Sisco, Z.D. & Bryant, A.R. (2017). A Semantics-Based Approach to Concept Assignment in Assembly Code. In: *Proceedings of the 12th International Conference on Cyber Warfare and Security, Dayton, Ohio, USA, 2-3 March 2017.*

A Semantics-Based Approach to Concept Assignment in Assembly Code

Zachary D. Sisco, Adam R. Bryant
Wright State University, Dayton, USA
sisco.8@wright.edu
adam.bryant@wright.edu

Abstract: Reverse engineering is a cyber-security task used to investigate functionality or identify vulnerabilities of compiled software. Reverse engineers analyze unprotected assembly code to these ends—which is difficult since assembly code is stripped of semantic information. In this paper, we present a formal method for mapping concepts to locations in assembly code and extracting information about their use. To do this, we model concept assignment using the operational semantics of a formal language. To guide concept assignment, we define a knowledge representation data model to integrate with the dynamic analysis process. The data model organizes concepts to reflect a reverse engineer's mental model when performing reverse engineering tasks. We illustrate our method by recognizing dynamically allocated data structures in assembly programs. By formalizing concept assignment in assembly code, tools and models can be developed that assist reverse engineers—thus improving their ability to investigate malware or discover vulnerabilities.

Keywords: Reverse engineering, program comprehension, assembly language, concept assignment, formal languages, binary analysis

1. Introduction

Reverse engineering compiled software requires analyzing and comprehending unprotected assembly code to understand the behaviors and intentions of the software. Understanding programs from assembly language representations is not an easy task. Assembly instructions do not contain any of the semantic information that the original source code may have had—like variable and function names, comments, and documentation. This makes it difficult to map higher-level concepts to locations in assembly code. This mapping problem is called the *concept assignment problem* (Biggerstaff et al., 1994). Nevertheless, solving the concept assignment problem for assembly code will enable more powerful automated approaches to formally describing classes of vulnerabilities and finding instances of vulnerabilities in fielded software. Our research is the first attempt at formalizing concept assignment in assembly programs. In this paper we contribute the following:

- We extend the operational semantics of a formal language to identify, trace, and elicit properties of dynamically allocated data structures from assembly code (Section 3).
- We define a knowledge representation data model to systemically organize concepts and semantic information that drive analyses executed in the formal language (Section 4).

This research fits into the larger scope of modeling comprehension in software reverse engineering. By formally modeling the cognitive processes of reverse engineers and how they "make sense" of assembly programs, we come closer to developing autonomous agents that can comprehend compiled software. This is one of the goals of the sensemaking theory proposed by Bryant (2012). Bryant's sensemaking theory describes how reverse engineers make sense of assembly programs through interactive mental model construction and goal-directed information seeking from reverse engineering tools (Bryant, 2012). Concept assignment is one such interactive process that reverse engineers utilize when comprehending assembly programs.

By modeling concept assignment in assembly code, cyber defense analysts can utilize automated or human-in-the-loop tools that formally verify the existence of concepts and programming constructs to aid the analyst in understanding malware or vulnerable software. In a cyber-warfare scenario, a cyber analyst can utilize automated concept assignment to help them comprehend exfiltrated compiled software. Since the concept assignment solution proposed in this paper is based on formal methods, an automated tool can formally prove if a program contains a certain feature or construct.

2. Background

Modeling concept assignment for assembly code requires understanding how reverse engineers make sense of assembly-language programs. Reverse engineers use tools to observe and interact with parts of a program. These tools provide the reverse engineer with information representations such as: assembly instructions, data stored in program sections, control-flow data, and system data (Bryant et al., 2011). Reverse engineering involves a range of processes that use these information representations. An essential process, *program analysis*, is the reading and comprehension of sequences of instructions where reverse engineers construct their mental representation of a program (von Mayrhauser & Vans, 1994). At a higher level, reverse engineers also recognize programming "plans" (or recurring program patterns) as a way to infer the intent of a program and ground previously known concepts into the program's code (Soloway & Ehrlich, 1984; Fix et al., 1993). For assembly-language programs, these interactions are necessary to accomplish goals related to understanding groups of instructions, the behaviors of the program, and the interactions between the program's components and the system (Bryant et al., 2013). In total, reverse engineers use these processes to construct and refine a complete "picture," or mental model, of the program (Bryant et al., 2013).

The *concept assignment problem* is the problem of recognizing concepts and assigning them to locations within a program in order to build an understanding of that program (Biggerstaff et al., 1994; Rajlich, 2009). Concepts

include programming constructs (Biggerstaff et al., 1994) (e.g. instantiating a class, iterating through a linked list, or allocating dynamic memory), or features of the program itself (Rajlich, 2009)—i.e. the part of a program that exhibits a certain behavior. Feature location techniques have been researched to automate this process in source code. These employ combinations of static, dynamic, and textual analysis techniques such as comparing execution traces for the existence of a feature (Wilde & Scully, 1995), searching the dependency graph of the program (Chen & Rajlich, 2000), and pattern matching text in source code comments and identifiers (Petrenko et al., 2008). Although related in goals, these feature location techniques focus on source code instead of assembly code.

In this paper, we restrict the concept assignment problem to assembly language representations of programs. Restricting the problem to assembly language presents challenges that do not exist with concept assignment in source code. For instance, the lack of data types in assembly makes it difficult to determine if an instruction is relevant to the analysis. Additionally, assembly does not have higher-order abstractions like functions—there are only conditional and unconditional jumps. Feature location techniques that rely on tracing function calls and pattern matching source code comments and identifiers fail with assembly code because those artifacts do not exist.

3. Method

We approach the assembly-language concept assignment problem by first lifting assembly code into a simpler, but formally specified intermediate language. A formal language allows us to define special operational semantics and policies to identify, trace, and elicit properties of concepts in assembly code. We choose the intermediate language developed by Schwartz et al. (2010) called SIMPIL: a Simple Intermediate Language. Its only data type is a 32-bit integer and all of its expressions are side-effect free. Schwartz et al. (2010) argue that despite its simple grammar, SIMPIL, "is powerful enough to express typical languages as varied as Java and assembly code," and, "is representative of internal representations used by compilers for a variety of programming languages." SIMPIL was first developed to formalize the algorithms for dynamic taint analysis and forward symbolic execution by describing them via run-time semantics. Their formalization of these algorithms motivated our approach to the concept assignment problem. We present our modified grammar of SIMPIL in Section 3.1.

The primary example in this section is a program that creates a heap-allocated array—a common data structure that can grow or shrink how much memory it uses. The code in Listing 1 is a C program that uses *malloc* to dynamically allocate space for three variables on the program heap. This example is adapted from Eagle (2011).

```
int main() {
    int *heap_array = (int*)malloc(3 * sizeof(int));
    int index = 2;
    heap_array[0] = 10;
    heap_array[1] = 20;
    heap_array[index] = 30;
}
```

Listing 1: C program for creating a heap-allocated array.

Once compiled, we use GNU Binutils (GNU, 2014) to disassemble the program into the following assembly language representation (shown in Listing 2).

```
0000000000400506 <main>:
  400506:       push   rbp
  400507:       mov    rbp,rsp
  40050a:       sub    rsp,0x10
  40050e:       mov    edi,0xc
  400513:       call   400400 <malloc@plt>
  400518:       mov    QWORD PTR [rbp-0x8],rax
  40051c:       mov    DWORD PTR [rbp-0xc],0x2
  400523:       mov    eax,QWORD PTR [rbp-0x8]
  400527:       mov    DWORD PTR [rax],0xa
  40052d:       mov    rax,QWORD PTR [rbp-0x8]
  400531:       add    rax,0x4
  400535:       mov    DWORD PTR [rax],0x14
  40053b:       mov    eax,DWORD PTR [rbp-0xc]
  40053e:       cdqe
  400540:       lea    rdx,[rax*4+0x0]
  400548:       mov    rax,QWORD PTR [rbp-0x8]
  40054c:       add    rax,rdx
  40054f:       mov    DWORD PTR [rax],0x1e
  400555:       leave
  400556:       ret
```

Listing 2: Assembly code for creating a heap-allocated array.

Without modifying SIMPIL, we lift this assembly code to the following intermediate representation shown in Listing 3. For convenience, we convert the hexadecimal numbers to decimal and change any 64-bit references to conform to SIMPIL's 32-bit integer data type.

Since SIMPIL was designed for dynamic taint analysis, it treats all system and library calls as "input" to the program. Therefore, on line 6 of Listing 3, the assembly instruction "*call 400400 <malloc@plt>"* is lifted to the expression get_input(*malloc*).

```
1  esp := esp − 1
2  store(esp, ebp)
3  ebp := esp
4  esp := esp − 2
5  edi := 3
6  eax := get_input(malloc)
7  store(ebp − 1, eax)
8  store(ebp − 2, 2)
9  eax := load(ebp − 1)
10 store(eax, 10)
11 eax := load(ebp − 1)
12 eax := eax + 1
13 store(eax, 20)
14 eax := load(ebp − 2)
15 edx := eax * 1 + 0
16 eax := load(ebp − 1)
17 eax := eax + edx
18 store(eax, 30)
19 esp := ebp
20 ebp := load(esp)
21 halt
```

Listing 3: Disassembly in Listing 2 lifted to SIMPIL.

3.1 Extending SIMPIL

Now we extend SIMPIL to perform concept assignment and recognize that the instructions in Listing 3 initialize a heap-allocated array. After this, there will be several pieces of information we can extract from our analysis:

- How much memory was allocated to the program heap;
- The pointer in memory to the data structure;
- The total number of elements possible to allocate;
- The data stored in the array;
- If any access to the heap array is out-of-bounds (possible buffer overflow).

The rules and policies that follow arise from the fact that *malloc* is passed as the argument to get_input(·). Knowledge of system and library calls is essential for concept assignment to work in this formalism. Disassemblers and debuggers such as Hex-Ray's IDA Pro (Hex-Rays, 2015), the GNU Project Debugger (GNU, 2016), and GNU Binutils (GNU, 2014) already identify system calls. So it is assumed that this information is available for our analysis. To use it in the intermediate language we pass the name of the system call as the source for get_input(·). This requires a change in the grammar of SIMPIL which is shown in the last row of Table 1.

$program$	::=	$stmt$*
$stmt\ s$	::=	var := exp | store(exp, exp) | goto exp | assert exp | if exp then goto exp else goto exp | halt
$exp\ e$	::=	load(exp) | $exp \diamond_b exp$ | $\diamond_u exp$ | var | get_input(src) | v
$\diamond_b$	::=	$+$ | $-$ | $*$ | $/$ | $\vee$ | $\wedge$ | $<$ | $\leq$ | $>$ | $\geq$ | $=$
$\diamond_u$	::=	$-$ (unary minus) | $\neg$ (logical negation)
$value\ v$	::=	32-bit unsigned integer | $\perp$
src	::=	string

Table 1: The grammar of SIMPIL modified to accept string arguments as sources for get_input(·).

In Table 2, we introduce additional notation to SIMPIL to track the value returned from a *malloc* call, the amount

of memory allocated to the heap, and the variables that reference the pointer to the heap-allocated array. For convenience, if a rule or policy maps an element from Table 2 to a key value that does not exist the mapping will return the null value, ⊥. If a null value is present in any binary or unary boolean expression, the result will be **F**, false.

To demonstrate the use of the values in Table 2 we construct the operational semantics for the relevant statements and expressions—get_input(*src*), load(*exp*), and store(*exp, exp*). The goal here is to track the pointer that is returned from the statement get_input(*malloc*) in line 6 of Listing 3.

Expression		Interpretation
is-heap-pointer h	$::=$	**T** | **F**
heap-size r	$::=$	Integer
H_μ	$::=$	Maps addresses to heap pointer status h
H_ρ	$::=$	Maps addresses to size r
φ	$::=$	Variable that holds value of pointer

Table 2: New notation for tracking heap-allocated memory in SIMPIL.

Operational semantics model a programming language's execution by explicitly describing the state changes of the program context. Operational semantics in SIMPIL are of the form:

$$\frac{\text{computation}}{\langle \text{current state} \rangle, \text{stmt} \rightsquigarrow \langle \text{end state} \rangle, \text{stmt'}}$$

In this form, each rule is read bottom to top, and left to right with the name of the rule appended to the right in capital letters (Schwartz et al., 2010). As each rule is applied, it changes the state of the program context, which is defined in terms of a set of variables described below.

The modified rules are presented in Figure 1. For space, the unmodified operational semantics—such as ASSIGN and BINARYOPERATOR—are not included here, but they can be referenced in the original work by Schwartz et al. (2010). There are five meta-syntactic variables that the rules use in the execution context: (Σ) a list of program statements that maps a statement number to a statement; (μ) the current memory state which maps a memory address to the current value at that address; (Δ) maps variable names to their current values; (*pc*) the program counter; and (ι) the current statement. The notation $\mu, \Delta \vdash e \Downarrow v$ denotes starting from the context given by the memory address and value defined by μ and Δ and evaluating an expression *e* to a value *v* in that context.

The definition for the function $P_{\text{heapcheck}}$, which verifies that loads and stores are within the bounds of a heap, can be found in Table 3. It can equivalently be expressed as: (*dest* < *base* + *size*) ∨ ¬*isheap*. Table 3 lists all of the policies used in the operational semantics in Figure 1. Policies that return truth values are indicated with *P*, and piecewise-valued policies that update variables are indicated with Greek letters corresponding to the variable name.

Policy	Computation
$P_{\text{malloc}}(src)$	$src =$ 'malloc'
$P_{\text{heapcheck}}(dest, base, size, isheap)$	$isheap \longrightarrow (dest < base + size)$
$\rho_{\text{malloc}}(src, val)$	$\begin{cases} val & P_{\text{malloc}}(src) \\ \bot & \text{otherwise} \end{cases}$
$\Phi(ptr, cond)$	$\begin{cases} ptr & cond \\ \bot & \text{otherwise} \end{cases}$

Table 3: New policies for tracking heap-allocated memory in SIMPIL.

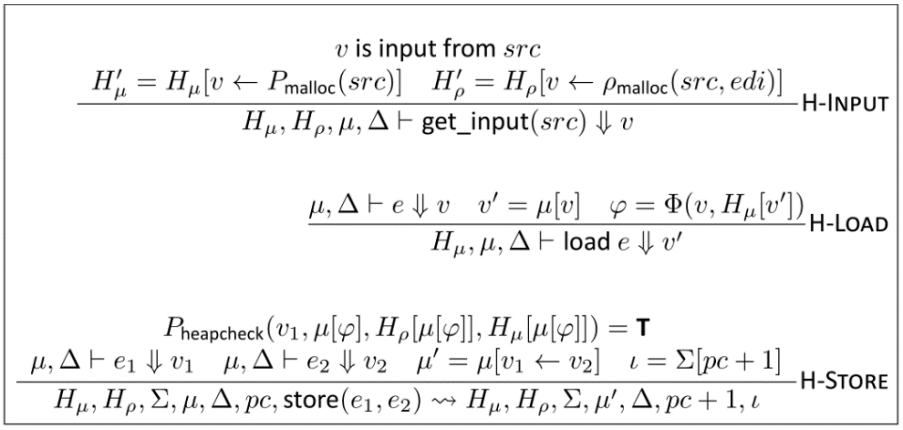


Figure 1: Modified operational semantics for tracking heap-allocated memory.

Tracing heap allocation starts at get_input(*src*) on line 6 of Listing 3. Since the *src* argument is equal to 'malloc', the H-INPUT rule updates the heap status H_μ by mapping the base address in heap memory—the value returned from get_input(*malloc*)—to **T**, true. Also, the rule updates the heap size H_ρ by mapping the base address in heap memory to the amount of memory allocated. The *edi* variable provides the size of the memory block.

One pattern that occurs in the assembly code in Listing 2 and the lifted SIMPIL program in Listing 3 is that the local variable that points to the heap-allocated array is always loaded into a register before use—in this case *eax*. In Listing 3, instances of this pattern are found in lines 9, 11, and 16. The H-LOAD rule is important because it recognizes this pattern by storing the pointer value in φ. The H-STORE rule then uses φ in its policy $P_{\text{heapcheck}}$ to check that if φ points to a base array address in the heap then the data being stored must fit within the array's bounds. If the policy does not evaluate to true as asserted in H-STORE then the program execution terminates in error.

3.2 Example execution

To demonstrate how the new semantics, policies, and notation work together, we analyze lines 6–13 of the SIMPIL program in Listing 3. Table 4 decomposes the calculations of each line in the execution of the program. For demonstration, we make up the values of *ebp*, *edi*, and get_input(*malloc*). Starting on line 6, get_input(*malloc*) returns a pointer to the array in heap memory which has value 200. H-INPUT marks 200 in H_μ as a pointer to a block of heap memory and also maps 200 in H_ρ to the value of *edi*—the size of the memory block allocated. Line 7 stores the pointer as a local variable on the stack (address *ebp* − 1 = 100). Nothing is written to the heap and execution proceeds. Line 8 stores the integer value 2 as a local variable on the stack (address *ebp* − 2 = 99). Line 9 loads the pointer to the heap array into *eax* and invokes H-LOAD. This rule assigns φ the value of *ebp* − 1. Then, on line 10, the value 10 is written to memory at the location referenced by *eax*. This invokes the H-STORE rule, which verifies that the statement satisfies the premise for $P_{\text{heapcheck}}$. Lines 11–13 perform similar actions except *eax*, the pointer to the heap array, is offset by one. The result after line 13 is a heap-allocated array with two elements: 10 and 20.

Line	Statement	Δ	μ	H_μ	H_ρ	Rule	φ
	start	$\{ebp \to 101,$ $edi \to 3\}$	$\{\}$	$\{\}$	$\{\}$		$\bot$
6	$eax :=$ get_input(malloc)	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-INPUT	$\bot$
7	store$(ebp-1, eax)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{100 \to 200\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-STORE	$\bot$
8	store$(ebp-2, 2)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-STORE	$\bot$
9	$eax :=$ load$(ebp-1)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-LOAD	100
10	store$(eax, 10)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-STORE	100
11	$eax :=$ load$(ebp-1)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-LOAD	100
12	$eax := eax + 1$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 201\}$	$\{200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$		100
13	store$(eax, 20)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 201\}$	$\{201 \to 20$ $200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-STORE	100

Table 4: Computations for lines 6–13 from Listing 3.

With the execution trace in Table 4, we extract the information listed in Section 3.1. First, we construct a set that holds the pointer values to all heap-allocated memory blocks (in this case there is only one).

Let $X := \{x \in H_\mu \mid (\exists y \in \mu)[x = \mu[y] \wedge H_\mu[x]]\}$. Then for $X = \{200\}$ we know:

- How much memory was allocated to the program heap: $H_\rho[200] = 3$
- The pointer in memory to the data structure: $X = \{200\}$
- The total number of elements possible to allocate:
  - Since all values are 32-bit unsigned integers, each element's size is the same: $H_\rho[200]/1 = 3/1 = 3$.
- The data stored in the array:
  - $\{(x, \mu[x]) \mid 200 \le x < 200 + H_\rho[200] \wedge \mu[x] \neq \bot\} = \{200 \to 10, 201 \to 20\}$
- If any access to the heap array is out-of-bounds (possible buffer overflow).
  - $P_{\text{heapcheck}}$ passed for each check. The program did not terminate in error.

To demonstrate the array bounds checking we present an example of a failed heap check in Table 5. The program statements are identical to the example in Table 4 except line 12 where the offset is changed from 1 to 4, which is outside the bounds of our heap. In the evaluation of H-STORE on line 13, $P_{\text{heapcheck}}$ is passed the following parameters: $P_{\text{heapcheck}}(204, 200, 3, \mathbf{T})$. This evaluates to false so the execution terminates in error. This example demonstrates the ability for this technique to detect possible heap buffer overflows.

Line	Statement	Δ	μ	H_μ	H_ρ	Rule	φ	pc
$\vdots$	$\vdots$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 200\}$	$\{200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$		100	12
12	$eax := eax + 4$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 204\}$	$\{200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$		100	13
13	store$(eax, 20)$	$\{ebp \to 101,$ $edi \to 3,$ $eax \to 204\}$	$\{204 \to 20$ $200 \to 10$ $100 \to 200$ $99 \to 2\}$	$\{200 \to \mathbf{T}\}$	$\{200 \to 3\}$	H-STORE	100	$error$

Table 5: Computations demonstrating detection of writing outside of the heap-array's bounds.

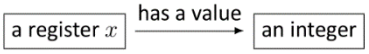
4. Integration with data model

Next, we integrate a knowledge representation data model with our extension of SIMPIL. The data model acts as a "back end" for the formal language. In this way we can store and retrieve knowledge, tasks, and concepts beyond the *malloc* and dynamic array examples presented here. We require a data model that can integrate with the specification of the formal language through concise mappings. For this we use Spivak and Kent's Ontology Logs, or *ologs*.

4.1 Ologs

Developed by David Spivak and Robert Kent, the olog is a category-theoretical model for knowledge representation (Spivak & Kent, 2012). Although closely related to other data modeling concepts such as relational databases and the Resource Description Framework (RDF) (W3C, 2014), ologs have distinct advantages due to their grounding in categories. Unlike RDF graphs, olog diagrams can commute—meaning that two paths in a diagram with the same start and end points are equivalent. Also, using category theory, we can define functors between the data model and the formal language. This will be the primary method of integrating and instantiating data.

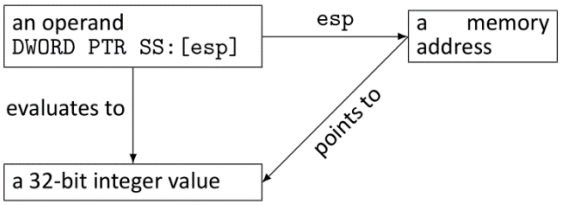
An olog is a category that models a real-world situation by connecting objects with arrows and labeling them. The primary objects are *types* and *aspects*. *Types* are the objects that represent abstract concepts. A type is depicted as a box with a singular indefinite noun phrase (Spivak & Kent, 2012). *Aspects* describe how objects can be measured, viewed, or regarded. An aspect of *A* is $A \rightarrow B$, where *B* is a set of possible result values for *A*. For example:



(1)

This olog is read as "a register *x* has a value which is an integer."

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 data models do not. Diagram 2 presents an example assembly operand for referencing a value from the stack.

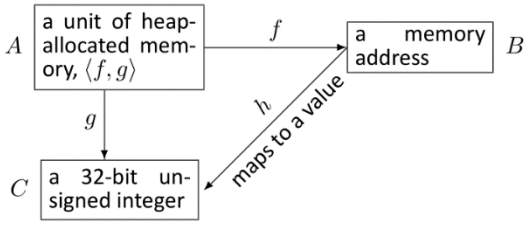


(2)

This diagram is commutative; whichever path is taken leads to the same result.

4.2 Instantiating data

Equally important to representing concepts is instantiating data for those concepts. The olog in Diagram 3 represents data stored in a block of heap memory with an address and corresponding value. We denote the olog by *H*.



(3)

Instance data of *H* can be represented in a table much like a relational database. Table 6 contains instance data for H where each column is a type in the olog.

A	B	C
0	200	10
1	201	20
2	202	30

Table 6: Instanced data for olog in Diagram 3.

Spivak & Kent (2012) describe how to load an olog with instance data by constructing a set-valued functor from an olog $C \rightarrow$ **Set**, where **Set** denotes the category of sets. For the olog *H* in Diagram 3, an *instance* of *H* is a

functor $I : H \rightarrow$ **Set** that consists of:

- a set $I(x)$ for each object x in H,
- a function $I(f) : I(x) \rightarrow I(y)$ for each arrow $f : x \rightarrow y$ in H , and
- for each fact (path-equivalence) declared in H (there is only one: $g = f; h$), an equality of functions: $I(g) = I(f); I(h)$.

We use Table 6 to construct such a functor with concrete values. For each object (A, B, and C) there are sets

$$I(A) = \{0, 1, 2\},$$
$$I(B) = \{200, 201, 202\},$$
$$I(C) = \{10, 20, 30\}.$$

For each arrow in H (f, g, and h) there are functions

$$I(f) : I(A) \rightarrow I(B) : \{0 \mapsto 200; 1 \mapsto 202; 2 \mapsto 202\},$$
$$I(g) : I(A) \rightarrow I(C) : \{0 \mapsto 10; 1 \mapsto 20; 2 \mapsto 30\},$$
$$I(h) : I(B) \rightarrow I(C) : \{200 \mapsto 10; 201 \mapsto 20; 202 \mapsto 30\}.$$

And the function equality $I(g) = I(f); I(h)$ is evident by the composition of functions.

To construct instance functors from the operational semantics of our formal language we define a special policy. For example, in the heap-array semantics in Section 3, we add a function to the computation step of H-STORE that is invoked any time data is stored in an allocated block of heap memory.

Let the special function Inst(a, v, H) := $I : H \rightarrow$ **Set**, and take as arguments an address a, a value v, and an olog H (the same one referenced in Diagram 3). It constructs an instance functor as follows:

$$I(A) = \{max(A) + 1\} \qquad I(f) : \{max(A) + 1\} \mapsto \{a\} \qquad \text{assert } I(g) = I(f); I(h)$$
$$I(B) = \{a\} \qquad I(g) : \{max(A) + 1\} \mapsto \{v\}$$
$$I(C) = \{v\} \qquad I(h) : \{a\} \mapsto \{v\}$$

The final assertion is the return value. When added to the computation step of H-STORE, it is in logical conjunction with $P_{\text{heapcheck}}$ like so: Inst(v_1, v_2, H) $\wedge$ $P_{\text{heapcheck}}(v_1, \mu[\varphi], H_\rho[\mu[\varphi]], H_\mu[\mu[\varphi]])$ = **T**. The conjunction ensures that the operation is indeed a valid write to a heap data structure.

4.3 Representing knowledge

We can use ologs to represent the concepts introduced in Section 3 such as semantic rules and policies. These concepts, along with example instance data from Section 3, are presented in Figure 2. The instance data is abbreviated for space and clarity.

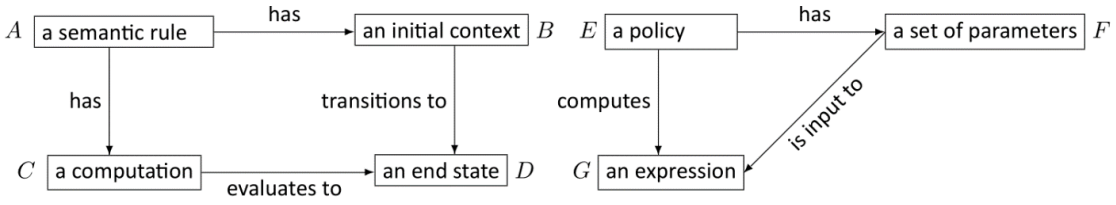
Let's suppose there is a new analysis we want to perform with extended SIMPIL. After tracing the allocation of dynamic memory, we want to see if the memory block is freed during the execution of the program. Consider the assembly code in Listing 4 for freeing a pointer to a block of heap-allocated memory. The lifted SIMPIL code—after converting to 32-bit representation—is in Listing 5.

```
mov     rax,QWORD PTR [rbp-0x10]
mov     rdi,rax
call    <free@plt>
```

Listing 4: Assembly instructions for freeing dynamically allocated memory.

```
1 eax := load(ebp - 2)
2 edi := eax
3 eax := get_input(free)
```

Listing 5: The instructions from Listing 4 lifted to SIMPIL.



A	B	C	D
H-INPUT	$\Delta \vdash$ get_input$(src)\ldots$	v is input from $src\ldots$	v
H-LOAD	$\Delta \vdash$ load $e\ldots$	$v' = \mu[v]\ldots$	v'
H-STORE	store$(e_1, e_2)\ldots$	$\mu' = \mu[v_1 \leftarrow v_2]\ldots$	$\mu', pc+1, \ldots$

E	F	G
P_{malloc}	src	$src =$ 'malloc'
$P_{\text{heapcheck}}$	$dest, base, size, isheap$	$isheap \longrightarrow (dest < base + size)$
ρ_{malloc}	src, val	$\begin{cases} val & P_{\text{malloc}}(src) \\ \bot & \text{otherwise} \end{cases}$
Φ	$prt, cond$	$\begin{cases} ptr & cond \\ \bot & \text{otherwise} \end{cases}$

Figure 2: Ologs and instanced data for operational semantics and policies.

To incorporate this into the analysis, we add a new policy, $P_{\text{free}}(src)$:= (src = 'free'), as well as a new rule identical to H-INPUT except with an extra computation step. We call it H-FREE-INPUT with added computation:

$$H'_\mu = H_\mu[edi \leftarrow (H_\mu[edi]_1, [P_{\text{free}}(src) \wedge edi = \mu[\varphi] \wedge H_\mu[edi]_1])],$$

where $H_\mu[x]_i := \pi_i(H_\mu[x])$ for (i = 1, 2) picks out the first and second element of the ordered pair through the left (π_1) and right (π_2) projections at $H_\mu[x]$ for key value x. This redefines H_μ as a map from an address to an ordered pair (a, d) where a and d are boolean values representing allocation and deallocation. Now we add these as entries of instance data to the rule and policy ologs from Figure 2 like so:

A	B	C	D
H-FREE-INPUT	$\Delta \vdash$ get_input$(src)\ldots$	$H'_\mu = H_\mu[edi \leftarrow (H_\mu[edi]_1, [P_{\text{free}}(src) \wedge \ldots$	v

E	F	G
P_{free}	src	$src =$ 'free'

Next, we group these rules and policies together so that they associate with the same "task." Perez & Spivak (2015) do this by mapping ologs—thus constructing a system of ologs connected by morphisms similar to the *aspects* defined in Section 4.1. This technique is useful for our data model because we can define a "task" as a system of ologs and pick out the ologs that execute that task with the proper operational semantics. A general system for a task looks as follows: D ⇜ S ⇝ P, where D are ologs relating to data, S are ologs relating to operational semantics, and P are ologs for policies. Applying this to the "free task", we define two functors $F^\flat$: S → D and $F^\sharp$: S → P. Since we are relating ologs with instantiated data, we construct instance functors I : S → **Set**, J : P → **Set**, and K : D → **Set**, which are represented by the following tables:

$I(A) :=$

a semantic rule
H-INPUT
H-LOAD
H-STORE
H-FREE-INPUT

$J(E) :=$

a policy
P_{malloc}
$P_{\text{heapcheck}}$
P_{free}
ρ_{malloc}

$K :=$

a map
H_μ
H_ρ

Then, in order to map only the information associated with the "free task" we choose the following morphisms for $F^\flat$ and $F^\sharp$:

$$\lceil \text{a map} \rceil \xleftsquigarrow{\text{writes if a heap address is freed to}} \lceil \text{a semantic rule} \rceil \xrightsquigarrow{\text{confirms a system call to free via}} \lceil \text{a policy} \rceil,$$

denoted by α and β going from left to right. Then, the natural transformation $p : I \Rightarrow K \circ F^\flat$ conforming to α and the natural transformation $q : I \Rightarrow J \circ F^\sharp$ conforming to β are as follows:

$(\alpha, p) :=$

a semantic rule	**writes if a heap address is freed to**
H-FREE-INPUT	H_μ

$(\beta, q) :=$

a semantic rule	**confirms a system call to free via**
H-FREE-INPUT	P_{free}

This mapping picks out the information for the "free task" and demonstrates how conceptual data can be

organized. For more on olog mappings and the underlying category theory, see (Perez & Spivak, 2015, Section 3).

5. Related work

Performing security analyses on binary programs using formal languages and semantics is not new. BitBlaze (Song et al., 2008) and the Binary Analysis Platform (BAP) (Brumley et al., 2011) are examples of related projects that perform binary analysis using an intermediate language. Both platforms provide many security-relevant program analysis tools such as control flow and dependency graphs, symbolic execution, program verification, and taint analysis. Their central feature is that all analyses are based off of the operational semantics of an intermediate language. This is also a limitation and it is one of the reasons why we incorporate a knowledge representation data model to expand the scope of possible analyses.

6. Conclusions

This paper presents a first and novel approach to the concept assignment problem in assembly language. This research is relevant to cyber security researchers and those attempting to automate the comprehension of compiled software from assembly language representations. By automating aspects of comprehension, cyber defense and security analysts can utilize tools that identify concepts and programming constructs in fielded software to increase understanding and support analysts in discovering the intentions of binary code. Given an assembly language program, we can use the formal techniques presented in this paper to prove the existence of a feature in that program. Before now, no formal methods were available that modeled the concept assignment problem in assembly language. Through this research, our goal is to improve automation and support for reverse engineers and cyber security professionals to handle threats faster, more effectively, and at greater scale.

As noted by Biggerstaff et al. (1994), concept assignment is a hard problem and automating it requires a rich knowledge base of concepts from the problem domain. Thus, the formal method presented in this paper is limited by the richness of the knowledge base. If the semantics for a concept do not exist, then it cannot be formally shown to exist in a given program. Therefore, more analyses—including all relevant semantics and policies—need to be constructed for recognizing concepts in assembly code. Immediate examples that follow from this paper include recognizing heap-allocated `structs`, heap-allocated arrays of `structs`, and file and input manipulation. More advanced examples involve composing simpler analyses together to recognize classes of vulnerabilities from binary code such as buffer overflows, use-before-free errors, null pointer dereferencing, and format string errors. Additionally, despite lifting to an intermediate language, architecture differences may still be a limitation for recognizing concepts. For instance, on some implementations the parameter for `malloc` may be passed as a stack value instead of via the `edi` register. There is nothing in the current approach that automatically detects this difference in parameter assignment.

References

Biggerstaff, T., Mitbander, B. & Webster, D. (1994). Program understanding and the concept assignment problem. *Communications of the ACM*. 37(5), 72–82.

Brumley, D., Jager, I., Avgerinos, T. & Schwartz, E. J. (2011). BAP: A binary analysis platform. In: *International Conference on Computer Aided Verification, Snowbird, UT, USA, 14-20 July 2011*. Berlin, Heidelberg, Springer. pp. 463–469.

Bryant, A. (2012). Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations. PhD thesis. Air Force Institute of Technology, Ohio, USA.

Bryant, A., Mills, R., Peterson, G. & Grimaila, M. (2011). Software reverse engineering as a sensemaking task. *Journal of Information Assurance and Security*. 6(1), 483–494.

Bryant, A., Mills, R., Peterson, G. & Grimaila, M. (2013). Top-level goals in reverse engineering. *Journal of Information Warfare*. 12(1), 32–43.

Chen, K. & Rajlich, V. (2000). Case study of feature location using dependence graph. In: *Proceedings of the 8th International Workshop on Program Comprehension, Limerick, Ireland, 11 June 2000*. Washington, DC, IEEE. pp. 241–249.

Eagle, C. (2011), *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. San Francisco, No Starch Press.

Fix, V., Wiedenbeck, S. & Scholtz, J. (1993). Mental representations of programs by novices and experts. In: *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems, Amsterdam, Netherlands, 24-29 April 1993*. New York, ACM. pp. 74–79.

GNU. (2014). *GNU Binutils*. Available from: https://www.gnu.org/s/binutils [Accessed 18 September 2016].

GNU. (2016). *GDB: The GNU Project Debugger*. Available from: https://www.gnu.org/s/gdb [Accessed 18 September 2016].

Hex-Rays. (2015) *The IDA Pro Disassembler and Debugger*. Available from: https://www.hex- rays.com/idapro [Accessed 18 September 2016].

Perez, M. & Spivak, D. I. (2015). Toward formalizing ologs: Linguistic structures, instantiations, and mappings.

[Preprint] Available from: http://arxiv.org/abs/1503.08326v2 [Accessed 30 December 2016].

Petrenko, M., Rajlich, V. & Vanciu, R. (2008). Partial domain comprehension in software evolution and maintenance. In: *16th IEEE International Conference on Program Comprehension, Amsterdam, Netherlands, 10-13 June 2008.* Washington, DC, IEEE. pp. 13–22.

Rajlich, V. (2009). Intensions are a key to program comprehension. In: *17th International Conference on Program Comprehension, Vancouver, BC, 17-19 May 2009.* Washington, DC, IEEE. pp. 1–9. Available from: doi:10.1109/ICPC.2009.5090022 [Accessed 30 December 2016].

Schwartz, E. J., Avgerinos, T. & Brumley, D. (2010). 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, Oakland, CA, USA, 16-19 May 2010.* Washington, DC, IEEE. pp. 317–331.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering.* SE-10(5), 595–609.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P. & Saxena, P. (2008). BitBlaze: A new approach to computer security via binary analysis. In: *Proceedings of the 4th International Conference on Information Systems Security (ICISS), Hyderabad, India, 16-20 December 2008.* Berlin, Heidelberg, Springer. pp. 1–25. Available from: http://dx.doi.org/10.1007/978-3-540-89862-7_1 [Accessed 9 January 2016].

Spivak, D. I. & Kent, R. E. (2012). Ologs: A categorical framework for knowledge representation. *PLoS ONE.* 7(1). Available from: doi:10.1371/journal.pone.0024274 [Accessed 9 January 2017].

von Mayrhauser, A. & Vans, A. M. (1994). Comprehension processes during large scale maintenance. In: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 16-21 May 1994.* Los Alamitos, IEEE. pp. 39–48.

W3C. (2014). *RDF Primer*. Available from: https://www.w3.org/TR/rdf-primer [Accessed 18 September 2016].

Wilde, N. & Scully, M. C. (1995). Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice.* 7(1), 49–62.