Modeling Information Flow for an Autonomous Agent to Support Reverse Engineering Work

Zachary D. Sisco, Patrick P. Dudenhofer, and Adam R. Bryant

Abstract

Reverse engineering is a cyber defense task used to investigate malware, reconstruct functionality of compiled software, and identify vulnerabilities from closed-source software code already being used in operational contexts. While research in this area has mainly focused on techniques to extract information from binary code, it is also important to understand the capabilities and limitations of the human involved in the reverse engineering process (both defensively and offensively) so we can design better information representations and effectively allocate appropriate tasks to autonomous agents. In this paper, we describe our introductory work in developing agent models of reverse engineering. We review what is known about reverse engineers' mental models, then describe and characterize four human-computer interaction patterns involved in reverse engineering from a cognitive task analysis. Finally, we present a category theoretic model to describe how reverse engineers trace information flow when performing static analysis. Our approach is a first step in modeling, simulating, and optimizing the human interaction components of these tasks to increase the speed, scale, and accuracy of cyber defense efforts.

Index Terms

reverse engineering, program comprehension, sense-making, behavior modeling

I. Introduction

In the domain of cyber defense, software reverse engineering is a complex problem. The threat of cyber attacks from rogue agents, competing corporations, and hostile nation-states has been continually increasing along with the number, severity, and sophistication of these attacks. As cyber security measures are deployed, attackers quickly reverse engineer them to create additional cyber attack vectors. The state of cyber security reflects the Red Queen effect—more and more resources are being expended to stay ahead of increasingly sophisticated and dangerous cyber attacks against larger and more prevalent computer systems.

Automated tools are desperately needed to keep up with emerging cyber security threats, to discover vulnerabilities in software, and to assess the liabilities of closed-source software. The scope of the problem is beyond what manual analysis and detection solutions can handle.

Cyber defense analysts perform reverse engineering to understand binary code through observation and analysis of low-level instructions stripped of semantic information. Through the use of specialized tools, a reverse engineer interacts with an executable program and gathers information about its behavior. Because of the scale and scope of cyber attacks, there is a need to develop better processes and tools that allow reverse engineers to understand binary code more quickly and accurately. Many automated tools already exist to extract and represent information from binary files, but what is needed is a way to model the most challenging part of reverse engineering: actually comprehending the code itself and communicating that comprehension to others. Although the number of inputs and complexity of the comprehension problem means that any results will be necessarily incomplete or heuristic, modeling, simulating, and optimizing this process can improve the speed and scale of standard classification and analysis techniques, reduce learning time for new cyber defense analysts, and increase the identification and analysis throughput for the thousands upon thousands of potential software vulnerabilities and their corresponding exploiting malware.

To understand comprehension in reverse engineering, Bryant proposed a theory of sensemaking in reverse engineering that involves interactive mental model construction [1]. This theory describes how an agent (human or automated) forms and manipulates goals, concepts, state representations, and mental

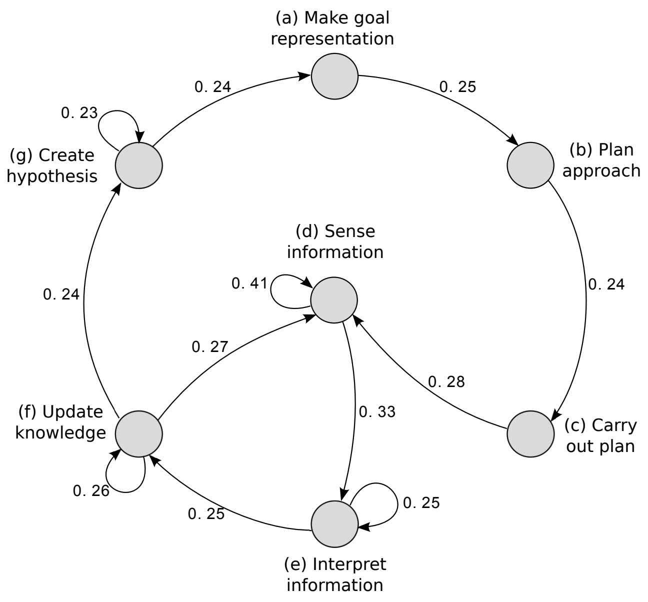

Fig. 1. Interactive Sensemaking Process with Example Transition Probabilities

representations by interacting with information from reverse engineering tools (Figure 1). Bryant's theory represents reverse engineers' sensemaking process through goal-directed information seeking. There are many open areas of research required to apply the theory to modern cyber defense problems, one of which is the focus of this paper: *the need to cast the sensemaking problem in mathematical formalisms*. To formalize the sensing and interpretation activities of the theory, we developed a model of information flow to guide an agent's goals, actions, and hypotheses.

Our overarching research goal is to understand and model how people make sense of complex information so we can improve automation, interaction, and training. Once we understand this process, we can use this information to design automation, intelligent assistants, and information presentations that help cyber defense analysts perform their tasks in less time and with less effort.

Our general research process involves:

1) Investigating the task (analyzing human performance data),
2) Modeling performance of the task,
3) Simulating and analyzing the model's performance, and
4) Optimizing the task through improved training, automation, and interfaces.

This requires us to be able to simplify, understand, generalize, and adapt the sensemaking theory into an executable cognitive model. To do this, we have to decompose the reverse engineering task into simpler *sensemaking* components and develop data structures and algorithms to model a reverse engineer's mental model construction.

Our contribution is a methodology to model how information flows between reverse engineers and their task environments. Since the work presented here is introductory, we made the decision to focus on modeling the 'base case'—analyzing unprotected assembly code. We treat this as the base case because it is a task involved in other, more complex tasks such as vulnerability discovery or malware analysis. Because of this decision, scenarios with obfuscated or protected binaries are excluded from scope. It is our goal that by focusing on small code examples and designing our model with composability in mind, the small-scale components presented in this work can be composed together to simulate more complex tasks.

We first provide an overview of what is known about the content and structure of reverse engineers' mental models and describe four interaction patterns used by analysts as they interact with reverse engineering tools: navigation, translation, experimentation, and elaboration. Finally, we present a knowledge representation data model that represents an executable program and its attributes in terms of information types (Section 3) and our information flow model of the interaction pattern of *experimentation* (Section 4).

II. Reverse Engineers' Mental Model Construction

While recent reverse engineering research centers around developing software tools to automatically extract information from binary code, discover bugs, and handle anti-analysis techniques in malware [2], [3], [4], [5], [6], [7], there has been very little work to understand the cognitive work involved with reverse engineering. Tools such as program disassemblers, hexadecimal editors, decompilers, de-obfuscators, unpackers, and network packet sniffers are often created ad-hoc by reverse engineers for specific tasks and have become known for being difficult to use and failing to take into account human factor design considerations [8], [6].

Software reverse engineering tools provide the task environment in which the reverse engineering activities are situated. Some of the information representations presented by a typical reverse engineering task environment include assembly instructions, data bytes, control flow information, and functional information [1]. Reverse engineers work backward from these representations to undo the compilation and assembly processes that created the executable program originally [9].

A. Executable vs. Source Code Representations

Other work involving the cognitive aspects of understanding programs investigated how reverse engineers make sense of source code representations. Our research, however, studies the mental models involved with understanding binary code. Modern reverse engineers analyze executable files, processes on disk, program artifacts, and secondary information (such as instruction traces or function traces collected through dynamic analysis) that lack contextual information like variable names, data structures, and function signatures found in source code [10], [11], [12], [2], [13], [14]. This means reverse engineers rely on interaction and experimentation with the program to develop concepts about the program.

B. High-level Processes

Reverse engineering executable programs is an interactive activity, which requires gathering and interpreting information from reverse engineering tools, using tools to manipulate various parts of the system, and creating a mental model of the program from this interaction [1]. Some interactive processes involved in reverse engineering are:

- Program analysis – reading and understanding source code,
- Plan recognition – inferring design intent [15],
- Concept assignment – matching programming constructs to code locations [16],
- Re-documentation – creating new descriptions of the program's behavior, and
- Architecture recovery – recreating the program's structure and functional organization [17].

Of these processes, program analysis is the phase in which reverse engineers build up their mental representations of a program by reading program code [18]. In program analysis, reverse engineers both consume and produce knowledge about software through a number of cognitive activities:

- Putting together a hierarchical structure of the program,
- Mapping code areas to the goals of the program,
- Learning and recognizing recurring patterns,
- Connecting pieces of knowledge gathered from exploring the code, and
- Grounding concepts in the program's source code [19].

Program analysis involves abstracting low-level data into high-level concepts [20] and using mental models of concepts [16], programming plans [15], and control flow representations to connect the various pieces of the environment and learned information together. Comprehension is a necessary step in accomplishing this task [21].

C. Interaction Patterns

Reverse engineers investigate programs through four major interaction patterns:

- *Navigation* – finding items of interest in the code,
- *Translation* – interpreting how code would be represented in a higher-level language,
- *Experimentation* – following the program's execution to see how data values change, and
- *Elaboration* – determining the components of a program and explaining the properties of the program in terms of these components.

Navigation involves maneuvering through a program's code to learn what is there or to find some particular item, object, feature, or location, such as a location where a text string is read and manipulated by the program. Navigation is sometimes aided by visual features (or beacons) such as a memorable sequence of instructions a person remembers as being relevant to a program's behavior [22].

Translation is the process of interpreting the higher-level *meaning* of small sequences of instructions. In some cases this means creating representations of instruction sequences in a higher-level programming syntax such as C or Python. In other cases, it involves annotating the code with text comments or drawing diagrams to represent the program's operation.

Experimentation involves single-stepping through a disassembled program in a debugger and monitoring changes in the program's values as the program executes. Experimentation activities have been investigated in other studies such as Vessey et al. [23] in which participants traced changes in the values of variables (in source code) by stepping through the program's statements and mentally simulating the effects of each statement. Our notion of program experimentation is very similar to Vessey's description of *tracing*, but reverse engineers also perform larger scale experimentation in which they run the program from breakpoint to breakpoint to piece together a mental picture of how the program changes along the way. The participants in Bryant's cognitive task analysis described piecing together a picture of the program's functionality as a series of higher-abstraction "events" or "behaviors" that the program performed in between breakpoints [1].

Elaboration is the process of discovering the objects, attributes, and relationships that should make up the person's mental model of the program. Elaborations take the form of abduction-based inquiry [24] in which a person:

- Makes a conjecture about the meaning of an object or relationship,
- Gathers information to confirm or refute the conjecture,
- Evaluates how the information integrates with the person's current mental model, and
- Decides to accept, reject, or remain unsure about the conclusions.

Elaboration is performed iteratively throughout the process of reverse engineering as the person seeks to harmonize his or her mental model of the program with the information received from the reverse engineering tools.

The navigation, translation, and experimentation activities all serve to support elaboration of the components of a program, the attributes of those components, and how the components and attributes are related. Through these and other activities, reverse engineers incrementally learn what programs do, how they are structured, and how they work. They can be considered strategies for the reverse engineers to obtain a "situation model" to mentally represent the important elements in the program.

D. Mental Representations

Reverse engineering work involves interaction of a human-machine system that consists of:

- *The system* – the program, hardware, and software elements,
- *The interface* – information and affordances in the reverse engineering tools,
- *Working memory* – elements in the person's attention,
- *Background knowledge* – the person's stored semantic and episodic knowledge, and
- *The situation* – events, actions, behaviors, and inferences about what is currently taking place [25].

This paper deals with all of these elements of the system, but primarily with the components that make up the reverse engineer's model of a *situation*.

Bryant [26] analyzed data from interviews with subject matter expert reverse engineers and found that reverse engineering goals revolved around understanding and elaborating the details of groups of instructions, functions, the program's interface to the operating system, and the program itself. Additionally, as people reverse engineer, their behaviors revolve around uncovering the objects, attributes, and relations [27] in a system (such as a program) to piece together a mental picture of these structures.

Modeling knowledge for a reverse engineering agent 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. Some means to organize representations of knowledge are categories, composite objects, components, networks, and conceptual constructs [28], [29], [30], [31]. Each of these representations has its own formalism for encoding knowledge, but serves as a way to represent mental objects, their relations to other objects, and the constraints and relationships that connect them. Our mathematical representations are aimed at building an initial ontology as a background knowledge base for an agent to model reverse engineering.

E. Domain Ontologies

Ontologies have been proposed for malware behavior classifications [32] and for describing the cyber security of computer systems [33]. Work has also been done to create vulnerability ontologies for cloud computing environments [34]. However, multiple literature searches found a decided lack of proposals for ontologies describing information gleaned from static and dynamic binary analysis and related reverse engineering tasks. Binary analysis information will need to be represented in a coherent manner in order to build effective agent systems.

There are a number of components that must be elicited and represented to develop agent models for reverse engineering comprehension. Some of these include modeling the jobs, tasks, sub-tasks, and unit-tasks. Others include modeling various semantic facts from background knowledge like behavior of the tools, assembly language operations and rules, how processes normally execute, how system calls work, how memory is organized, how assembly statements map to high-level program plans, and so on. Other areas of knowledge involve procedural rules about how the agent should perform various elements in the task like how to perform actions in the tool, how to manipulate and update its mental model, and how to work with its goals. Still other modeling is required to assemble meta-knowledge about the task that the agent is currently undertaking. These include beliefs about behaviors of the program, facts about the task scenario, the current task and goal, information the agent is currently attending to, hypotheses that are active in the current situation, and approaches that have been tried but abandoned.

In the next section, we focus on the process of program experimentation with information flow and present a data model to anchor our discussion.

III. Knowledge Representation Data Model

In order to model how reverse engineers interact with reverse engineering tools during program experimentation, we define a knowledge representation data model to store and categorize the information from the tools and the concepts from a reverse engineer's working memory. The data model must be sufficiently expressive to capture the semantics of information gathered from the environment. An ontology is a natural choice for this knowledge representation component. Indeed, in the works presented by Alnusair and Zhao [35] and Zhang et al. [36], ontology languages like OWL[1] and RDF[2] are used for modeling software comprehension. Alnusair and Zhao developed representations of conceptual source code knowledge and design patterns. Zhang et al. constructed ontology-based program representations for identifying security flaws.

[1]https://www.w3.org/TR/owl2-overview/

[2]https://www.w3.org/TR/rdf-primer/

Our goals are similar except for two distinctions: (1) our reverse engineering task has no available source code, only a binary executable; and (2) we gather and interpret data from the whole task environment to analyze system-wide information flow. We also seek to design the data model to be able to compose individual facts and abstract them into more complex representations. For this we turn to Spivak and Kent's Ontology Logs, or ologs.

A. Ologs

Developed by David Spivak and Robert Kent, the olog is a category-theoretical model for knowledge representation [37]. Although closely related to other knowledge representation languages such as SQL, OWL, and RDF, ologs have distinct advantages due to their grounding in categories. Unlike RDF, olog diagrams can commute—meaning that two paths in a diagram can be equivalent. Another advantage is an expressiveness and semantic clarity that cannot be achieved with graphs. Morphisms can also be defined that act as constraints between ologs to integrate them into a single information system (discussed in the conclusion of this section).

At a high level, an olog is a category that models a real-world situation by connecting objects with arrows and labeling them. The primary objects are *types* and *aspects*.

Types are the objects that represent abstract concepts. A type is depicted as a box with a singular indefinite noun phrase. Examples follow:

$$\boxed{\text{a register } x} \qquad \boxed{\text{a 32-bit memory address}} \tag{1}$$

Compound types are also permitted. For example, this compound type describes a binary expression:

$$\boxed{\begin{array}{l}\text{a triple } (a,b,\diamond) \text{ where } a \text{ is} \\ \text{the first operand, } b \text{ is the} \\ \text{second operand, and } \diamond \text{ is} \\ \text{the operator}\end{array}} \tag{2}$$

Aspects describe how objects can be measured, viewed, or regarded. An aspect of A is $A \to B$, where B is a set of possible result values for A. For example:

$$\boxed{\text{a register } x} \xrightarrow{\text{has value}} \boxed{\text{an integer}} \tag{3}$$

This olog is read as "a register x has value which is an integer." Formally, an aspect f can be defined as a function $f : X \to Y$, where X is the *domain of definition* and Y is the *set of result values*. To be valid, any value $x \in X$ must map to *at most one* value $y \in Y$.

Facts are olog diagrams that commute. That is, there are two paths in the diagram that are equivalent. This is a key notion that category theory provides that graph-based knowledge representation models do not. Below is an example that describes an assembly jmp instruction:

$$\begin{tikzcd}
\boxed{\begin{array}{l}\text{an instruction} \\ \text{``jmp } x\text{''}\end{array}} \arrow[r, "x"] \arrow[dr, "x"'] & \boxed{\begin{array}{l}\text{a 32-bit} \\ \text{memory} \\ \text{address}\end{array}} \arrow[d, "\text{is}"] \\
& \boxed{\begin{array}{l}\text{a hexadecimal} \\ \text{integer from} \\ \text{00000000 to} \\ \text{FFFFFFFF}\end{array}}
\end{tikzcd} \tag{4}$$

Note that the label used for two of the aspects in the above example were simply the variable name x. Spivak and Kent use this convention as an alias for the label "yields, via the value of x." The abbreviation reduces clutter in the diagram but maintains clarity.

B. *Ologs in Software Reverse Engineering*

With the basics of ologs defined, we now present the ologs for common information types in a reverse engineer's task environment. We identify the information types with direction from the work by Bryant et al [38].

1) Assembly Instructions: These are the assembly instructions of a disassembled executable program. The primary three categories are:

1) Data movement (such as mov, pop, $push$),
2) Arithmetic and logical operations (such as add, xor), and
3) Control flow (such as jmp, cmp, $call$, ret).

To keep the knowledge representation model concise while remaining platform agnostic, we choose an intermediate language to build the ologs. From the motivating work on the formalization of dynamic taint analysis by Schwartz et al., the intermediate language they present, SIMPIL, is powerful enough to express the semantics of languages from Java to assembly [39].

Since this intermediate language was designed for dynamic data-flow analysis, it is an appropriate choice for the basis of this information type. Using SIMPIL, the generic ologs we define are:

1) Assignment

$$\begin{tikzcd}
\boxed{\text{a statement ``}a := b\text{''}} \arrow[r, "b"] \arrow[d, "a"'] & \boxed{\text{an expression}} \arrow[d, "\text{yields}"] \\
\boxed{\text{a variable}} \arrow[r, "\text{has assignment}"'] & \boxed{\text{an integer}}
\end{tikzcd} \tag{5}$$

The assignment olog takes the result of the expression on the right side and assigns it to the variable on the left. The diagram is commutative.

2) Binary expression

$$\begin{tikzcd}
\boxed{\text{a binary expression ``}a \diamond b\text{'' as a triple } (a, b, \diamond)} \arrow[r, "\diamond"] \arrow[d, "\text{has value}"'] \arrow[dr, "a", "b"'] & \boxed{\text{a binary operator}} \arrow[d, "\text{operates on}"] \\
\boxed{\text{an integer}} & \boxed{\text{an expression}} \arrow[l, "\text{yields}"]
\end{tikzcd} \tag{6}$$

More complicated is the olog for binary expressions. The new relation here is the operator, $\diamond$, which has the aspect of "operates on." This denotes $\diamond$ operating on the two given expressions a and b.

3) Unary expression

$$\begin{tikzcd}
\boxed{\text{a unary expression ``}\diamond\, a\text{'' as a pair } (a, \diamond)} \arrow[r, "\diamond"] \arrow[d, "\text{has value}"'] \arrow[dr, "a"] & \boxed{\text{a unary operator}} \arrow[d, "\text{operates on}"] \\
\boxed{\text{an integer}} & \boxed{\text{an expression}} \arrow[l, "\text{yields}"]
\end{tikzcd} \tag{7}$$

Similar to the binary expression is the olog for the unary expression. The only difference being that there is only one given expression, a.

4) Goto statement

$$\begin{tikzcd}
\boxed{\text{a statement ``goto } y\text{''}} \arrow[r, "\text{jumps to}"] \arrow[d, "y"] & \boxed{\text{a memory address}} \arrow[d, "\text{is}"] \\
\boxed{\text{an expression}} \arrow[r, "\text{yields}"] & \boxed{\text{an integer}}
\end{tikzcd} \tag{8}$$

The olog for the goto statement has a similar structure to the assignment olog. This olog introduces how some variables have explicit use as memory addresses. The diagram also commutes.

5) Get_input expression

$$\begin{tikzcd}
\boxed{\text{an expression ``get\_input(}src\text{)''}} \arrow[r, "src"] \arrow[dr, "\text{returns}"'] & \boxed{\text{a source}} \arrow[d, "\text{returns}"] \\
 & \boxed{\text{an integer}}
\end{tikzcd} \tag{9}$$

The expression get_input(src) returns input from a source src. If the source is not relevant, then src is ignored and passed as blank, get_input($\cdot$). Since get_input($\cdot$) is the only source of user input, this statement is used to represent any return value from a system or library call [39]. The diagram commutes.

6) Load expression

$$\begin{tikzcd}
 & \boxed{\text{an expression ``}load(y)\text{''}} \arrow[r, "y"] \arrow[dr, "\text{loads from}"'] & \boxed{\text{an expression}} \arrow[d, "\text{yields}"] \\
\boxed{\text{an integer}} & & \boxed{\text{a memory address}} \arrow[ll, "\text{has value}"]
\end{tikzcd} \tag{10}$$

The olog for the load expression describes how the expression y yields an address in memory and the value at that address is returned.

7) Store statement

$$\begin{tikzcd}
 & \boxed{\text{a statement ``}store(x, y)\text{''}} \arrow[dl, "x"'] \arrow[dr, "y"] & \\
\boxed{\text{an expression}} \arrow[d, "\text{yields}"] & & \boxed{\text{an expression}} \arrow[d, "\text{yields}"] \\
\boxed{\text{an integer}} \arrow[rr, shift left, "\text{is stored at}"] & & \boxed{\text{a memory address}} \arrow[ll, shift left, "\text{is}"]
\end{tikzcd} \tag{11}$$

The olog for the store statement takes the value of x and stores it at the memory location that is y.

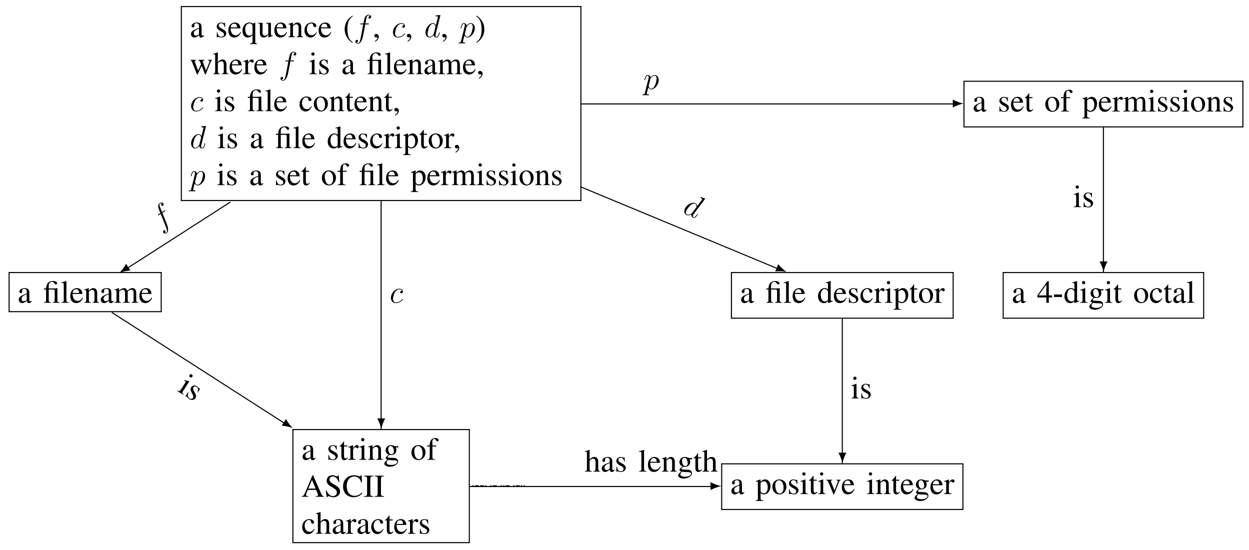
a sequence (f, c, d, p) where f is a filename, c is file content, d is a file descriptor, p is a set of file permissions
p
a set of permissions
is
f
d
a filename
c
a file descriptor
a 4-digit octal
is
is
a string of ASCII characters
has length
a positive integer

Fig. 2. An olog to describe an observed file.

2) Program Data: This information type represents the state of the processor, its register values, and flags as well as data stored in memory locations, files, or system objects. Although a string of bytes may have a number of different interpretations, the olog of this information type is basic. The object in question (register, flag, memory location, etc.) maps to a value (integer, ASCII text, floating point, etc.).

3) Observable Features: Information in this category consists of windows being opened, files being created, or system objects being written to [38]. These are the changes that the reverse engineer directly observes when executing the program. An olog for a file observed by the reverse engineer is shown in Figure 2.

4) Control Flow: This information type represents branching and splitting the assembly instructions into functional components. An example for a basic olog that describes a function is:

$$\begin{tikzcd}
\boxed{\text{a function with parameters } (p_1, \ldots, p_n)} \arrow[r, "\text{has}"] \arrow[d, shift right=6, "p_1"'] \arrow[d, "p_2"] \arrow[d, shift left=6, "\cdots\ p_n"] & \boxed{\text{a return address}} \arrow[d, "\text{is}"] \\
\boxed{\text{a parameter}} \arrow[r, "\text{is}"] & \boxed{\text{an integer}}
\end{tikzcd} \tag{12}$$

These ologs define the different information flow types in a reverse engineer's task environment. They can also be chained together to form more complex facts; an example is shown in Figure 3.

In general, to "chain together" multiple ologs Spivak and Kent describe how to construct systems of ologs [37]. A system represents the connections and commonalities shared by a set of ologs. For instance, let $\mathbf{X} = \{S_1, S_2, S_3\}$ be a finite set of ologs. Then S_{123} is the "common ground world-view" between all three ologs—which can be empty. For each subset $\mathbf{Y} \subseteq \mathbf{X}$ there is a map from $S_{123} \to S_{\mathbf{Y}}$ which represents any common-ground between any two ologs. The system is illustrated in Figure 4. Through this mechanism we can compose ologs together to represent more complex concepts.

As an agent reverse engineers an executable program, their actions and the data they gather can be represented in these ologs. With the knowledge representation data model defined, we now propose the information flow model.

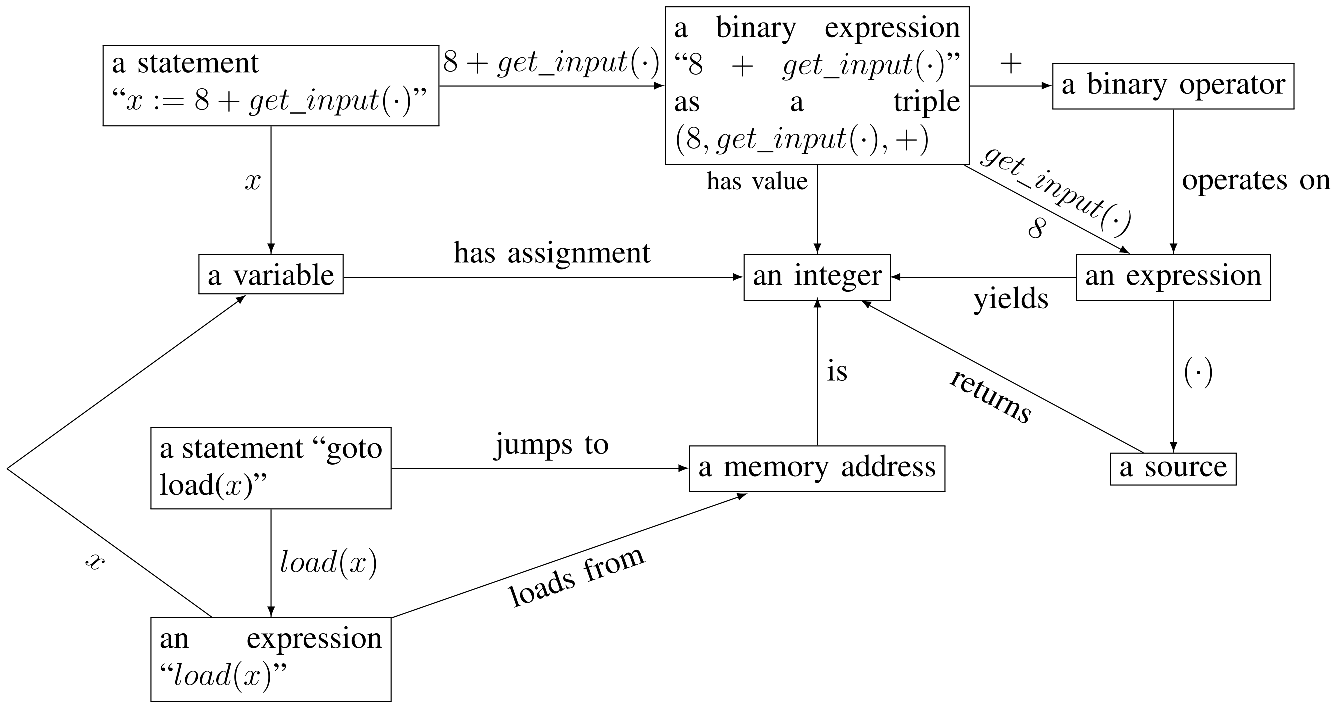

Fig. 3. A chain of several ologs. Chaining ologs is not limited to a single instruction. This diagram is made up of two instructions: (1) $x := 8 + get\_input(\cdot)$ and (2) $goto\ load(x)$

$$\begin{tikzcd}
 & S_1 & \\
S_{12} \arrow[ur] \arrow[d] & S_{123} \arrow[l] \arrow[r] \arrow[d] & S_{13} \arrow[ul] \arrow[d] \\
S_2 & S_{23} \arrow[l] \arrow[r] & S_3
\end{tikzcd}$$

Fig. 4. A generic system of ologs. Shared common-ground between all ologs in the set is represented by S_{123} and its mappings. Pair-wise commonalities are represented similarly by S_{12}, S_{13}, and S_{23}.

IV. Modeling Information Flow

The reverse engineer’s task environment is a complex distributed system. Discrete parts of the environment produce an abundance of data and can be manipulated by the agent in a number of ways. To capture information flow in this distributed system we introduce the notion of “flow.” Information flow is observed when “a part X of a distributed system being of type α carries the information that another part Y is of type β” [40]. We present an example to illustrate this notion.

A. A Motivating Example

Consider an agent’s task environment while reverse engineering an executable program. Let it be split into several parts: assembly instructions, program data, control flow data, and the observable features. We illustrate it as follows:

$$\begin{tikzcd}
\text{Assembly Instructions} \arrow[dr, no head] & & \text{Program Data} \arrow[dl, no head] \\
& \text{Binary Executable} \arrow[dl, no head] \arrow[dr, no head] & \\
\text{Control Flow} & & \text{Observable Features}
\end{tikzcd} \tag{13}$$

Suppose that the section of code an agent is debugging appears as follows:

```
            1:  add  eax,0x8
            2:  jmp  eax
                  ...

jmp_target:
            syscall(create_file)
```

The idea is that this part of the program adds 8 to a register (line 1) and jumps to the location in memory corresponding to that value (line 2). Then, at the jump target, a system call is executed that creates a file in the current directory (which the agent observes).

Using the notion of information flow described in the section above, let's analyze the example scenario. For the first line, an instruction of type ADDTOREGISTER carries the information that there is a program-data object of type REGISTERMODIFY. Next, an instruction of type JUMP carries the information that there is a control flow object of type JUMPTARGET. Then, an instruction of type SYSCALL being executed carries the information that there is an observable feature of type FILECREATED. The types are elementary descriptions, but the notion still stands. This leads us to another theoretical framework motivated by category theory that rigorously defines the notion of "flow."

B. Channel Theory

We choose Channel Theory to formalize information flow in software reverse engineering. Developed by Jon Barwise and Jerry Seligman [40], Channel Theory uses categories to model information flow in a way that captures how agents reason about their environment using partial information. This is a key requirement in the information flow model, as the reverse engineer does not have a complete understanding of the executable program, but can still reason about it. We now define the key objects of Channel Theory.

1) Classifications: A *classification* is a structure $\mathbf{A} = \langle A, \Sigma_A, \vDash_A \rangle$ made up of a set of *tokens* A (the set of objects to be classified); a set of *types* Σ_A (the set of objects used to classify the tokens); and a binary relation $\vDash_A$ between A and Σ_A that determines which tokens are classified by which types [40]. The statement $a \vDash \alpha$ can be read as "an object a is of the type α."

To support the flow of information, we need structures that handle the logic of a system. For a classification $\mathbf{A}$, a *sequent* is a pair (Γ, Δ) of sets of types of $\mathbf{A}$. A token a *satisfies* the sequent (Γ, Δ) if $(\forall\, \alpha \in \Gamma)\ (a \vDash_A \alpha) \Rightarrow (\exists\, \delta \in \Delta)$ such that $(a \vDash_A \delta)$.

Γ *entails* Δ in $\mathbf{A}$, denoted $\Gamma \vdash_A \Delta$, if every token of $\mathbf{A}$ satisfies (Γ, Δ). If $\Gamma \vdash_A \Delta$, then (Γ, Δ) is a *constraint* supported by the classification $\mathbf{A}$. Constraints relate the types of a classification. The set of all constraints supported by a classification **A** is the complete theory of **A**. The regularities of a system are represented by the constraints. Information flows along these regularities at the system level. These regularities require mappings from parts of the system to the system as a whole. These mappings are called *infomorphisms*.

2) Infomorphisms: Let $\mathbf{A} = \langle A, \Sigma_A, \vDash_A \rangle$ and $\mathbf{C} = \langle C, \Sigma_C, \vDash_C \rangle$ be two classifications. An *infomorphism* between **A** and **C** is a pair $f = \langle f^\wedge, f^\vee \rangle$ of functions such that for all tokens c of **C** and all types α of **A** the following is true:

$$f^\vee(c) \vDash_A \alpha \text{ iff } c \vDash_C f^\wedge(\alpha).$$

$f^\wedge$ and $f^\vee$ act in opposite directions, i.e. $f : \mathbf{A} \rightleftarrows \mathbf{C}$. $f^\wedge$ maps types from the left classification to types of the right. $f^\vee$ maps tokens the other way. A valid infomorphism makes the following diagram commute:

$$\begin{tikzcd}
\Sigma_A \arrow[r, "f^\wedge"] \arrow[d, dash, "\vDash_A"] & \Sigma_C \arrow[d, dash, "\vDash_C"] \\
A & C \arrow[l, "f^\vee"]
\end{tikzcd} \tag{14}$$

An *information channel* is a family of infomorphisms with a common codomain called the *core*, **C** [40]. Formally, $\mathcal{C} = \{f_i : \mathbf{A}_i \rightleftarrows \mathbf{C}\}_{i \in I}$. Classifications $\mathbf{A}_i$ represent the parts of a distributed system. The core represents the system as a whole. A token in **C** connects to related tokens in other parts of the system by means of f_i.

C. How Information Flows

We now consider the example scenario (Diagram 13) from Section 4.1 in terms of an information channel.

$$\begin{tikzcd}
\mathbf{A}_1 \arrow[dr, dash, "f_1"] & & \mathbf{A}_2 \arrow[dl, dash, "f_2"'] \\
& \mathbf{C} & \\
\mathbf{A}_4 \arrow[ur, dash, "f_4"] & & \mathbf{A}_3 \arrow[ul, dash, "f_3"']
\end{tikzcd} \tag{15}$$

Let the classifications correspond to the parts of the task environment: $\mathbf{A}_1$ to assembly instructions; $\mathbf{A}_2$ to program data; $\mathbf{A}_3$ to observable features; and $\mathbf{A}_4$ to control flow. Applying the machinery presented in the previous section, the $\mathbf{A}_1$-type (the assembly instruction type) SYSCALL has a translation $f_1^\wedge(\text{SYSCALL})$ in **C**. Similarly, the $\mathbf{A}_3$-type FILECREATED has a translation $f_3^\wedge(\text{FILECREATED})$ in **C**. Thus, the instruction being a system call carries the information that a file is created by the following inference:

$$f_1^\wedge(\text{SYSCALL}) \vdash_{\mathbf{C}} f_3^\wedge(\text{FILECREATED})$$

Given the context of a particular token of **C**—in this case, a particular executable program—the translations of types in **C** provide the regularities that support information flow.

D. Composing and Integrating the Models

In Section 3, we described how to compose ologs into more complex representations. Spivak and Kent relate these systems of ologs to Barwise and Seligman's information channels in the following way [37, Section 4]. An olog represents the world-view of a single part of a distributed system (a classification in Channel Theory). The functors between ologs are the channels which allow communication to occur. Such a distributed system is (in the Channel-theoretic sense) an information channel with a core and translations (olog morphisms) that connect the parts to the core.

To unify the olog data model and the information flow model, we describe a way to integrate the two. The ologs drive the semantic information being extracted from the task environment. The information flow model provides the logic that supports flow. A straightforward method to combine the two is to use the ologs as the types for the classifications in the information flow model. Let's consider the classification for assembly instructions ($\mathbf{A}_1$ in the previous section). The tokens for this classification are instructions represented by tuples. For example, a token $a = \langle add, eax, 8\rangle$ and a token $b = \langle jmp, eax\rangle$. Recall that the types for a classification are the objects that classify the tokens. Then the type α such that $a \vDash \alpha$ corresponds to the olog for the BINARYEXPRESSION statement defined in Section 3.2, Diagram 6. Similarly, the type β such that $b \vDash \beta$ is the olog for the GOTO statement defined in Section 3.2, Diagram 8. This integration of the two models allows for concurrent design. As new ologs are defined for the data model, the classifications in the information flow model are updated at the same time. Since the ologs are used directly in the information flow model, no translation needs to be performed and thus no semantic information is lost.

V. CONCLUSIONS AND FUTURE WORK

This paper proposed a model of information flow in a reverse engineer's task environment as part of a reverse engineer's program experimentation process. The model is a first step in framing Bryant's theory of sensemaking in mathematical formalisms. Formalizing this aspect of sensemaking in reverse engineering provides a framework to model how autonomous, cognitive agents perform complex tasks. This research is relevant to cyber defense and artificial intelligence researchers and those attempting to model complex comprehension tasks in other disciplines. The general methodology proposed in this paper is not limited to software reverse engineering but can apply to other cyber defense domains that involve sensemaking like network traffic analysis and cyber threat assessment. More work needs to be done to fully specify the information flow model, demonstrate it, and determine its validity. Ultimately, the goal is to uncover specific areas for improvement in automation, support, and training to cyber defense professionals to help them handle cyber security threats faster, more accurately, and at a greater scale.

Creating agents to model reverse engineering tasks requires several technologies, many that have yet to be defined or implemented. Some well-defined reverse engineering tasks can be modeled in a straightforward, algorithmic fashion such as binary disassembly, feature extraction, function comparisons, and metadata extraction. Other reverse engineering tasks are not as straightforward or well-defined such as discovering intent, finding patterns, discovering bugs, and decomposing architectures. To effectively model tasks in either category will require mapping both the binary software characteristics and the analyst's mental models to formal computer-readable specifications.

Our research created an initial domain ontology for representing knowledge of reverse engineers. Future research will focus on cognitive models to provide a computational framework for modeling and simulating cognitive tasks in malware and vulnerability analysis. Future activities for this research are to improve our abstracted concept model and to model a unit task end-to-end. The concept model can be represented in a machine-readable first-order predicate language such as RDF (resource description framework) as a node-graph of concepts and relationships and the actions of the analyst to extract program intent can be described in terms of this concept model in a cognitive architecture such as ACT-R [41] or MIDCA [42]. Such a cognitive model can then be simulated and, when connected to currently available software reverse engineering tools such as the radare2 disassembly framework [43], act as a prototype automated agent performing analysis tasks.

Additionally, more ologs need to be defined to complete the knowledge representation data model. This paper focused on instruction-based ologs, but the program data, observable features, and control flow categories need more thorough treatment. A demonstration of the olog data model should include a computable implementation. Conveniently, ologs can be translated to RDF or to a relational schema while preserving many of the ologs useful properties [37]. Once implemented, the olog data model can be incorporated into a cognitive architecture as described above. In this respect, it provides the background knowledge for an autonomous agent reverse engineering software.

Further, the information flow model needs to be fully specified in Channel Theory. Each classification in the model should have defined tokens, types, and constraints. Infomorphisms from each classification to the core need to be constructed. The infomorphisms provide the translations between parts of the system and give the ability to apply a local logic onto a classification to reason about it [40]. To demonstrate flow, the information flow model should simulate several end-to-end reverse engineering tasks.

Given the information flow model's grounding in first-order logic, it shares the same limitations of other first-order theories. An individual classification can be shown to be sound or complete by analyzing the satisfiability of its sequents. However, as a whole, the information flow model—essentially, a model of a program to be reverse engineered—is a theory. As such, it is limited by necessarily not being complete. Under the construction presented here, given any input, the results of a program can be observed and a model can be constructed. However, the converse does not hold, demonstrating the model's incompleteness.

The information flow model is also limited in handling advanced binary analysis scenarios. There are no mechanisms to correctly trace information flow in obfuscated or encrypted binaries. It is our goal that by designing the models to be composable and addressing the simpler case of analyzing unprotected assembly code, we can simulate more advanced scenarios by composing the parts into more complex representations.

References

[1] A. Bryant, "Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations," Ph.D. dissertation, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 2012.

[2] 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," in *In Proceedings of the 4th International Conference on Information Systems Security (ICISS).* Springer, 2008, Conference proceedings, pp. 1–25.

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

[4] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel.* Addison-Wesley Professional, 2006.

[5] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ACM, 2007, Conference proceedings (article), pp. 5–14.

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

[7] D. Quist and L. Liebrock, "Visualizing compiled executables for malware analysis," in *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on.* IEEE, 2009, pp. 27–32.

[8] R. Canzanese, Jr., M. Oyer, S. Mancoridis, and M. Kam, "A survey of reverse engineering tools for the 32-bit microsoft windows environment," 2005. [Online]. Available: https://www.cs.drexel.edu/~spiros/teaching/CS675/

[9] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler.* No Starch Press, 2011.

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

[11] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code.* Addison-Wesley, 2005.

[12] U. Drepper, "What every programmer should know about memory (2007)," 2010. [Online]. Available: http://people.redhat.com/drepper/cpumemory.pdf

[13] S. Hanov, "Static analysis of binary executables," *Citeseer*, pp. 97–102, 2009. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.6265&rep=rep1&type=pdf

[14] A. Kapoor, "An approach towards disassembly of malicious binary executables," The Center for Advanced Computer Studies, University of Louisiana at Lafayette, Master's Thesis, 2004.

[15] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, 1984.

[16] T. Biggerstaff, B. Mitbander, and D. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[17] S. Tilley, "A Reverse-Engineering Environment Framework," Carnegie-Mellon Software Engineering Institute, Technical report, 1998.

[18] A. von Mayrhauser and A. M. Vans, “Comprehension processes during large scale maintenance,” in *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society Press, 1994, pp. 39–48.

[19] V. Fix and S. Wiedenbeck, “Mental representations of programs by novices and experts,” *Proceedings of INTERCHI*, pp. 74–79, 1993.

[20] G. Gannod and B. Cheng, “A formal approach for reverse engineering: a case study,” *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pp. 100–111, 1999.

[21] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, “On the comprehension of program comprehension,” *ACM Transactions on Software Engineering Methodology*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014.

[22] R. Brooks, “Towards a theory of the comprehension of computer programs,” *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983.

[23] I. Vessey, “Expertise in debugging computer programs: A process analysis,” *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[24] J. Josephson, *Abductive Inference: Computation, Philosophy, Technology*. Cambridge Univ Pr, 1996.

[25] M. A. Vans, “Program understanding behavior during corrective maintenance of large-scale software,” *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 31–70, Jul. 1999.

[26] A. Bryant, R. Mills, G. Peterson, and M. Grimaila, “Top-level goals in reverse engineering,” *Journal of Information Warfare*, vol. 12, no. 1, 2013.

[27] H. Wang, T. R. Johnson, and J. Zhang, “The order effect in human abductive reasoning: an empirical and computational study,” *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 18, no. 2, pp. 215–247, 2006.

[28] L. Barsalou, W. Yeh, B. Luka, K. Olseth, K. Mix, and L. Wu, “Concepts and meaning,” in *Chicago Linguistics Society 29: Papers from the Parasession on Conceptual Representations*, K. Beals, G. Cooke, D. Kathman, K. McCullough, S. Kita, and D. Testen, Eds. Chicago Linguistics Society, 1993.

[29] B. Chandrasekaran, A. Goel, and Y. Iwasaki, “Functional representation as design rationale,” *IEEE Computer*, vol. 26, no. 1, pp. 48–56, 1993.

[30] K. Morik, B. Kietz, W. Emde, and S. Wrobel, *Knowledge Acquisition and Machine Learning*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA, 1993.

[31] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2011.

[32] A. Grecio, R. Bonacin, O. Nabuco, V. Monte Afonso, P. Licio De Geus, and M. Jino, “Ontology for malware behavior: A core model proposal,” in *2014 IEEE 23rd International WETICE Conference (WETICE)*, June 2014, pp. 453–458.

[33] P. D. Zegzhda, D. P. Zegzhda, and T. V. Stepanova, “Approach to the construction of the generalized functional-semantic cyber security model,” *Automatic Control and Computer Sciences*, vol. 49, no. 8, pp. 627–633, 2016.

[34] P. Kamongi, S. Kotikela, K. Kavi, M. Gomathisankaran, and A. Singhal, “Vulcan: Vulnerability assessment framework for cloud computing,” in *Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on*, June 2013, pp. 218–226.

[35] A. Alnusair and T. Zhao, “Towards a model-driven approach for reverse engineering design patterns,” in *In Proceedings of the 2nd International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering*, 2009.

[36] Y. Zhang, J. Rilling, and V. Haarslev, “An ontology based approach to software comprehension - reasoning about security concerns in source code,” in *Proceedings of the 30th Annual International Computer Software and Applications Conference*, ser. COMPSAC 2006. IEEE Computer Society Press, 2006, pp. 333–342.

[37] D. I. Spivak and R. E. Kent, “Ologs: A categorical framework for knowledge representation,” *PLoS ONE*, vol. 7, no. 1, p. e24274, 01 2012.

[38] A. Bryant, R. Mills, G. Peterson, and M. Grimaila, “Software reverse engineering as a sensemaking task,” *Journal of Information Assurance and Security*, vol. 6, p. 6, 2011.

[39] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask),” in *2010 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2010, pp. 317–331.

[40] J. Barwise and J. Seligman, *Information Flow: The Logic of Distributed Systems*. Cambridge University Press, 1997.

[41] J. Anderson, *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press, USA, 2007, vol. 3.

[42] M. T. Cox, T. Oates, and D. Perlis, “The integration of cognitive and metacognitive processes with data-driven and knowledge-rich structures,” in *Proceedings of the Annual Meeting of the International Association for Computing and Philosophy*, 2013.

[43] pancake, “Radare2,” 2016. [Online]. Available: http://radare.org/r/