of human comprehension processes in software reverse engineering. This paper also describes the application of techniques borrowed from knowledge engineering and decision making research to study cognitive processes involved in reverse engineering.

The paper is organized as follows: First, a discussion of the data representation in a reverse engineering task environment is presented. Next conceptual models of situation understanding and sensemaking processes are presented to outline the major functions of each construct. Next, a methodology is presented for eliciting and representing knowledge from reverse engineers to capture (1) the basic semantics and structure of knowledge required to solve problems intelligently in the task domain, and (2) behavioral processes involving sensemaking in the task domain for further, more refined investigation.

This paper's original contribution is the synthesis and refinement of models of sensemaking and situation awareness with cognitive processes in software reverse engineering.

II. Data Representation in a Reverse Engineering Task Environment

Reverse engineers analyze programs to discover and correct implementation flaws in software, to verify and strengthen security protections in software-based systems, to mitigate potential attacks on software, or to understand potentially malicious code [6], [29], [30], [80].

Reverse engineering a program from its executable form may be required because the source code of the program is not available. In this case, reverse engineers work with assembly language representations of programs, and may have to generate those representations themselves. Analyzing programs from assembly language is more complicated because assembly language typically has a one-to-one mapping with machine code the computer understands, and thus lacks programming abstractions that exist in higher-level languages [76].

The benefit of working from only assembly-level representations is that reverse engineers can develop a very detailed understanding of the exact behavior of programs they investigate. This understanding relies on a much smaller number of unchecked assumptions than learning about a program from its published application programming interface or even its source code. In this way, reverse engineers can collect observable data about how software actually interfaces with hardware and performs operations. In fact, many people who reverse engineer programs from binary representations do not consider an activity reverse engineering if it means reading source code [17], [29], [30].

1) *Assembly Instruction Data*

Once a program is compiled, it can contain thousands or millions of assembly language instructions. These instructions perform operations on the processor and memory, provide calls to operating system functions, and access and manipulate data [74], [27]. On x86-based processors, the assembly language bytes represent instructions that can be translated by a disassembler into opcodes such as *push*, *mov*, *jnz*, and *call* and register name or memory address operands.

In the x86 instruction set architecture, instruction mnemonics can be of various lengths [33], so even disassembling the instructions correctly can prove to be difficult [71]. Processor architectures may have several ways to accomplish the same behavior using different assembly instructions. One instruction opcode may have exactly the same effects as another seemingly unrelated opcode. Or the exact effects of two different instructions can be different, but the major effect of the different instructions is the same. For example, in x86 the *nop* instruction simply exchanges a register for itself, an operation with no side effects [33]. Any other set of instructions with no side effects could be considered basically the same instruction.

The actual instruction that executes can vary depending on the instruction prefixes which comprise the first bytes of the instruction, other bytes within the instruction (such as the mod R/M byte), or the state of the CPU and settings on bit flag registers. Additionally, the x86 instruction set architecture also allows a flexible form of addressing where an instruction may start at any addressable byte [33], [71]. This means that a multi-byte instruction might be perceived as one instruction when read from its first byte, but a different instruction when read from the second or third byte.

A reverse engineer can step through a disassembled program in a debugger instruction by instruction or read sequences of instructions to interpret their meanings. A reverse engineer can also monitor how the value of register or memory values change with each instruction, a process called data slicing [89].

Besides assembly instructions, data from the environment can include bytes in hexadecimal, control flow sequences, sequences of system calls, trace data, register values, memory values, state transition diagrams, graph structures, or text. Data can also include indications from interfaces with the program, data traveling over hardware, a system bus, or a network. It may also consist of operating system data such as the contents of interrupt vector tables, system call tables, and process control blocks.

2) *Program Data*

A string of bytes in a program's data section may have several different structural interpretations, depending on how it is represented and on which byte the disassembler attempts to begin to interpret the string [41]. The string of bytes could represent a sequence of instructions, flags to change control flow, or function call arguments. Data stored in memory locations, registers, or system objects can be represented as strings of binary values, integers, floating point values, ASCII or UNICODE text strings, hexadecimal values, or as parts of more complex data structures.

Program data can include the state of the processor and the contents of the registers, memory regions outside of the process which the program might gain access to, and file or registry contents which the program may change through system calls.

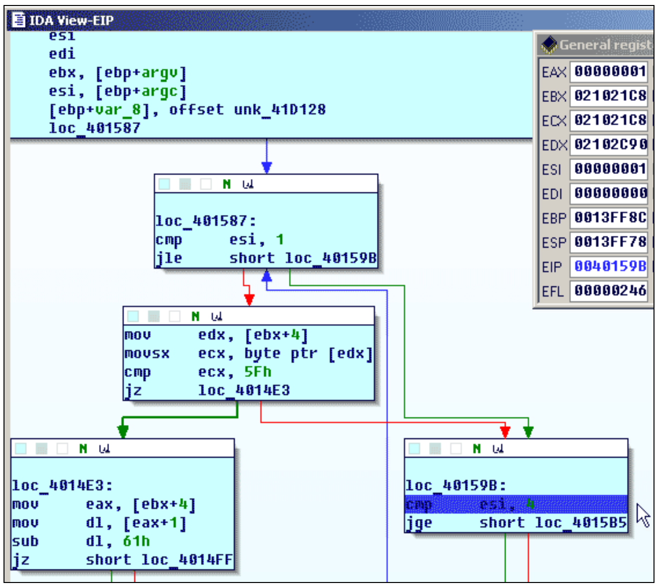
3) *Visual Data*

Another data representation sometimes available to reverse engineers are functional or control-flow abstractions. A control-flow graph of a program breaks the program up into functional components (often basic blocks demarcated by *jmp* or *call* instructions). The graph is a visualization which allows the user to see connections between the basic blocks. The ability to see the program in basic blocks can provide visual cues to help a person divide the program into meaningful modules. Some tools, such as Hex Rays' Interactive Disassembler [28] provide the ability to step through a control flow graph view as shown in Figure 1.

In addition to control flow visualizations, there are other visual aspects to reading code. Brooks [8] describes the use of beacons, which are textual, structural, or visual referents that help people reading code match up mental representations of programming structures with actual implementations in the program.

4) *Instrumentation Data*

Data from system probes can provide reverse engineers additional sources of information. Running systems may have directly observable changes, such as windows that open, files being created in a folder, or system data structures being written to. Other system changes can be detected by using programs or tools to gather the data. If built-in operating system monitoring tools do not provide enough information (or are not trusted), reverse engineers can use third-party tools or write their own tools to support introspection.



```
    esi
    edi
    ebx, [ebp+argv]
    esi, [ebp+argc]
    [ebp+var_8], offset unk_41D128
    loc_401587

loc_401587:
cmp     esi, 1
jle     short loc_40159B

mov     edx, [ebx+4]
movsx   ecx, byte ptr [edx]
cmp     ecx, 5Fh
jz      loc_4014E3

loc_4014E3:
mov     eax, [ebx+4]
mov     dl, [eax+1]
sub     dl, 61h
jz      short loc_4014FF

loc_40159B:
cmp     esi, 4
jge     short loc_4015B5
```

```
EAX 00000001
EBX 021021C8
ECX 021021C8
EDX 02102C90
ESI 00000001
EDI 00000000
EBP 0013FF8C
ESP 0013FF78
EIP 0040159B
EFL 00000246
```

Figure 1. Graph-based debugging in IDA Pro (from [28])

Reverse engineering tools can be roughly categorized as probe tools (or sensors) and control tools. Probe tools provide information about the program, the operating system, hardware, and other parts of the system directly. Control tools change the functionality of the program, system, hardware, etc. in order to help the reverse engineer gather information. Both probes and control tools provide the reverse engineer information about the system and the program under investigation.

For example, a reverse engineer can run a program to list running processes and threads before starting an application, then check which processes have changed after starting it. Another tool could indicate if a program has encrypted sections when stored on disk or reconstruct mangled import functions that a program relies on [32]. Other tools allow viewing the file header data in a program [44], monitoring and capturing network packets from a system's network interface, or capturing system memory to detect changes to system data structures [93]. Some can detect and capture other programs that open and close quickly to avoid being detected [26].

III. Situation Awareness and Understanding

Situation awareness is a term used to describe the degree of a person's perception of relevant information elements in their current task environment, their integration of this information with their task goals, and their ability to project the state of these elements into the future [18]. The term is commonly used to describe a person's attention to data in the environment that is relevant to the task at hand, the overall mission and personal goals [66].

Various methods have been used from the disciplines of knowledge engineering and human factors psychology to measure situation awareness and to better understand decision making in complex environments. Such approaches have been used to study the cognitive work of pilots [18], air traffic controllers [73], [19], unmanned aerial vehicle operators [16], nuclear power plant workers [62], electronic warfare technicians [47], nurses [14], fire fighters [39], and many other types of workers. The underlying theme of these methods is the assumption that when people perform tasks, their understanding and awareness of what is going on is a primary input into the decisions they must make in the course of the task.

Software reverse engineering requires the reverse engineer to understand the task environment in great detail. These tasks involve the heavy use and understanding of automated technologies, namely computer programs. The work is often security-related and as such can be geared toward determining whether a system or program should be trusted by other users. Some human factors studies looked at the link between subjects' understanding of a particular technology and their ability to perform in tasks requiring the technology [67], [68], [69]. In these studies, the subjects' understanding of the technology used made a difference in how well they were able to work with it. Software reverse engineering tasks leverage similar situation understanding requirements on reverse engineers.

There are several measures of situation awareness and situation understanding, but there is no agreement on measures that are general across task environments [18], [66], [81], [82]. Additionally, in many task domains, people performing tasks with automated systems are actively engaged with and must interact and provide inputs to the systems [90].

Often, with highly complex automated systems, the role of the person using the technology is relegated to a "monitor, exception handler, and manager of automated resources" [65]. However, awareness and understanding of the "state" of an automated system like a program requires more than the passive monitoring of the state variables. For reverse engineers to gain situation understanding of their task environment, they have to identify the elements of state information and comprehend the processes involved in those programs [65].

IV. Sensemaking

Where situation awareness refers an understanding of perceptual elements in the environment, sensemaking refers to the processes that enable one to come to that understanding and maintain it [40]. Sensemaking is a term used to refer to humans' capability to actively comprehend the significance of ambiguous events and data [92]. The sensemaking process is typically described as an ongoing integration of knowledge from a mental model of a situation, available data about the situation, and perceptual information from the environment. It is also regarded as the basis for intuitive decision making [39].

Sensemaking is the comprehension process that takes place when expectations turn out to be incorrect. In a task, when people do not have to integrate surprising results, their reasoning and actions are considered operating under business as usual. However, when they experience surprise, they have to make sense of the new information. Sensemaking hinges on instances where data from the environment is inconsistent with a previous understanding of the world.

The sensemaking process is described by [38] as integrating what is conjectured with what is known, connecting what a person infers and actually observes, finding explanations for ambiguous data, diagnosing ambiguous symptoms, and identifying problems (Figure 2). Since these different functions describe a great number of cognitive capabilities, there is a possibility that sensemaking actually describes a class of reasoning capabilities encompassing a number of separate but related process.

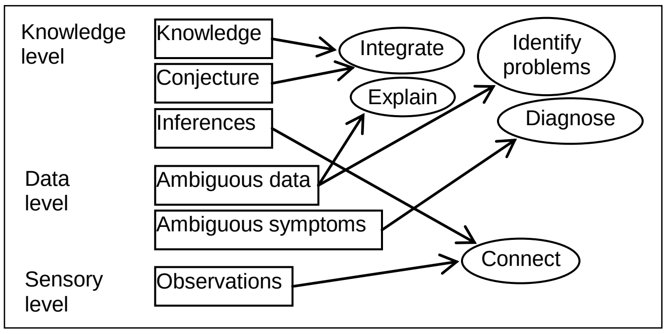


Figure 2. Sensemaking Functions Mapped from [38]

During the sensemaking process various cognitive processes allow a person to simultaneously reason about data and the semantic meanings in that data [38]. When a person successfully makes sense of observable data, that person is said to understand the data as well as the contextual frame of reference of the data.

When a person has inconsistency between data from the task environment and a mental model, that person is faced with the problem of whether to re-evaluate the current understanding of the situation or to maintain the dissonance caused by this inconsistency. Trouble in this integration process often leads to poor decisions and reasoning errors, exemplified by the number of failures leading to George Custer's defeat at Little Big Horn [23]. Previously-held beliefs can also prevent a person from integrating new knowledge, leading to many types of decision biases and predictable errors in decision making [71], [34], [35], [42], [87]. These errors force the person to choose between distrusting the conflicting data, distrusting the data sources, or maintaining an inconsistent belief set.

The process of reconciling and integrating sources of information and data from the environment comes up in many applications of intelligent behavior, but the atomic processes by which humans integrate knowledge and sensory information is still not yet well understood [5] [7].

In Klein's data-frame sensemaking model [38], problem-solvers simultaneously recognize and construct frames from available data and manage current frames of reference. A frame is a representation for a hypothesized mental structure imposed upon data to organize it.

A frame, used in this sense, helps people create constraints on internal reasoning processes which help them reason about their task environments without having to consider all available data or possible states [37], [46]. The processes of managing a frame in the Klein model are forming a frame, elaborating what data is in a frame, questioning an existing frame, and reconstructing a frame. When people "manage frames" they are defining, connecting, and filtering the data they attend to and seek out.

Zhang, et al. [95] present a conceptual model of sensemaking as used in intelligence analysis tasks (Figure 3). This model also 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
3. Accretion, tuning, and restructuring mental models [63]

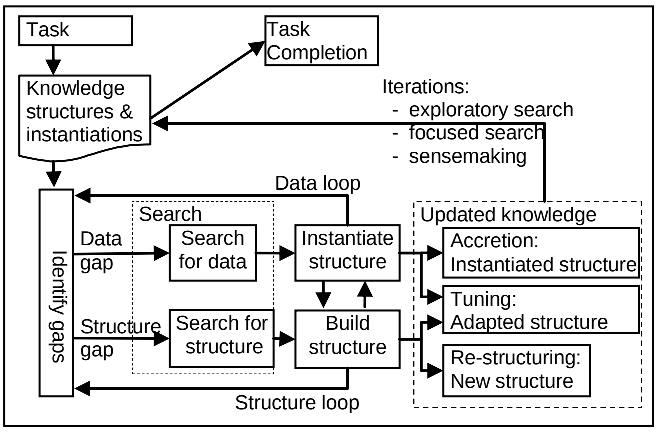


Figure 3. Model of sensemaking (adapted from [95])

The first activity, identifying gaps, involves learning that there are inconsistencies between held knowledge and perceived sensory information. This first activity does not always happen though because of human limitations. For instance, many times people have difficulty appropriately attending to information from the environment which contradicts their previously held mental model [36].

The second activity, seeking information and structure, involves developing hypotheses to account for the disparity, seeking data to build, support, or refute the hypotheses, and seeking a structure to integrate all of the information.

The third activity involves accretion, tuning, and restructuring of the mental model. Accretion is the addition of new information to prior knowledge, tuning is an adjustment to the person's background knowledge that previously existed, and restructuring involves re-evaluating the entire frame of reference. According to the Zhang's model [95], these activities take place until a reasonably explanatory mental model is completed or until the decision task is complete.

A. Representing Mental Models

Empirical studies suggest that problem solvers query and manipulate mental models during task performance [34]. These mental models are internally-stored representations that enable problem solvers to reason about tasks and task environments. Their presence and use is supported by a number of systematic and predictable errors in reasoning which can be detected through controlled experiments [35], [42], [87].

A person's awareness of relevant data in the environment depends on the quality of that person's mental model or background knowledge of the environment [15], [34]. Experts performing a task can dramatically outperform novices at the same problem because of the content and structure of their background knowledge [15]. Because novices lack experience and background knowledge, they have trouble determining which data from the environment is relevant to the task. This is especially pronounced when the task involves processing large amounts of information [94], [95].

1) Procedural Knowledge

The organization of a person's mental model, and thus the ability to determine which data is relevant or significant depends on mentally stored patterns of interaction [56]. Domain-specific patterns of interaction can be sequences of actions or higher-level sequences of more general behaviors [10], [11]. An example of such a pattern of interaction is a person's experience troubleshooting automobile problems. Troubleshooting in the automobile maintenance domain may be similar enough to troubleshooting in other domains that a person may have significant knowledge transfer when troubleshooting in other domains [12].

Other stored mental patterns of interaction constitute domain-dependent patterns or schemas, and may not be transferable outside of the particular task environment. An example of a domain-specific pattern is knowledge of the sequences of key presses that enable a special move, like throwing a Hadoken fireball in the Street Fighter II video game. The knowledge and mastery of many patterns of key presses does not transfer to the mastery of tasks in other domains, though it may transfer to the mastery of other video games involving similar button combinations.

When solving a problem, people can match mentally stored patterns of actions with aspects of the current situation. This matching enables them to predict likely future states, and to develop general task strategies to accomplish the task. Additionally, more robust stored mental models enable people to specify what sequences of inputs or actions are needed to accomplish each function in their current task strategy [10]. In solving a problem, 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 [52]. This indicates that procedural patterns of knowledge are composed of organizations of goals, operators, methods, and selection rules within a domain [9].

Experts at a task tend to have more refined and better organized procedural models about their tasks, which enable them to perform tasks much more quickly than novices. Experts have a more accurate representation of what data elements constitute a meaningful state in the problem space. Experts in a task domain also have better situation understanding and better situation awareness as tasks unfold. They are better at monitoring their progress toward problem solutions and at estimating the difficulty of tasks than novices [15], [70].

2) Declarative Knowledge

Experts also tend to have more refined declarative knowledge models than novices, which allow them to perform tasks more accurately [54]. Declarative knowledge consists of facts about a domain and the organization of those facts. This includes semantic content, such as concepts and facts, and episodic content, which includes knowledge about events, or when and where knowledge was acquired [77], [85].

Declarative knowledge has also been referred to as explicit knowledge [51], information [86], taxonomy or ontology [25]. Declarative knowledge enables people to match data elements and affordances in the task environment with stored procedural patterns [2]. To gauge where they are in the process and select future goals, strategies, and actions, people can use the match between their declarative knowledge of the domain and their stored patterns of interactions or procedural schemas. These matches between the concept space and the procedural space allow them to infer explanations about events that have already happened, predict future states of the task environment, and select goals, operators, methods, and selection rules to perform tasks in the domain environment.

B. *Domain-Specific Knowledge Schemas*

Both declarative and procedural knowledge tend to be organized around memorized patterns [63], [85]. In a famous 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. When novices were given the same task, they could only recall two to three pieces. 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 [15]. 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. Other studies indicate that people reconstruct information from their indexes into episodic memories of an environment [77], [85].

Problem solvers construct, modify, and reconfigure their mental models constantly. In knowledge-lean tasks, which are tasks without a lot of semantic content, problem solvers are able to tune, chunk, and group items to strengthen their associations in memory, and can remember problem states they have seen previously. Through methods like these, they rapidly improve their ability to select task operators with experience [3], [24], [88].

In knowledge-rich tasks, like software reverse engineering, some argue that a finite set of generic procedural tasks can be combined with well-organized knowledge-models to create intelligent behavior in the task [11][12].

V. Eliciting Mental Models

The field of knowledge engineering has produced and refined several methods to extract and represent the mental models and cognitive processes of people solving problems in various work domains [47], [73], [75], [90]. In some cases, the methods are used to uncover the processes underlying human intelligence and in others they are aimed at capturing the knowledge about a domain.

A. *Cognitive Task Analysis*

For either purpose of knowledge elicitation, one method for gaining knowledge about task processes is through a task analysis. The task analysis is aimed at breaking down a complex task into smaller tasks so that each step can be analyzed or so that work or automation supports can be designed to make the work more efficient or cost-effective. Hierarchical decomposition of a task has been found to be useful because it is the means by which humans work with information from complicated problem domains [31]. Hierarchically decomposing a task into its component structures is both a modeling challenge and an in-depth analytical exercise [11].

Background knowledge about a reverse engineering task can be discovered through document analysis [20]. Because of the extensive background knowledge required of reverse engineers, collecting, organizing, and representing factual knowledge about the reverse engineering domain is a large undertaking. One approach to capturing declarative background knowledge is to gather sources, such as textbooks and articles which are used as training and education materials, and specifying the semantics of relevant reference material as an ontology specification in a formal description language [25].

There are limitations with explicitly capturing background knowledge from source texts and training materials. First, extensive ontology modeling is extremely time consuming and potentially error-prone. Second, there are still not practical techniques available to verify and validate large declarative knowledge bases. Third, capturing all of the background knowledge about each of the knowledge sub-domains in reverse engineering might be too much to solve a particular type of task. As a knowledge base gets larger and larger, an agent or automation support tool that reasons over that knowledge base will need to have more intelligent and efficient selection rules to sort and filter through the possibilities entailed by the knowledge base [12], [49].

Cognitive task analysis methods can be used to determine the knowledge that is required for well-defined subset of a cognitive work domain. Often a cognitive task analysis will contain multiple methods which reinforce each to triangulate results and reinforce the reliability of the findings [47]. Depending on the particular methods used, a cognitive task analysis methodology can extract knowledge at a very high level or in great detail.

Subject matter expert interviews are often used in cognitive task analysis to isolate the appropriate terminology and to scope the investigation to relevant subsets of domain tasks [39]. Structured interviews can provide insight and rich qualitative data, which can provide reference knowledge for other task analysis methods. They can also aid in the selection of a prototypical task to guide further investigation.

B. *Verbal Protocol Analysis*

Another procedure often included in task analyses is a verbal protocol analysis. A verbal protocol is a method that involves observing a person performing a task while he or she thinks aloud [21]. When combined with qualitative interview data from experts, this type of analysis allows researchers to form several types of inferences. Researchers can use the recorded verbal data to map participants' actions to concepts the participants use in their task. It can also be used to identify beacons in the task environment, and to infer goals, sub-goals, and reasoning strategies used in the task [59].

Timing data is also very useful. For example, when a participant stops speaking during a think-aloud protocol, this can provide insight into the person's retrieval from long term memory, and can indicate steps in the task that may be more cognitively demanding than others [21].

The number of inputs and outputs a system handles dramatically increases the number of possible states, which can make modeling problem solving in knowledge-rich tasks like reverse engineering challenging. Because of the large state space, enumerating the state space over all available variables is potentially not feasible. For this reason, knowledge engineers must understand what makes up an actual mental representation of a system, what constitutes a state in the task environment, and with what variables those states are constructed.

Researchers can use a verbal protocol to learn about heuristics or short-cuts people use to make faster decisions when their available choices are ambiguous. People develop domain-specific heuristics with experience in a task or through instruction. Heuristics enable people to filter through non-essential information and prune the space of actions or states to those that seem reasonable for the task. Since heuristics are often specific to a problem domain [31], [46], [54], they represent an essential component of background knowledge about the domain [31]. Additionally, learning about problem solving heuristics may provide insight into common or generic tasks which have structure and functional properties that can be exploited with more general representations [11].

VI. Sensemaking in Reverse Engineering

Software reverse engineering involves observing and analyzing a program in order to discover what it does, how it works, and at a higher-level, what the goals of the program's designer were [13]. Tasks in reverse engineering require the ability to quickly integrate background knowledge with the current situation and use a collection of tools, skills, processes, knowledge, logic, and creativity to understand the structure and functionality of complex software systems [17]. It can be seen as a blend of intuition and analytics, where expert reverse engineers can move between both types of reasoning as the need arises.

Some conceive reverse engineering as a process of abstraction from low-level details about software to high-level concepts [22]. Others conceive it as a set of interconnected analysis activities aimed at understanding a program to achieve some goal [17]. Tilley [84] describes reverse engineering as transforming software artifacts into a mental model through "mental pattern recognition" and forming "more abstract system representations." Muller & Kienle [50] describe these abstraction levels as source text, structural, functional, architectural, and application.

While reverse engineers relate observable low-level software representations to higher-level concepts, they also must monitor and seek information at very concrete levels. Descriptions of reverse engineering as simply a process of moving from concrete to abstract representations miss the point that semantic concepts exist at low and high levels of abstraction, and that neither type is reasoned about in isolation. Studies suggest that reverse engineers work at multiple levels of abstraction simultaneously to understand the code and switch between levels of abstraction as their comprehension needs dictate [8], [45], [53], [91].

Reverse engineering involves an active observation process, and its activities closely resemble those from the scientific method. The process of learning about a program is analogous to how a biologist learns about the properties of a cell or a physicist learns about properties of matter.

Sensemaking in reverse engineering requires eliciting information about the program, developing representational frames from data and background knowledge, and integrating representational frames to new data and the person's knowledge [38], [83], [95].

A. *Sensemaking Functions in Reverse Engineering*

Reverse engineers must possess detailed "how-to" knowledge about how to accomplish many different subtasks. At a general level though, the process of reverse engineering has much in common with the process of scientific discovery. These generic functions of reverse engineering programs canonically involve the interleaving of:

- Goal construction
- Planning
- Carrying out a plan (business as usual)
- Generating hypotheses or questions
- Determining needed information
- Experimentation to seek data
- Instrumentation to isolate unavailable data
- Evaluating and integrating
- Updating the mental model

For a mental reference, the top-level of a hierarchical extended finite state machine representation of these functions is shown in Figure 4. In the diagram, each state can be expanded into its own encapsulated state diagram to further outline the processes underlying each state. These functional state diagrams are presented as a framework to organize the cognitive task analysis work and the information that comes from it.

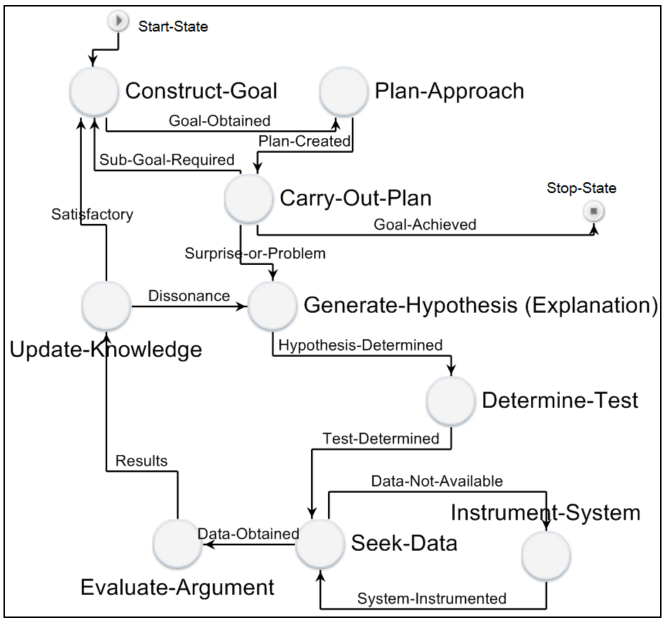


Figure 4. Sensemaking Functions in Reverse Engineering

1) *Goal construction (Construct-Goal)*

One of the first things needed in solving a reverse engineering problem is a goal. In simple planning problems like the blocks world domain, or Towers of Hanoi, it is taken for granted that the goal is present, and that a suitable representation of the goal state exists. In more complex problems, sometimes the goal is not known from the outset or is only known in a very vague way. The goal may come from information in the task environment such as a prompt, or indirectly, such as when the person figures out what should be done by interpreting and synthesizing clues in the task environment.

Because the amount of available data in a reverse engineering task is enormous and exists across many different parts of a system, one of the first steps is to identify or develop specific goals to constrain the analysis. Isolating the overall reverse engineering problem to smaller sub-goals provides a set of constraints that can help reverse engineers filter the data they need to look at.

These goals are many times constructed in situ by the needs that arise in the course of analyzing the reverse engineering problem, planning a course of action, and carrying out the plan. If a person notices things that “look weird” in the program, it is often an indication that the program is performing unexpected(and possibly malicious) functionality which changes the current goal for the reverse engineer, and which may change the reverse engineer’s top-level goal as well.

2) *Planning (Plan-Approach)*

Once the goal is developed and understood, a planning process is developed to construct one or more sequences of actions to move towards completion of the desired objective. One can also construct sequences of actions that are derived backward from the goal state. Attempts to construct partial and fully-specified sequences of actions are all contained under the umbrella of planning [64].

Planning in reverse engineering also involves the selection of alternative courses of action when there are multiple competing ways to accomplish a task or subtask. Creating a plan to get access to all encrypted instructions is an example of a plan in a reverse engineering task, and there are multiple ways to gather that data, each of which requires varying levels of effort, knowledge, skill and time.

3) *Carrying Out a Plan (Carry-Out-Plan)*

Carrying out a plan is the process of following the sequence of planned actions, and is termed “business as usual.” As long as the person can execute the plan, there is no trouble. As the plan is carried out, the person will need to construct sub-goals and sub-plans in order to proceed along a sequence of actions. For this, they may have to go back to the process of goal construction and planning in order to solve intermediate problems along the way.

At some point, the reverse engineer may encounter a problem or experience some sort of surprise which stalls their progress. This could be arriving in an unexpected state in the task environment or perceiving data that indicates a problem in reaching one of their goals (based on prior knowledge [38], [39]. This is essentially where sensemaking processes begin in the overall problem solving process.

4) *Generating Hypotheses (Generate-Hypothesis)*

Reverse engineers also develop questions and hypotheses about the program they are studying [8]. Reverse engineering has been referred to as “the iterative refinement of hypotheses” about a program [84]. Sometimes this may be done implicitly during the task. For instance, a reverse engineer might wonder, “How would the control flow change if I switched the zero flag at this instruction?” or “What will the EAX register hold when this function returns?” Hypotheses can also be explicit, as in hypotheses specified in reverse engineering test plans.

As reverse engineers become more familiar with the patterns of interaction of the system being studied, they can develop on-the-fly hypotheses, like “I think there might be a structured exception handler that the code is invoking which makes the program terminate here.” The recognition and use of this type of domain-specific structural schema in generating hypotheses and questions is characterized as one of the features of a sensemaking process [56].

5) *Determining Needed Information (Determine-Test)*

Once a hypothesis or question is established, there needs to be some way to answer the question or to support or refute the hypothesis. This results in a “data loop” as shown earlier in Figure 3 [56][95]. In the data loop, a hypothesis has been constructed, and a test must be developed to determine whether the hypothesis should be supported, refuted, or whether it should be suspended until further testing can take place.

The test can be generated through a “what-if” type of inference where the hypothesis is assumed to be true, the consequents of that hypothesis being true are also assumed, and then the person determines how to test whether or not those consequents hold. The generation of a hypothesis and the generation of a test are both complicated phenomena which need more investigation before developing an atomic model of generativity. This is a task for future research.

The process to determine the needed information and a test to gather that information can produce facts about the environment, but it can also produce additional concepts or relationships, such as constraints in how data are related. In this way, the function to determine the needed information and to construct a test could account for the “structure loop” in [95] as well as the “data loop.”

6) *Experimenting to Seek Data (Seek-Data)*

Once a reverse engineer has a structure (a set of hypotheses), that structure brings a number of inferences and consequents that should hold if those inferences turn out to be true. Like experimentation in science, reverse engineering involves seeking confirming or contradictory evidence about a hypothesis using data from the program environment.

Once the idea of the test is constructed, the data seeking phase is carried out, which is another planning process, whereby a sequence of actions is constructed that can gather the needed data from the environment. To gather data, reverse engineers have to isolate a behavior of interest, either by providing the program an artificial input like changing a flag on a register, or by observing the behavior in an isolated way.

An example of this is making a plan to test whether or not a program is spawning another process. After the hypothesis is constructed (that the program is spawning another process), and a test is developed based on an inference from a structural understanding of the environment (if the program is spawning another process then another process will be present at some time after the program is run), then a plan can be carried out to gather that data:

1. Run a program to prevent processes from exiting
2. Run the target program
3. Run a separate program to monitor open processes
4. Determine if another process was run

Through this process of interaction, data seeking, and planning in the environment could return information as a result, which can be used to generate new facts in declarative knowledge about the environment, about the structure of elements in the environment, or about more abstract concepts related to elements in the environment.

7) *Instrumentation (Instrument-System)*

If the necessary data is not readily available, a process of instrumentation must take place. Instrumentation refers to the active development of capabilities to gather data about a program or system. In many scientific fields, instrumentation involves the placement and measurement of sensors that measure physical, chemical, or electrical properties.

In reverse engineering tasks, sensors or probes are not always available for use, and often the properties sought are more highly structured than single variables. Because of this, skilled reverse engineers develop their own tools through adapting existing hardware or software tools or programming their own software to solve measurement and information gathering tasks. Programming skill and troubleshooting skills are therefore fundamental capabilities of reverse engineers.

8) *Evaluating and Integrating (Evaluate-Argument)*

Once the reverse engineer has obtained information to fill in gaps in their knowledge, they must interpret the data in order to assess its quality, assess whether it answers the question adequately, and assess whether and what new knowledge should be added to their existing knowledge structures.

In informal experiments (like formal ones), things can go wrong such as when other processes affect the way program behavior represents itself, or when the experiment only works for certain test conditions. The process of evaluation and integration also involves determining if the data or its sources should be trusted. For example, the person conducting a test to see if processes are present may think: “if a rootkit is on my system, it may be hiding the presence of a process from my analysis program.”

Given this evaluation, when a person attempts to update the mental model, they could dismiss the idea or investigate it further if it causes too much dissonance or trouble with other knowledge about the task environment.

9) *Updating the Mental Model (Update-Knowledge)*

Updating the mental model consists of the processes of accretion, restructuring, and tuning knowledge in a mental model outlined in [63] and included in the model in [95]. 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 from the evaluating phase.

Hypotheses, experiments and observations can provide reverse engineers with knowledge about the system, but do not provide a complete model of a large program. With large programs, reverse engineers must quickly discover the most relevant pieces of knowledge about the program. The updated mental model should enable the person to filter through large amounts of data to determine what is relevant better than they were able to during previous loops.

Other major tasks that require an updated or emerging mental model of the program include naming concepts, assigning concepts to locations in the code, and recognizing assigned concept locations [60].

VII. Knowledge Modeling for Reverse Engineering

In order to understand the automation needs that would support reverse engineering tasks, it is important to develop knowledge requirements for those tasks. Knowledge is typically divided into procedural and declarative knowledge, which serve different purposes. Declarative knowledge is the explicit encoded knowledge that can be recalled and recited. This consists of concepts in the domains, facts, and facts about how concepts relate to each other. On the other hand, procedural knowledge is composed of *how-to* knowledge that allows a person to perform a task when they come across a situation.

A. *Modeling Declarative Knowledge*

Von Mayrhauser and Vans [91] studied programming knowledge and divided what is arguably declarative knowledge into separate conceptual models for the program, the situation, the domain (what they called the top-down model), and background knowledge. In the top-down model, programmers chunk information about program, instructions, modules, the problem domain, and theorized goals and plans of the programmer, and synthesize it all into their existing knowledge base. The knowledge base includes schemas of concepts and concept families that are used in developing the program model, situation model, and domain model of a program [53], [91].

The declarative knowledge that a reverse engineer must understand is extensive. Software reverse engineering requires knowledge of thousands or hundreds of thousands of facts about how code is written, compiled, linked, and loaded; the operating system which manages running programs; the processor architecture on which the program and operating system runs; and sometimes other hardware, firmware, and input/output devices that can be involved or invoked. These facts provide the reverse engineer the ability to make inferences about how changes made in one part of the system can affect other parts of the system.

Since the knowledge required by a reverse engineer is broad as well as deep, reverse engineers’ declarative knowledge models will likely be very organized and sophisticated. To model and simulate problem solving (and sensemaking) in reverse engineering, this extensive base of knowledge will need to be considered by either limiting the scope of the model or expanding its encoded knowledge. Some sub-domains of declarative knowledge are shown in Table 1.

The declarative knowledge models of reverse engineers help them determine what elements in the environment are relevant to their task. Declarative knowledge also provides reverse engineers the frame of reference in their planning tasks and helps them determine what data to seek from the environment, how to develop a hypothesis, how to isolate the needed data, and how to interpret the results of their experimentation.

Debugging and troubleshooting skills
Computer programming
Program compilation and interpretation
Object file linking and executable loading
Program execution
File and file header formats
Library, API, and operating system calls and internals
Assembly language (multiple instruction sets)
Firmware and hardware
Processor structure and function
Communication protocols
Anti-debugging and anti-reverse engineering tricks

Table 1. Declarative Knowledge Sub-Domains in Reverse Engineering

In addition to factual knowledge, reverse engineers’ declarative knowledge models contain causal relationships about the data, which can be represented as constraints on existing knowledge, or as predicate statements on constant variables. The causal relationships may take the form of an abstract cause and an abstract effect and concrete instantiations of the causal concept and the effect concept. Sensemaking functions can exist across different levels of sensory cues, information, and background knowledge (Figure 5).

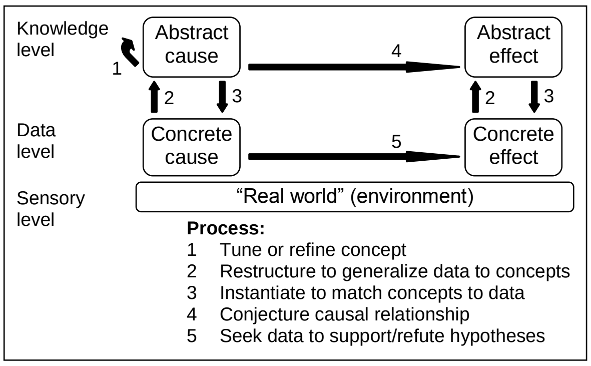

Knowledge level
Abstract cause
Abstract effect
Data level
Concrete cause
Concrete effect
Sensory level
"Real world" (environment)
Process:
1 Tune or refine concept
2 Restructure to generalize data to concepts
3 Instantiate to match concepts to data
4 Conjecture causal relationship
5 Seek data to support/refute hypotheses

Figure 5. Integrating cause and effect

Reverse engineers' declarative knowledge models of a program as it runs are augmented with structural information from the environment, including the layout of memory, patterns representing programming constructs; functional information, such as input and output system calls used by the program; and behavior information such as whether the program writes to disk or communicates over the network. Many types of structural information can be represented either in terms of cause and effect or in terms of concepts in the domain.

The set of concepts in a task domain is a central component to a person's declarative knowledge [63]. A concept represents a template or class which describes a set of objects or entities. Often the template concept is based on an object's categorization with a group or its similarity to a prototypical member of that group. Categorizations among concepts are often defined by shared attributes between similar or related entities [64].

A schema is a grouping of concepts by shared or similar attributes that captures sets of relations and attributes that an entity might have [4]. In a schema, data members are organized by their attributes, and new entities are classified into categories based on how they match the attributes of available categories [55], [63].

As discussed earlier, a frame is another representation that is used to organize background knowledge [38], [48]. A frame is a "network of nodes and relations" that represents a person's background knowledge of a domain. These frames can be organized into a frame system which identifies how the knowledge is structured. Each frame is made up of a number of slots which are blank or unfilled attributes. The slot may contain both a value, and a slot type which specifies what kind of data is able to occupy that slot.

Other means to organize representations of declarative knowledge are categories, composite objects, components (especially concerning systems or devices), networks, and conceptual constructs [4], [12], [49], [64]. Each of these representations has its own formalism for encoding knowledge, but each serves as a way for people to formally represent mental objects and their relations to other objects.

One example to illustrate a potential formalism for encoding knowledge from a task analysis is a model in the ACT-R cognitive architecture [1]. In the ACT-R theory (which is contained in the ACT-R software), knowledge in memory is divided into declarative knowledge and procedural knowledge representations.

In ACT-R, declarative knowledge is composed of chunks, which can be retrieved from long-term or short-term memory. Chunks are pieces of memory, which have a chunk-type and a set of slot and value pairs. A chunk-type is a template for a chunk which declares what types of slots can be held by a chunk of that type. A slot is an attribute which describes some property of a mental object or entity. A chunk's slot can take on various values, and when it does so it represents specific knowledge about an entity's attributes.

In Figure 6, a chunk-type is declared called register, and all registers can have a number-bits slot and a register-type slot. After that, three chunks are added to ACT-R's declarative memory module which represent the EAX, AX, and CR0 registers on the Intel x86 processor architecture.

```
(chunk-type register number-bits type)

(add-dm
    (eax ISA register
       number-bits 32
       type general-purpose)
    (ax ISA register
       number-bits 16
       type general-purpose)
    (cr0 ISA register
       number-bits 32
       type control-register))
```

Figure 6. Declarative knowledge in ACT-R

The ACT-R declarative knowledge does not model formal domain property constraints as would be the case with formal description logic or first order logic. However, through the combination of encoded declarative knowledge and encoded production rules, sophisticated task behavior has been modeled in various task and problem-solving domains [2].

B. *Modeling Procedural Knowledge*

Procedural knowledge in ACT-R is defined by production rules, which are patterns consisting of a left-hand side and a right-hand side. The left-hand side of a production rule contains a set of state conditions which represent a pattern of state variables to be matched by the agent performing the task.

The right-hand-side of a production rule represents a set of actions to be accomplished by the agent. While an agent is engaged in a problem-solving task, if the state of the environment meets one of the sets of conditions defined on the left-hand side of a production, the actions on the right-hand side of that production fire. The action side of a production often changes the state of the variables in one or more slots in the chunk listed on the action (or right-hand-side) of the production (Figure 7).

Essentially productions in ACT-R create a state-machine representation of a problem-solving process. Particular productions in a cognitive architecture like ACT-R represent states in an extended finite state machine and the condition rules and patterns of production firing represent the transition between those states. Modeling tasks in a cognitive architecture provide the modeler a way to simulate and predict human behavior in a cognitive task, and often seek to be "cognitively plausible" rather than optimal [2].

```
(p follow-jmp
   =goal>
      isa              goal
      state            following-jump
      target           =destination
==>
   +goal>
      isa              goal
      current-instr    =destination)
```

Figure 7. A simple ACT-R production rule

Modeling cognitive tasks at the level of detail required by ACT-R requires many commitments to architectural and attentional details which may not be required in other production-based expert systems or artificial intelligence frameworks. However, it has the advantage of being able to model a task in a way that allows the investigation of how humans actually perform the task.

VIII. Conclusion

This paper provided an overview of the current state of research in sensemaking, situation understanding, and situation awareness and how these concepts can be applied to the problem of reverse engineering. This paper also offered an overview of a structured cognitive task analysis methodology to enable elicitation of knowledge that reverse engineers use in daily practice. It then described knowledge representation formalisms suited to the modeling of cognition in context, namely the ACT-R cognitive architecture. Finally, it presented a background on various processes in software reverse engineering in the context of the sensemaking components that those tasks levy on individuals.

Modeling and simulating intelligent behavior in complicated, knowledge-rich tasks like software reverse engineering is a challenging undertaking, but the steps here presented can enable the cyber security and modeling and simulation communities to produce more robust models of intelligent reverse engineering behavior. These steps include isolating the domains of interest, isolating the knowledge required in those domains, verifying the knowledge needed to complete reverse engineering tasks, and then developing and refining formal models of conceptual and procedural knowledge, such as ontologies and production rules, until they can handle and simulate intelligent behavior in the task.

Areas for future research that would improve researchers' ability to model problem-solving behavior in knowledge-rich tasks like software reverse engineering include:

- The development of better tools and methods to automate the extraction, verification, and validation of declarative knowledge from documents
- Streamlined methodologies to extract and represent relevant declarative and production knowledge from examples of task behavior
- Better and more refined development processes to model declarative and procedural knowledge in complex task environments

With the current state of cyber security and the constant threat from cyber attacks, increasingly intelligent tools and automated assistance are more important than ever. Executable reverse engineering models could feasibly help engineers test new software before it is released to the world, provide intelligent assistance to those performing security-related work, or provide intelligent tutors that are sensitive to the needs of new reverse engineers and cyber security professionals.

References

[1] J. Anderson and C. Lebiere. *The Atomic Components of Thought*, Lawrence Erlbaum Associates, 1998.

[2] J. R. Anderson. *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press, New York. 2007.

[3] M. E. Atwood and P. G. Polson. "A process model for water jug problems", *Cognitive Psychology*, 8 (2), pp. 191-216, 1976.

[4] L. W. Barsalou, *Frames, Concepts, and Conceptual Fields*, Lawrence Elbaum Associates, 1992.

[5] B. Birrer, R. Raines, R. Baldwin, M. Oxley, and S. Rogers. "Using Qualia and Hierarchical Models in Malware Detection", *Journal of Information Assurance and Security*, 4, pp. 247–255, 2009.

[6] B. Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Wordware, 2009.

[7] R. A. Brooks. "Elephants don't play chess", *Robotics and Autonomous Systems*. (6) 1-2. pp. 3-15, 1990.

[8] R. Brooks. "Towards a theory of the comprehension of computer programs", *International Journal of Man-Machine Studies*, 18, pp. 543-554, 1983.

[9] S. K. Card and T. P. Moran. *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, 1983.

[10] J. M. Carroll and J. R. Olson. "Mental Models in Human-Computer Interaction", in M. Helander (Ed.), *Handbook of Human-Computer Interaction*, pp. 45-65, Elsevier Science Publishers, 1988.

[11] B. Chandrasekaran. "Generic tasks as building blocks for knowledge-based systems: The diagnosis and routine design examples", *The Knowledge Engineering Review*, (3) 3, pp. 183-210, 2009.

[12] B. Chandrasekaran and T. R. Johnson. "Generic tasks and task structures: History, critique and new directions", *Second Generation Expert Systems*, pp. 239-280, 1993.

[13] E. J.Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*. pp. 13-17. 1990.

[14] B. Crandall and K. Getchell-Reiter. "Critical decision method: A technique for eliciting concrete assessment indicators from the intuition of NICU nurses", *Advances in Nursing Science*, (16) 1. p. 42, 1993.

[15] A. D. DeGroot. *Thought and Choice in Chess*. Amsterdam University Press, 1965.

[16] S. R. Dixon, C. D. Wickens and D. Chang. "Mission control of multiple unmanned aerial vehicles: A workload analysis", *Human Factors*, (47) 3, p. 479, 2005.

[17] E. Eilam. *Reversing: Secrets of Reverse Engineering*, Wiley, 2005.

[18] M. R. Endsley. "Measurement of situation awareness in dynamic systems", *Human Factors*, 37, 65-84, 1995.

[19] M. R. Endsley and M. D. Rodgers. "Situation awareness and information requirements analysis for en route air traffic control", *Human Factors and Ergonomics Society Annual Meeting Proceedings*, (38) 1. pp. 1071-1813, 1994.

[20] R. S. Englemore and E. Feigenbaum. "Expert Systems and Artificial Intelligence", in JTEC Panel on Knowledge-Based Systems in Japan, May 1993.

[21] K. Ericsson and H. Simon. *Protocol Analysis: Verbal Reports as Data* (Rev. ed.). MIT Press, Cambridge, MA, 1993.

[22] G. C. Gannod and B. H. C. Cheng. "A framework for classifying and comparing software reverse engineering and design recovery techniques", *Proceedings of the Sixth Working Conference on Reverse Engineering*, p. 77, 1999.

[23] D. C. Gompert and R. L. Kugler. "Custer in cyberspace", National Defense University Center for Technology and National Security Policy, Washington, D. C., 2006.

[24] J. G. Greeno. "Hobbits and orcs: Acquisition of a sequential concept", *Cognitive Psychology*, 6 (2). pp. 270–292, 1974.

[25] N. Guarino. "Formal Ontology and Information Systems", *Proceedings of FOIS'98*, pp. 3-15, IOS Press, Amsterdam, 1998.

[26] HBGary, Inc. Flypaper. www.hbgary.com/free-tools1980

[27] J. Hennessy, D. Patterson, and D. Goldberg. *Computer Architecture: a Quantitative Approach*, Morgan Kauffman, Inc., 2003.

[28] Hex-Rays, Inc. The IDA Pro Disassembler and Debugger. www.hex-rays.com/idapro/

[29] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[30] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.
[31] P. Huang and K. Sycara. “Learning from and about the opponent”, in A. Kott and W. McEneaney (eds), *Adversarial Reasoning: Computational Approaches to Reading the Opponent’s Mind*, Chapman & Hall/CRC, 2007.
[32] ImpRec, The World’s Most Famous IAT Rebuilder Tool. http://www.woodmann.com/collaborative/tools/index.php/ImpREC
[33] Intel Corporation. *IA-32 Intel Architecture Software Developers Manual*, Intel Corporation, 2001.
[34] P. N. Johnson-Laird. *Mental Models: Towards A Cognitive Science of Language, Inference and Consciousness*, Harvard University Press, 1983.
[35] P. N. Johnson-Laird. *How We Reason*, Oxford University Press, USA. 2006.
[36] D. Kahneman *Attention and Effort*. Prentice-Hall Inc., 1973.
[37] C. Kemp. *The Acquisition of Inductive Constraints*, PhD Dissertation, MIT, 2007.
[38] G. Klein, J. K. Phillips, E. L. Rall and D. A. Peluso. “A data-frame theory of sensemaking”, *Expertise out of Context: Proceedings of the Sixth International Conference on Naturalistic Decision Making*, pp. 113–155, 2003.
[39] G. Klein. “Developing expertise in decision making”, *Thinking & Reasoning*, (3) 4, pp. 337-352, 1997.
[40] G. Klein, B. Moon and R. R. Hoffman. “Making sense of sensemaking: alternative perspectives”, *IEEE Intelligent Systems*, (21) 4, pp. 70–73, 2006.
[41] C. Kruegel, W. Robertson, F. Valeur and G. Vigna. “Static disassembly of obfuscated binaries”, *Proceedings of the 13th USENIX Security Symposium*, pp. 255-270. 2004.
[42] N. Y. L. Lee and P. N. Johnson-Laird. “Synthetic Reasoning and the Reverse Engineering of Boolean Circuits”, *Proceeding of the Twenty-Seventh Annual Conference of the Cognitive Science Society*, pp. 1260-1265, 2005.
[43] S. Letovsky. “Cognitive processes in program comprehension”, *Empirical Studies of Programmers*, pp. 58-79, Ablex Publishing. Norwood, NJ, 1986.
[44] LordPE, available at http://www.woodmann.com/collaborative/
[45] tools/index.php/LordPE.
[46] D. Mellado, E. Fernandez-Media and M. Piattini. “A common criteria based security requirements engineering process for the development of secure information systems”, *Computer Standards & Interfaces*, (29) 2. pp. 244-253, 2007.
[47] Z. Michalewicz and D. Fogel. *How to Solve It: Modern Heuristics*, Springer-Verlag, New York, 2004.
[48] L. Militello and R. Hutton. “Applied cognitive task analysis (ACTA): a practitioner's toolkit for understanding cognitive task demands”, *Task Analysis*, pp. 90-113, 2000.
[49] M. A. Minsky. “Framework for Representing Knowledge”, MIT Artificial Intelligence Lab Technical Report, 1974.
[50] K. Morik, B. B. Kietz, W. Emde and S. Wrobel. *Knowledge Acquisition and Machine Learning*, Morgan Kaufmann Publishers, Inc., 1993.
[51] H. A. Muller and H. M. Kienle. “A small primer on software reverse engineering”, Technical Report, University of Victoria, 2009.
[52] I. Nonaka and H. Takeuchi. “The knowledge-creating company”, *Harvard Business Review: The Best of HBR*, www.hbr.org, 1991.
[53] D. Norman. “Some observations on mental models”, in *Mental Models*, Lawrence Erlbaum Associates, Inc., 1983.
[54] N. Pennington. “Stimulus structures and mental representations in expert comprehension of computer programs”, *Cognitive Psychology*, 19, pp 295-341, 1987.
[55] J. D. A. Petrowski and P. S. E. Taillard. *Metaheuristics for Hard Optimization*, Springer, 2006.
[56] J. Piaget. *The Child’s Construction of Reality*, Routledge, 1954.
[57] P. Pirolli and S. Card. “The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis”, *Proceedings of the International Conference on Intelligence Analysis*, pp. 2-4, 2005.
[58] P. Porras, H. Saidi and V. Yegneswaran. “A multi-perspective analysis of the Storm (Peacomm) worm”, SRI International, Computer Science Laboratory Technical Report, 2007.
[59] P. Porras, H. Saidi and V. Yegneswaran. “An analysis of Conficker’s logic and rendezvous points”, SRI International, Computer Science Laboratory Technical Report, 2009.
[60] M. I. Posner. *Foundations of Cognitive Science*. The MIT Press, MA, 1993.
[61] V. Rajlich. “Intensions are a key to program comprehension”, *Proceedings of the International Conference on Program Comprehension (ICPC’09)*, pp. 1-9, 2009.
[62] P. Rohatgi. “Side-channel attacks”, *Handbook of Information Security*, Wiley, Inc., pp. 241, 2006.
[63] E. M. Roth, D. D. Woods and H. E. Pople Jr. “Cognitive simulation as a tool for cognitive task analysis”, *Ergonomics* (35) pp. 1163-1198, 1992.
[64] D. Rumelhart and D. Norman. “Representation in Memory”, University of California, San Diego La Jolla Center for Human Information Processing, Technical Report, 1983.
[65] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, Inc., 2003.
[66] N. B. Sarter, D. D.Woods and C. E. Billings. “Automation surprises”, *Handbook of Human Factors and Ergonomics*, (2), pp. 1926-1943, 1997.
[67] N. B. Sarter and D. D. Woods. “Situation Awareness: A critical but ill-defined phenomenon”, *International Journal of Aviation Psychology*, 1, pp. 45-57, 1991.
[68] N. B. Sarter and D. D. Woods. “Autonomy, Authority, and Observability: The Evolution of Critical Automation Properties and Their Impact on Man-Machine Coordination and Cooperation”, *Proceedings of the 6th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design, and Evaluation of Man-Machine Systems*, Cambridge, MA. 1994.
[69] N. B. Sarter and D. D. Woods. “Pilot Interaction with Cockpit Automation: Operational Experiences with the Flight Management System”, *International Journal of Aviation Psychology*, 2(4), 303-321, 1992.
[70] N. B. Sarter and. D. D. Woods. “Pilot Interaction with Cockpit Automation II: An Experimental Study of Pilots' Model and Awareness of the Flight Management and Guidance System”, *International Journal of Aviation Psychology*, 4(1), 1-28. 1994.
[71] A. Schoenfeld and D. Herrmann. “Problem perception and knowledge structure in expert and novice mathematical problem solvers”, *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8 (5), pp. 484–494, 1982.
[72] B. Schwarz, S. Debray and G. Andrews. “Disassembly of executable code revisited”, *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 45-54, 2002.
[73] C. R. Schwenk. “Cognitive simplification processes in strategic decision-making”, *Strategic Management Journal*, (5) 2, pp. 111-128, 1984.
[74] T. L. Seamster, R. E. Redding, J. R. Cannon, J. M. Ryder and J. A. Purcell. “Cognitive task analysis of expertise in air traffic control”, *The International Journal of Aviation Psychology*, (3) 4, pp. 257-283, 1993.
[75] A. Silberschatz, P. Galvin and G. Gagne. *Operating System Concepts*, Addison-Wesley, New York, 1998.
[76] H. Simon and C. Kaplan. “Foundations of cognitive science”, in M. I. Posner (Ed.), *Foundations of Cognitive Science*, pp. 1-47, 1993.
[77] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena. “BitBlaze: A new approach to computer security via binary analysis”, *Information Systems Security*, pp. 1-25, 2008.
[78] S. Sparks, S. Embleton, R. Cunningham and C. Zou. “Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting”, *Proceedings of Computer Security Applications Conference*, pp. 477-486, 2007.
[79] L. R. Squire. *Memory and the Brain*, Oxford University Press, New York, 1987.
[80] M. Sutton, A. Greene and P. Armini. *Fuzzing: Brute Force Vulnerability Discovery*, Addison-Wesley Professional, 2007.
[81] P. Szor. *The Art of Computer Virus Research and Defense*, Addison-Wesley Professional, 2005.
[82] R. M. Taylor. “Situation awareness rating technique (SART): the development of a tool for aircrew systems design”, in *Situational Awareness in Aerospace Operations* (Ch 3), Neuillysur-Seine, France, NATO-AGARD-CP-478, 1990.
[83] Y. J. Tenney, M. J. Adams, R. W. Pew, A. W. Huggins and W. H. Rogers. “A principled approach to the measurement of situation awareness in commercial aviation”, NASA contractor report 4451, Langley Research Center, 1992.
[84] T. Tiemens. “Cognitive models of program comprehension”, Software Engineering Research Center Technical Report, 1989.
[85] S. Tilley. “A reverse-engineering environment framework”, Carnegie Mellon University Technical Report CMU/SEI-98-TR-005, 1998.

[86] E. Tulving. “Episodic and Semantic Memory”, in *Organization of Memory*, Academic Press, 1972.
[87] I. Tuomi. “Data is more than knowledge: implications of the reversed knowledge hierarchy for knowledge management and organizational memory”, *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, 1999.
[88] A. Tversky and D. Kahneman. “Judgment under uncertainty: Heuristics and biases”, in *Judgment and Decision Making: An Interdisciplinary Reader*, p. 35, 2000.
[89] K. VanLehn. “Cognitive skill acquisition”, *Annual Review of Psychology*, (47), pp. 513-539, 1996.
[90] I. Vessey. “Expertise in debugging computer programs: a process analysis,” *International Journal of Man-Machine Studies*, (23), pp. 459-494, 1985.
[91] K. J. Vincente. *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*, Lawrence Erlbaum Associates, 1999.
[92] A. von Mayrhauser and A. M. Vans. “From code understanding needs to reverse engineering tool capabilities”, *Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering (CASE’93)*, 1993.
[93] K. Weick. *Sensemaking in Organizations*, Sage Publications, Inc., 1995.
[94] Wireshark, www.wireshark.org.
[95] D. D. Woods, E. S. Patterson and E. M. Roth. “Can we ever escape from data overload? A cognitive systems diagnosis”, *Cognition, Technology, & Work,* (4) 1, pp. 22-36, 2002.
[96] P. Zhang, D. Soergel, J. Klavans and D. Oard. “Extending sense-making models with ideas from cognition and learning theories”, *Proceedings of the American Society for Information Science and Technology*, 45 (1), pp. 23, 2008.

Author Biographies

Adam R. Bryant is a PhD candidate at the Air Force Institute of Technology performing research studying the mental models of software reverse engineers. He received his MS in computer science and an MS in information resource management from the Air Force Institute of Technology in 2007. He received a BS in social psychology from Park University in 2001. He spent nine years as an active duty service member in the U.S. Air Force and currently works as a research scientist at the Air Force Research Laboratory in Dayton, Ohio.

Robert F. Mills received his PhD in electrical engineering from the University of Kansas in 1994, his MS in electrical engineering from the Air Force Institute of Technology in 1987, and his BS in electrical engineering from Montana State University in 1983. His research interests are in network management and security, insider threat mitigation, and mission assurance. He is a member of Eta Kappa Nu and Tau Beta Pi, and is a Senior Member of the IEEE.

Gilbert L. Peterson received his PhD in computer science from the University of Texas Arlington in 2001 and a BS in architecture from the University of Texas Arlington in 1995. His research interests include digital forensics, insider threat mitigation, and artificial intelligence.

Michael R. Grimaila received his PhD in computer engineering from Texas A&M University in 1999. His research interests include cyber incident detection, mission assurance, network management and security, information warfare, and systems engineering. He is a member of the ACM, Eta Kappa Nu, ISACA, ISC2, ISSA, Tau Beta Pi, and he is a Senior Member of the IEEE.