# Examining Self-Modifying Code

By

Andrew Ivarson

* * * * * * * * *

Submitted in partial fulfillment
of the requirements for
Honors in the Department of Computer Science

UNION COLLEGE

June 2015

# ABSTRACT

IVARSON, ANDREW Examining Self-Modifying Code. Department of Computer Science June 2015.

ADVISOR: Matthew Anderson, John Spinelli

Self-modifying code is used for both good and bad. Software companies use it as a means of hiding a program's internals. This can allow them to prevent hackers from "cracking" the authentication on their software. Self-modifying code is also used in malware to avoid being detected by malware analysts and anti-virus software. It is created when programs write to their own instruction memory space. The power of self-modifying code comes from being difficult to analyze, because traditional analysis tools don't account for changing instruction space. In this project, we implement an algorithm proposed by Anckaert et al.[1] to track self-modification. This algorithm can be used to generate a new type of control flow graph which represents a conservative estimate of all possible execution paths of a program. We implement this algorithm with the goal of identifying its strengths and weaknesses. We also produce visualizations of the resulting control flow graphs that show both positive and negative aspects of this algorithm. Using our visualizations, we have identified levels of sophistication the code in a program can possess for a useful visualization to be generated. We have defined sophistication in code to be related to the amount of jump and write instructions, and programs that have more complex sequences of interrelated jumps and writes yield far less useful graphs. On the other hand, programs that contain simple sequences of jumps and writes, or none at all, are easily visualized to the full understanding of the program's internals.

**Table of Contents:**



Andrew Ivarson, Class of 2015.

## 1. Introduction

Self-modifying code has been used in malware since the 1990s [2]. It is a broad domain which encompasses several sub-domains, but in general, self-modifying code is code that changes itself as it runs. This is accomplished in several ways, such as writing to instruction memory, dynamically compiling code, and representing code as dynamic pointers to parts of memory other than instruction memory. Of these three methods, this paper is concerned with the first: writing to instruction memory.

Self-modifying code is a powerful means of hiding a program's internals, because it is particularly difficult to determine behavior by reading through the instructions [1]. When a program uses self-modifying code, its actual properties are unknown until the self-modification happens. Before running such code, it could appear harmless, only to modify itself into exhibiting harmful behavior. In this project, the code in question is the code that is found in executable files: binary code. Here, this code is interpreted as assembly instructions.

This type of code has also been used in positive moral cases. Self-modifying code has been used in programming for hardware that had a limited amount of memory. It was used to optimize code for this type of limited-memory hardware [1]. Essentially, instructions over-wrote themselves instead of having more instructions that occupy more spaces in memory. Self-modifying code, in this case, allowed programmers for this type of hardware to write larger programs than were otherwise possible. It is also commonly used to protect software from being reverse engineered [3, 4].

In this paper, we are concerned with analyzing executable files containing self-modifying code. We describe our novel implementation that is based on a theoretical model in *A Model of Self-Modifying Code* [1]. Applications of our work could be of two forms. One, in diagnosing

malicious software which uses self-modifying code, and a second in maintaining and debugging of non-malicious software that uses self-modifying code.

## 2. Related Work and Background

Much research has been done in developing tools to analyze binary files in order to identify malware. There are many subtasks to identifying malware in a general sense. These tasks include code factorization, disassembly, analysis using heuristics based on patterns of known malicious behavior, and many others [5]. Overall, to accomplish these subtasks, there are two general approaches to analyzing malware.

### Static Analysis

Static Analysis is an attempt to analyze a file without executing it. The main application that we are concerned with is in attempting to trace all possible edges in a program. By edges, we mean to calculate each possible instruction that could follow any given instruction. If we create this list of all possible edges, then a superset of all possible traversals through the program will be represented by this set of all edges.

Because when one uses static analysis one does not run the program, the analysis does not need to perform each instruction, therefore it can perform much more quickly than an approach that needs to process and execute each instruction. The efficacy of static analysis alone has decreased over time [5], because it has become commonplace to obfuscate binary files to the point where a program's initial static disassembly is nothing like the code it actually

executes [5]. Because of its diminishing efficacy, analysts must carefully identify problems that can be solved using static analysis.

## Dynamic Analysis

Dynamic Analysis is analyzing a binary file by running it and observing the results of executing instructions. The field of developing these tools as seen enormous growth as the complexity of malware has increased [6], and is divided into two subfields:

## Simulated Dynamic Analysis

Simulated dynamic analysis is running executable files within a simulated environment. Essentially, these tools involve building a simulation of an operating system that runs on a simulation of a processor. Difficulties in this approach mainly stem from precision of simulation. Instruction sets, such as x86, have a massive number of legal instructions, and processors each have specific behaviors for handling errors in instructions. Malware, now, is often written to attempt to detect simulated dynamic analysis systems. If the malware manages to detect a simulator, it can change itself to not exhibit malicious behavior. Malware can detect simulators by issuing instructions that cause the processor to produce incorrect results, often due to hardware bugs. Because processors are such large, complex systems, these bugs are difficult for simulator writers to account for. Malware authors have the considerably easier task of identifying and exploiting these edge cases. Simulators need to accept billions of possible instructions (which is a lowball, considering instruction words are often 64 bits long), they often do not account for edge-case behaviors like these bugs and are therefore detectable by malware.

## Virtualized Dynamic Analysis

Virtualized dynamic analysis is a newer approach than simulated dynamic analysis. Virtualized dynamic analysis was created by processor companies (such as Intel) releasing virtualization software. This software enables instructions to be run within a controlled environment, but on the hardware itself. This controlled environment can allow analysts to, for example, step through each instruction individually, and because the processor actually runs the instruction, none of the edge-case behavior found in Simulated dynamic analysis applies [7]. Ether [7], for example, is a tool to analyze malware using virtualization, and it boasts 100% success of transparency on a set of 25,000 recent malware samples, all of which attempted to detect simulation.

The shortcoming of dynamic analysis is the speed. Simulating an entire processor, or stepping through a million-instruction binary file takes a considerable amount of time. Literature [6] suggest that the most effective analysis approach combines both static and dynamic analysis. Such an approach could exploit both the speed of static analysis and the fine granularity of dynamic analysis.

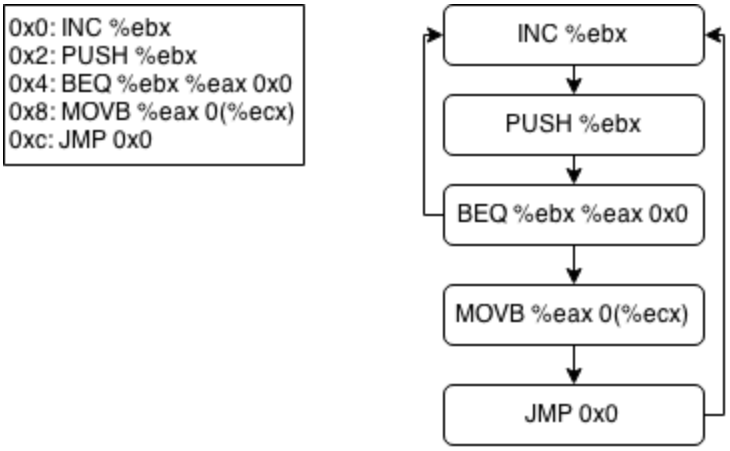



Figure 1: An example of source code (left) and its corresponding CFG (right)

## Background

Our research is an implementation based on a solution to a static analysis problem - the static analysis of self-modifying code. The goal for this type of static analysis is to build a control flow graph (CFG) of the binary file. A graph, here, is a set of nodes and edges. An edge is a directional indication of which nodes may follow which nodes. Nodes, here, are basic blocks: an instruction or sequence of instructions that occupy adjacent memory addresses. A control flow graph is a flowchart which represents all possible edges among the instructions in a program. In Figure 1, the CFG represents a non-self-modifying program, and thus it clearly represents all possible paths through the program. The problem [5] with drawing CFGs of programs with self-modifying code is that there is no point in which a static image of the binary file reveals all of the instructions that are run. This results in over-written instructions never being accounted for in the final image by all tools [6].

Anckaert et al [1] have proposed a solution to this problem by suggesting a State-Enhanced Control Flow Graph (SE-CFG) which is a conservative visualization for a program. An SE-CFG stores all possible instructions including all possible versions of those that

are overwritten. No tools are capable of building this style of CFG, therefore in this work we have implemented a tool to build SE-CFGs. This model is conservative by considering each possible instruction a valid node following an edge leading to that memory address.

In a traditional CFG, a node is a basic block. Basic blocks are sequences of instructions that are always executed in succession, and are in adjacent memory locations . For example, in FIgure 1, `INC %ebx` and `PUSH %ebx` could be put together to form a basic block, because `PUSH %ebx` is always executed after `INC %ebx`. Edges indicate which instructions lead to which instructions. An SE-CFG expands this definition of a node to also account for the memory address. A node is still a basic block, but the graph arranges these nodes according to their addresses. This allows for instructions beginning at the same address to be clearly recognized as over-written instructions. Edges maintain the same definition as in a traditional CFG. This expanded definition of a CFG allows the graph to show more than one instruction at a given memory address, and combined with the algorithm and data structures specified in Anckaert et al's paper these SE-CFGs can be constructed to represent self-modifying programs.

We will now demonstrate the SE-CFG in greater detail using Anckaert et al's example. This example shows their process of how a program is analyzed to generate an SE-CFG. To explain their example, we will begin by explaining their instruction set.

| Assembly | Binary | Semantics |
|---|---|---|
| `movb` *value to* | `0xc6` *value to* | set byte at address *to* to value *value* |
| `inc` *reg* | `0x40` *reg* | increment register *reg* |
| `dec` *reg* | `0x48` *reg* | decrement register *reg* |
| `push` *reg* | `0xff` *reg* | push register *reg* on the stack |
| `jmp` *to* | `0x0c` *to* | jump to absolute address *to* |

Figure 2: The instruction set used in Anckaert et al's example program[1]

Note the instruction set in Figure 2. It is based on x86, and has general-purpose instructions found in any typical modern instruction set: an instruction, `MOVB`, to write to memory, instructions `INC` and `DEC` to change register values, an instruction, `PUSH`, to push to the stack and an instruction, `JUMP`, to transfer execution. We make the assumption that the only instructions that can affect the state of program memory are `MOVB` instructions. We make this assumption with the goal of the algorithm in mind: the algorithm statically analyzes binary files to produce a graph of traversals through the program which contains self-modifying code. Because the basis for nodes are basic blocks, the only instructions that affect the shape of the graph are jump instructions (create discontinuities in the traversal) and instructions that write to memory (creating new instructions). The other three instructions (`INC`, `DEC`, and `PUSH`) are, for all intents and purposes, syntactic sugar to make the example program look more realistic, because none of them affect the state of program memory or of the control flow. `INC`, `DEC`, `PUSH`, and `MOVB` instructions can all be

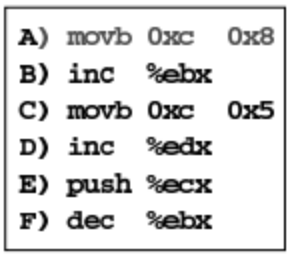
[1] [1] page 2

| Address | Assembly | Binary |
|---|---|---|
| 0x0 | movb 0xc 0x8 | c6 0c 08 |
| 0x3 | inc %ebx | 40 01 |
| 0x5 | movb 0xc 0x5 | c6 0c 05 |
| 0x8 | inc %edx | 40 03 |
| 0xa | push %ecx | ff 02 |
| 0xc | dec %ebx | 48 01 |

Figure 3: Anckaert et al's example program[2]

put in basic blocks. JUMP instructions, on the other hand, transfer execution to non-adjacent memory locations. This means that they cause the algorithm to create edges and additional nodes on the graph.

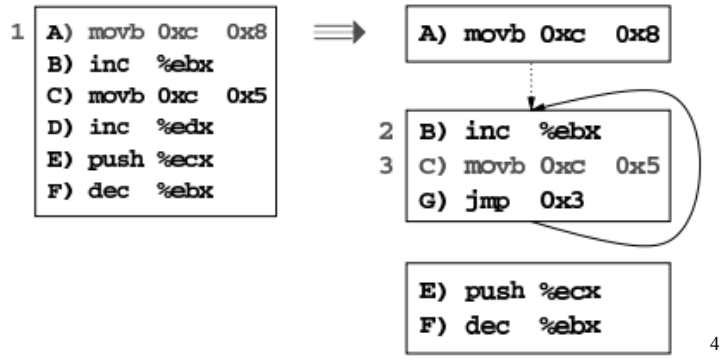
In the example program in Figure 3, we first observe that self-modification occurs and that there are no jumps. A CFG of this program would, therefore, look like:



[3]

In this diagram, instructions are labeled with letters rather than addresses. Note that there are no JUMP instructions, so the entire program can be put in a single basic block. Because of this, appears to be just a sequence of instructions. If we consider the memory overwrite in instruction A, we will consider the following diagram:

[2] [1] page 3
[3] [1] Figure 1



[4]

Note that in addition to the last diagram, we now have a second diagram, and numbers have been added next to instructions A, B, and C. These numbers indicate that instructions A, B, and C are the first three instructions executed when the program is run. The three-stemmed arrow between the two diagrams indicates that the graph changes after instruction A is executed. This change is because an instruction has been overwritten. Line A overwrites instruction D into this new instruction, G. Instruction G jumps to instruction B. It accomplishes this self-modification by writing the value 0xc to the address 0x8. Instruction D starts at address 0x8, so its opcode is changed from 0x40 to 0xc, while its data byte, 0x3, remains the same. 0xc is the binary opcode for `JUMP` in this instruction set, so the instruction changes to `jmp 0x3`. This makes it a jump to the second instruction, B. The graph then, because we are still considering a traditional CFG, change instruction D into instruction G. Numbers *1*, *2*, and *3* in this diagram indicate the order of execution for the first three instructions. Now we will consider instruction C, another self-modifying instruction.

[4] [1] Figure 1

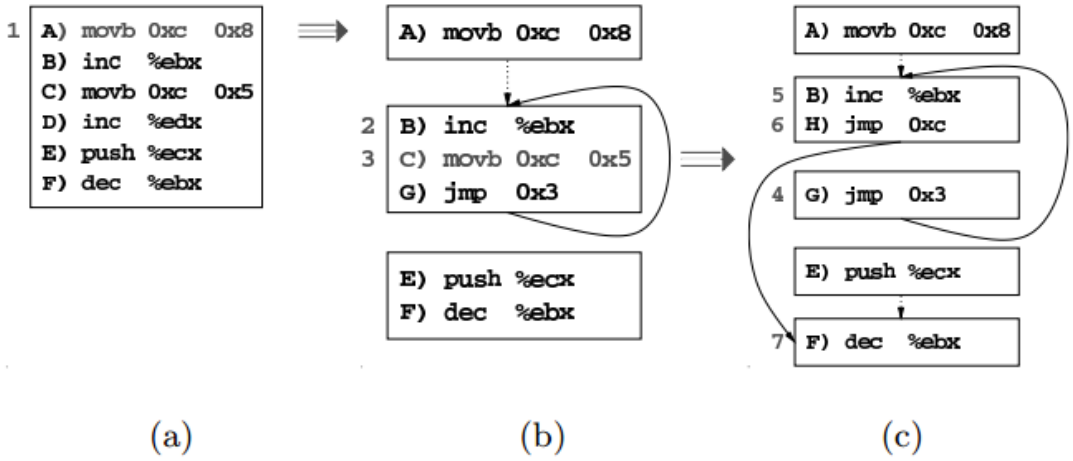



Figure 4: (a) the initial CFG; (b) the CFG after considering instruction A's self-modification; (c) the CFG after considering instruction C's self-modification.[5]

In Figure 4, we note the self-modification in instruction C. Instruction C overwrites the address 0x5 with the value 0xc. From the our last self-modification, we note that this changes the opcode of the instruction to *jmp*. This makes the instruction at 0x5, whose second bit is 0xc, *jmp 0xc*. This is a jump to the last instruction in the program, instruction F. CFGs (a), (b), and (c) from Figure B2c are, again, traditional CFGs that do not keep track of overwritten code. Anckaert et al then introduce the SE-CFG, which keeps track of all versions of each instruction.

Figure 5 synthesizes the three CFGs shown previously, by showing all of the possibilities in one graph. It demonstrates the original static image (A, B, C, D, E, F executed in succession) as well as the other considerations determined in the previous paragraphs. It shows that instruction H and C both existed at one point, as well as instructions D and G both existing at some point, but this graph does not have any information in it that explains how instructions C and H, and D and G are related unless we refer back to the memory map when the example is introduced. Anckaert et al now propose data structures and an algorithm to use these data

[5] [1] Figure 1

structures to produce an SE-CFG, which has properties to represent all possible instructions, including those which are only possible after original instructions are overwritten.

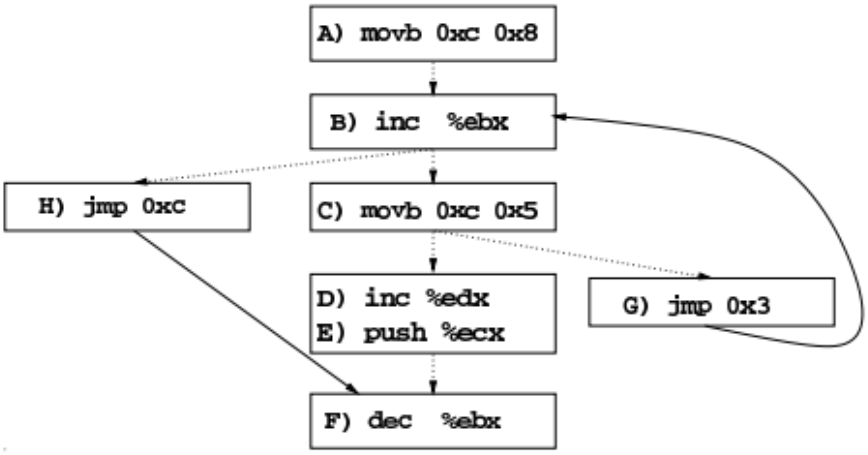



Figure 5: An SE-CFG of the running example[6]

The first data structure that Anckaert et al introduce is the *CodeByte*, which stores all possible values that a given address in memory can have for one program. For example, address 0x5 (the first byte of instructions C and H in Figure 5) in our running example can have the values 0xc6 (when it is a `MOVB` instruction) and 0xc (when it is a `JUMP` instruction). Therefore, the CodeByte representing address 0x5 contains the values 0xc6 and 0xc.

Anckaert et al then propose an algorithm to construct a list of CodeBytes which represents an entire program as CodeBytes. In this algorithm, which we will refer to as AMB for the remainder of the paper, an executable file is recursed over until all possible traversals are identified.. It first states how to initialize the CodeByte list, simply iterating through the program's instruction memory address space adding a CodeByte for each address. It then adds the address's corresponding value to the CodeByte. An image of the CodeBytes of this example

[6] [1] Figure 2

just after they are initialized is, therefore, just a list of addresses with one corresponding value stored for each address. AMB then recurses over input program adding new values to be associated with these addresses from the initial CodeByte list. Because AMB's full purpose is to gather enough information to produce an SE-CFG, it also keeps track of control flow changes, and assigns each instruction it encounters a list of other instructions that it can lead to. For example, instruction B in Figure 5 can lead to both instruction C and instruction H, so AMB keeps track of both of these possibilities to be represented in the graph. Now we will explain the algorithm line by line. Refer to Figure 6 throughout this explanation. The *while* loop at line 1 begins the computational process. The condition on the while loop refers to changes in instruction memory. The new instruction memory writes refer to the results of `MOVB` instructions, and those whose results have not yet been written to the CodeBytes. If a `MOVB` instruction's result is already stored in CodeBytes, new information is not written to the CodeByte. How changes to CodeBytes are processed is discussed later in AMB.

```
1     while (new instruction memory writes are occurring)
2           CodeBytes.setAllUnvisited()
3           recurse(start_address)
4     proc recurse(addr):
5           if (addr.isVisited()) then return
6           addr.setVisited()
7           for each: instruction starting at addr
8                 handleInstruction(instruction, addr)
9     proc handleInstruction(instruction, addr):
10          if instruction is a MOVB instruction
11                CodeBytes[addr].add(val written by instruction)
12                instruction.targets.add(next address)
13                recurse(next address)
14          else if instruction is a JUMP instruction
15                instruction.targets.add(target of instruction)
16                recurse(target of instruction)
17          else
18                instruction.targets.add(next address)
19                recurse(next address)
```

Figure 6: The AMB algorithm

The first instruction in the while loop, `CodeBytes.setAllUnvisited()`, is how the recursive algorithm figures out its base case to end its recursions. Each time an address in memory is visited by AMB, it is marked as visited. If AMB encounters a visited instruction, then it returns. This is shown in line 5. Each time an instruction is reached for the second time, the recursion ends and AMB checks for new writes. If new writes occurred, then it recurses again, otherwise it ends.

In the *recursive* function, AMB parses actual instructions and handles information for CodeBytes. *Recursive* begins by checking the base case: if the current instruction is visited, then return. It then marks the current address as visited and continues with nested for loops.

The outermost for loop iterates over every possible instruction that can start at the current address. In the running example, since the base address of the first instruction is 0xc6, we know that the instruction must be a `MOVB`, because 0xc6 is the opcode for `MOVB`. Because the CodeBytes have, at this point, only been initialized, the only possible instruction to start is `MOVB 0xc 0x8`. Now, the *handleInstruction* function saves CodeByte data, saves control transfer data, and initiates the next recursion.

First, AMB iterates over all possible CodeByte writes that the current instruction *ins* can perform. In the example, *ins* can only perform one write, assigning the value *0xc* to the address *0x8*. As this is the only write, the instruction in the for loop is only executed once. This instruction, codebyte[w].add(v) adds the value 0xc to the CodeByte storing the information for address 0x8. Now that a write instruction has occurred, the CodeBytes list has changed. Where it used to have only one possible value for address 0x8, it now has two: it's original value, and the value written by the *movb* instruction just discussed.

Now *Recursive* is called on the next address, 0x0 + 2, because *movb* is a 3-byte instruction in this instruction set. Because this first instruction caused a new change to the CodeBytes, even after the recursion ends from finding a visited instruction, the entire program will be recursed over again to check for more changes that could be caused by the change in the first recursion. Eventually, the algorithm will output a set of complete CodeBytes, containing all possible values for all possible bytes of memory:[7]

| 0x0 | 0x1 | 0x2 | 0x3 | 0x4 | 0x5 | 0x6 | 0x7 | 0x8 | 0x9 | 0xa | 0xb | 0xc | 0xd |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c6 | 0c | 08 | 40 | 01 | c6 | 0c | 05 | 40 | 03 | ff | 02 | 48 | 01 |
| | | | | | 0c | | | 0c | | | | | |

The point of filling this data structure is to better inform the SE-CFG, so that it contains important information about the state of program memory when each instruction is executed. Below is an enhanced SE-CFG of the running example that contains all of the information from the CodeBytes as well as the graph derived earlier.

Figure 7 contains every instruction as well as the corresponding CodeBytes for each instruction. For example, instruction A has the CodeBytes {0x0: [0xc6], 0x1: [0x0c], 0x2: [0x8]} attached to it. Instructions H and C share the Codebytes {0x5: [0xc6, 0x0c], 0x6: [0x0c]}, so these CodeBytes are linked to both instructions in Figure 7.

[7] [1] Figure 4

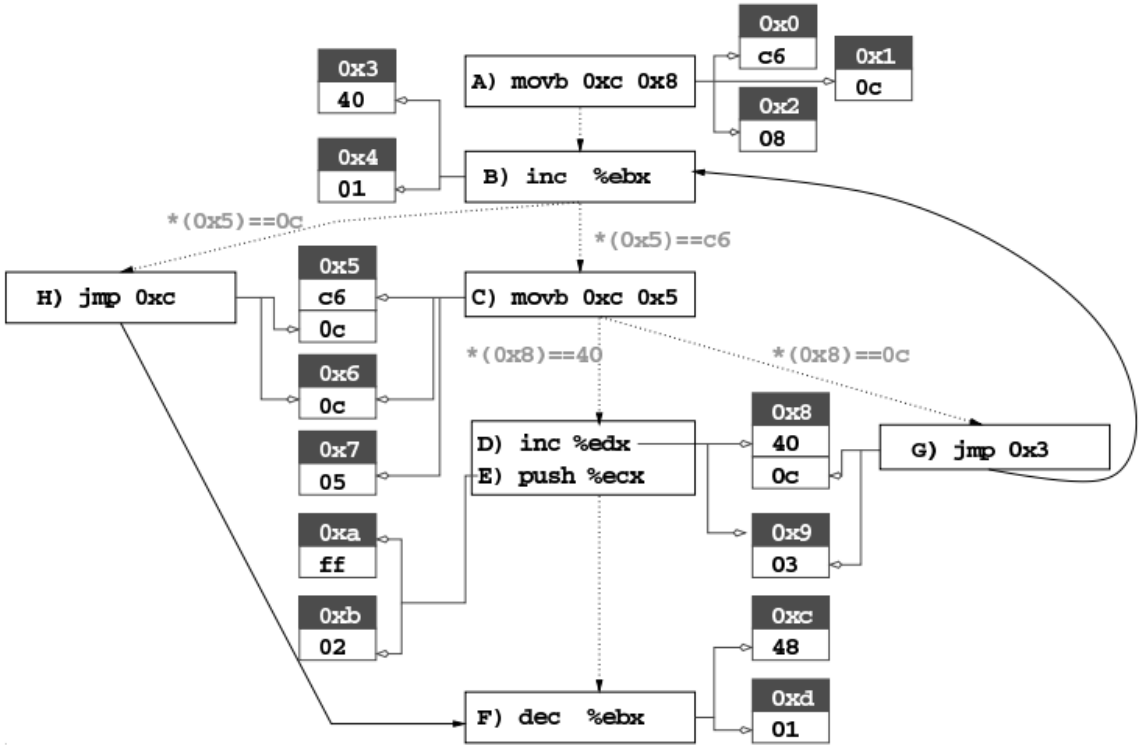



Figure 7: An SE-CFG that includes the contents of instruction memory as well as instructions that correspond to the contents of instruction memory.[8]

This representation is past the point that our research attempts to recreate Anckaert et al's approach. We find this representation too cluttered. We aim to synthesize Figure 7 with Figure 5 to produce a less cluttered representation.

[8] [1] Figure 6

## 3. Implementation

We have implemented this model in Java. Our main goal was to expand the test cases described in the paper, and create a data-driven extensible implementation. In our approach we have three goals which come from the nature of Anckaert et al's paper: it is a theoretical framework, not a description of a specific implementation. Our goals are built to expand AMB from a theoretical model to an implementation.

Our goals are:

### 1. Program examples

We aim to be able to demonstrate the idea using several, programmatically generated examples.To address this, the implementation needs to allow users to write their own programs. Our implementation provides a way to write and input binary files. This allows us to test AMB on instruction sets we develop, and future researchers to expand on ours. These programs are limited to the instruction sets, that we will now discuss.

### 2. Instruction Set Examples

Similar to 1, because it is a theoretical paper, only one program is shown. This limits the results in the paper to the instruction set used in the example. To address this, our implementation allows users to define their own instruction sets. Our implementation provides a way to create instruction sets limited by what we define as an abstract syntax, a syntax to represent a generalization of possible syntaxes. This abstract syntax represents all possible instructions that this algorithm can analyze. Because the algorithm is only concerned with

tracking instructions that write to memory and instructions that change control flow, the abstract syntax contains only three possible types of instructions: write instructions, jump instructions, and skip instructions. These instructions are sufficient for reporting on the AMB approach because they are the only types of instructions that AMB's approach is concerned with. The only instructions addressed in the algorithm are write instructions and control flow change instructions (ie, jumps), so the rest can be considered instructions to be skipped over by the algorithm, or skip instructions.

A write instruction is an instruction that writes to the instruction memory of the program. Instructions that change valuables in the general working memory of a program, like registers or RAM, are not counted as write instructions. This is because of how we have defined syntax for write instructions. They can never take dynamic information as input (such as the stack, registers, or RAM), so none of this dynamic information can affect write instructions.

A jump instruction is an instruction that goes to a target address. It can be the next address, it can be before the current address, or it can be after the current address. This is how we define an instruction that changes the control flow of the program.

A skip instruction is any type of instruction not named above. This includes instructions that perform actions such as stack manipulation, heap manipulation, register manipulation, ALU operations, NOPs, setting and checking register flags, etc.

We will now define an instruction set. This implementation defaults to the most simple instruction set given the problem this is addressing. This instruction set contains the following instructions:

Instruction Set: Basic.is

| opcode | #Bytes | name | abstract syntax |
|---|---|---|---|
| 00 | 3 | MOVB | WRITE |
| 01 | 2 | INC | SKIP |
| 02 | 2 | JMP | GOTO |

In the table above, Basic.is is defined as an instruction set made up of three instructions. The first instruction, whose opcode is 00, is named MOVB. It is, according to the abstract syntax, a write instruction. This means that it writes a value into memory. The only memory that can be written to in this implementation is instruction memory, all instructions that write outside are considered invalid. If we can find interesting results using their simplistic example, we believe future researchers can find even more interesting results by expanding instruction sets to entire assembly languages and, therefore, being able to input actual executable files.

Note that the instruction set definition above has no possible semantic detail other than the abstract syntax. It doesn't, for example, allow users to change the order of the data bytes. We have decided to make semantics of our instruction sets entirely internal, and do not allow customization of instruction semantics. Below, find the semantics for the Basic.is instruction set:

`MOVB AA BB`- assign the value AA to the location BB

`INC XX` - increment a register XX
`JMP XX` - jump to location XX

The semantics of all instruction sets including a WRITE instruction (such as `MOVB`), a GOTO instruction (such as `JUMP`), and a SKIP instruction (such as `INC`) will be identical. WRITE instructions will always use the first byte after the opcode as the value to be written, and the second byte after the opcode as the location to write to. All GOTO instructions will use the first data byte as the target location. The decision to limit instruction sets this way is, again, due to an attempt to maintain the simplicity of the example. This inhibits future work and would require significant source code changes to expand instruction sets beyond these simplistic semantics.

Given an instruction set, then, our implementation parses programs written for these instruction sets, analyzes the program using the AMB algorithm, and creates a visualization.

## 3. Visualizations

In the paper, Anckaert et al [1] do not describe a programmatic means of building their graphs. Their main contribution to the field of binary analysis is introducing the SE-CFG, and they only propose a way to build the data structure that this graph could depend on. To address this, the implementation must programmatically generate visualizations based on the results of the AMB Algorithm.

Given our goals for expansion, we have designed an implementation with three main tasks:

1) Parse
    a) Import an instruction set defined by the user
    b) Convert input binary files into assembly code according to the instruction set
2) Analyze the file using the AMB Algorithm
    a) Generate the data structure defined in the paper, CodeBytes
    b) Generate a data structure we have created, the InstructionByte, which we will explain later.
3) Visualize results of the analysis
    a) Draw a graphic representation of the data structures generated in step 2.

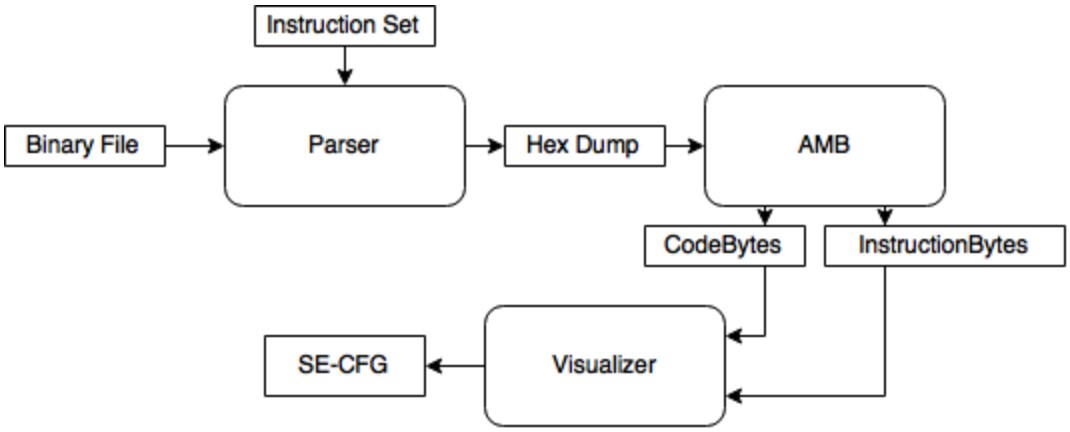



Figure 8: A flowchart of the input and output from each component of our implementation. First, the parser is given Binary Code and an Instruction Set. The parser then sends Hex Dump to AMB, which sends CodeBytes and InstructionBytes to the Visualizer. Finally, the Visualizer outputs an SE-CFG.

Figure 8 represents these three tasks visually. We will now examine these steps more closely through an example. For our input instruction set, we will use Basic.is, and the following example binary code:

```
SampleProgram.bin:
     01   01
     00   02   01
     01   01
```

**Parse:**

To parse a program, our implementation refers to the specified instruction set. In this case, the instruction set is Basic.is. Given this instruction set, it parses each token as a 2-digit hexadecimal number. For the ease of writing programs, we have written our parser to analyze line-by-line, instead of merely parsing 2-digit hexadecimal tokens. It parses these line-by-line instructions into a string of 2-digit hexadecimal tokens, so that they are not analyzed with any preconceived assumptions about what is allowed in the original program. Illegal instructions are, for example, legal in the original program. It is up to the AMB step to identify which instructions are impossible, and only up to the parser to prepare a hex dump and send instruction set information on.

## AMB Analysis

The implementation begins the analysis process by initializing the data structures: CodeBytes and InstructionBytes. The CodeBytes are initialized as a list. Each index in this list contains information that the memory address referred to by index contains. For SampleProgram.bin , then, the initial CodeBytes list is: [0x1, 0x1, 0x0, 0x2, 0x1, 0x1, 0x1]. Note that this list does not contain the visited flags, as defined in the Background section. The first object in the list, 0x1, is the first byte in memory, the opcode of the `INC` instruction, which is 0x1. The second is 0x1, the address of the register in the first instruction. Then 0x0, the opcode of the `MOVB` instruction.

An initial look at the CodeBytes reveals the following instructions:

```
INC  0x1
MOVB 0x2  0x1
INC  0x1
```

Each of these indices in the CodeBytes list, in reality, contains a list. The point of CodeBytes is to be able to track multiple possible values for each memory address. During the remaining of the analysis process these CodeByte indices will be added to as self-modifying code is encountered.

An InstructionByte is defined as a base address, a list of all possible instructions beginning at that base address, as well as a list of all memory locations targeted by all of these instructions. The first InstructionByte is initialized as:

0x0: [INC 0x1]
Targets: [ ]

The list of InstructionBytes is, then, initialized as as list of these given whatever addresses and instructions are in the program.

Note that Figure 9, our algorithm, is slightly different from the AMB algorithm. The choices we have made to change this algorithm are due to unclarity in the original algorithm for implementation. In Anckaert et al's explanation of the algorithm, they do not refer to an equivalent data structure of the InstructionByte.

```
0     proc main()
1           while (new instruction memory writes are occurring)
2                 CodeBytes.setAllUnvisited()
3                 recurse(start_address)
4     proc recurse(addr):
5           if (addr.isVisited()) then return
6           addr.setVisited()
7           for each: instruction starting at addr
8                 handleInstruction(instruction, addr)
9     proc handleInstruction(instruction, addr):
10          if instruction is a MOVB instruction
11                CodeBytes[addr].add(val written by instruction)
12                instructionBytes[addr].targets.add(next)
```

```
13              recurse(next address)
14        else if instruction is a JUMP instruction
15              instructionBytes[addr].targets.add(target)
16              recurse(target)
17        else
18              instructionBytes[addr].targets.add(next)
19              recurse(next address)
```

Figure 9: Our (slightly modified) AMB algorithm

They discuss *instructions* and *codebytes*, but do not describe a precise means of storing these instructions in parallel to the CodeBytes. This is the motivation for introducing InstructionBytes, and requires a change to the algorithm to account for them, instead of Anckaert et al's *instructions*.

Given our initialized set of CodeBytes and InstructionBytes, the algorithm begins at address 0x0 and computes the full set of CodeBytes and InstructionBytes.

## Visualizations:

The visualize step is an attempt Anckaert et al's example graphs from Figure 5 and Figure 7. This visualization and their visualization both uses a top-bottom layout, with the first memory address at the top, and last memory address at the bottom. Both visualizations also put instructions with the same base address on the same horizontal line. Anckaert et al's visualization uses a different scheme for drawing new instructions on overwritten base addresses. They maintain the original instructions in the center column, while putting new instructions to the left and right of the center. Our visualization keeps the original instruction on the far left, and it adds new instructions to the right of old instructions. We have implemented this visualization using Java's Swing library, and example graphs will be discussed in the results section. We originally considered using other libraries, such as Graphviz [9] and other tools

using the DOT language, but found the Swing library a more efficient use of time. Given more time, Graphviz would be ideal.

We have discussed our implementation and design decisions for running the AMB algorithm on user-written programs, using user-written instruction sets. Next, we discuss our results and evaluation.

## 4. Results and Evaluation

The results of this implementation will be judged based on the accuracy of the visualizations. If the visualizations produce correct SE-CFGs of the input programs then the implementation is considered correct. Additionally, our results will demonstrate positive and negative aspects of AMB.

We will begin by demonstrating a simplistic example program. This program is three increments to a register. CodeBytes never need to be updated here, and AMB does not need to recurse more than once because to memory never occur.

Figure 10 shows predictable results. There should be no edges other than from one instruction to the next because there are no jumps, and there should not be any additional instructions because there are not any writes.

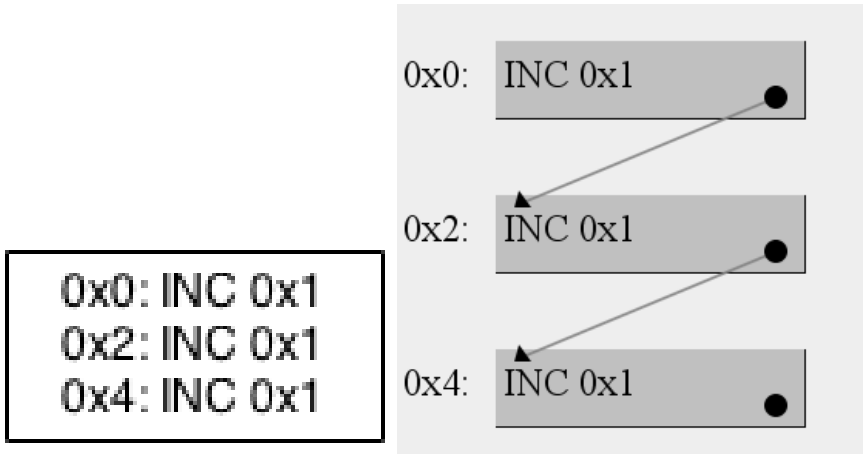



Figure 10: An example program (left) and its visualization (right) to be visualized and run through AMB

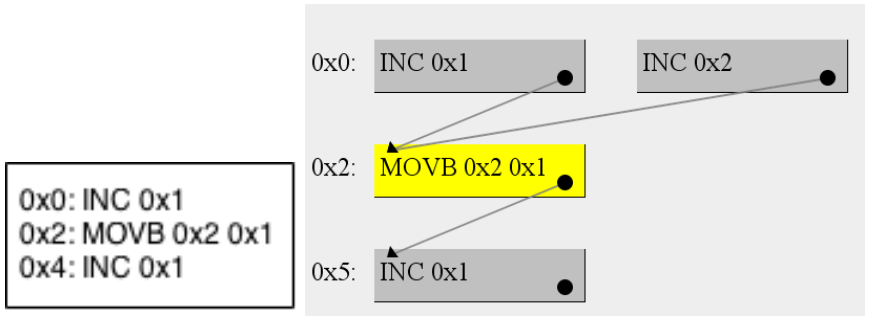



Figure 11: Example program number two (left), with no jumps and a single write instruction. Also its SE-CFG (right)

The program in Figure 11 should result in one instruction being added due to write instruction. The write instruction adds the value 0x2 to the CodeByte holding the data for 0x1. Again, Figure 11's visualization is a predictable result. `MOVB 0x2 0x1` should result in there being two versions of the second byte of the first instruction, which is shown in the graph. Again, the graph also correctly shows the control flow.

Now we move onto more complex examples. The visualizations are meant to be conservative. This has been maintained since the Figure 6. The definition of being conservative

we use here is to always draw an arrow to each possible instruction at the target address. Instead of storing which exact instruction our instructions lead to, we only store their target addresses. In Figure 12, we see a huge example of this conservative approach producing a large, complex graph. This program is a contrived attempt to have AMB recurse into itself as many times as possible. It does not infinitely recurse, but every time a write instruction occurs, it alters the write instruction that will happen in the next recursion through the code. We classify the programs in Figure 10 and Figure 11 has having a low level of sophistication, because they either do not self-modify, or only cause a small change to instruction memory. This results in very few edges being added to the graph.

```
0x0: MOVB 0x5 0xe
0x3: MOVB 0xe 0xb
0x6: MOVB 0xb 0x8
0x9: MOVB 0x8 0x5
0xc: MOVB 0x5 0x2
```

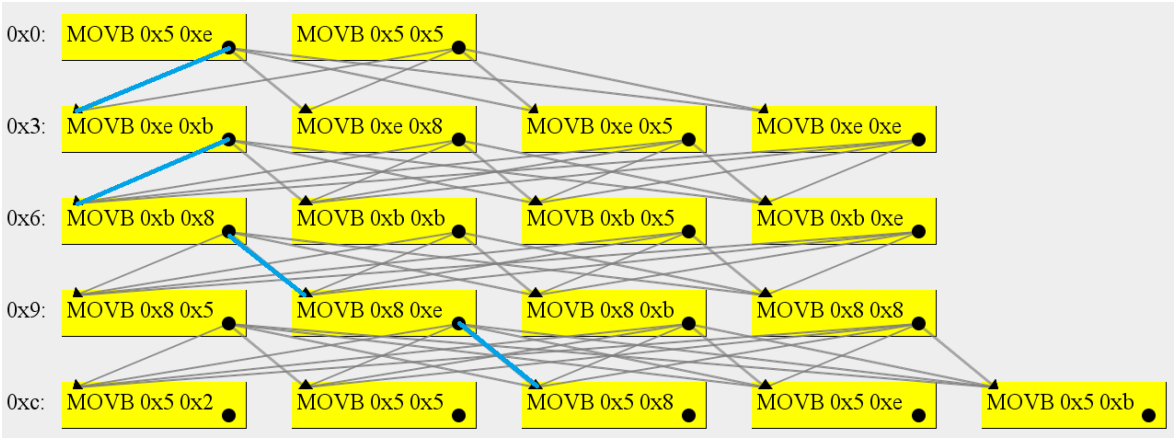



Figure 12: A third example program (above) and its SE-CFG (below). The edges highlighted in blue[9] indicate actual path of execution.

Now that we have demonstrated success in visualizing basic behavior, we will move on to more complex examples. Figure 12 is such an example. Here, we have a program of five

[9] Note that in Figure 12, as well as all future figures, we have *hand-drawn* the highlighted execution path for the sake of reading comprehension.

instructions, none of which are jumps, create a graph of 19 nodes and 60 edges. In fact, the graph never jump back to the beginning, so we know that only one of the instructions at each of these addresses is ever executed. The fact that each of these instructions has an edge leading to each instruction in the following memory address is due to our conservative estimate: we do not consider the state of memory at the time of each instruction. Instead, we consider all possible states of program memory. We classify the program in Figure 12 has having a high level of sophistication because each `MOVB` instruction causes another `MOVB` instruction to behave differently. This causes AMB to add many nodes and edges to the graph, because the outermost while loop in AMB must iterate several times.

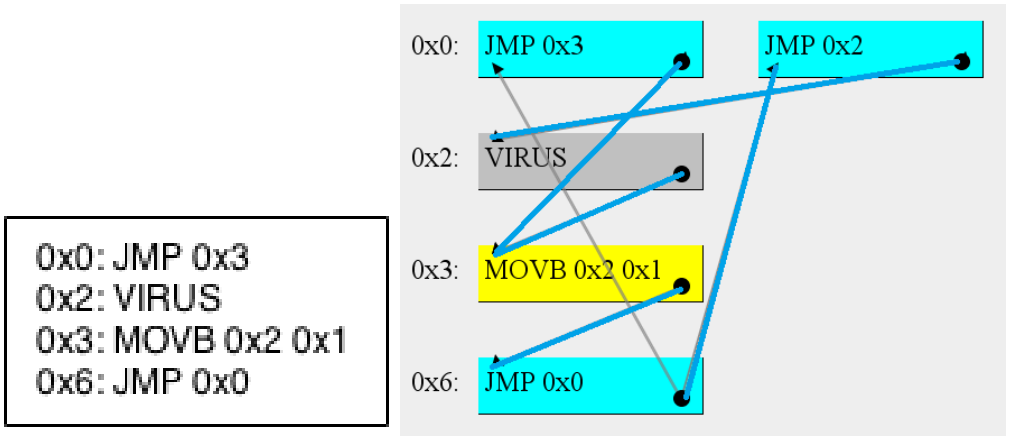



Figure 13: Another program and its SE-CFG. The edges highlighted in blue indicate actual path of execution.

For Figure 13, AMB is expected to output two instructions at address 0x0, because the second byte is overwritten by instruction 0x3. The two instructions at 0x0 should be: JMP 0x3 and JMP 0x2, because the MOVB instruction stores the value 0x2 to the second byte of the jump instruction. Then, because of the instruction 0x6: JMP 0x0, the VIRUS code becomes reachable. The program in Figure 13 is of a moderate level of sophistication. The `MOVB` instruction causes

AMB to add one node and three edges to the graph, and does not cause AMB to iterate the outmost while loop several times. This is because the `MOVB` instruction does not modify a `MOVB`, but instead modifies a `JUMP`.

The visualization in Figure 13 demonstrates what we hope to observe from the program in. There are now two possible instructions at address 0x0, and the JMP instructions both have edges going to their respective targets. This graph demonstrates and allows a reader to derive how the

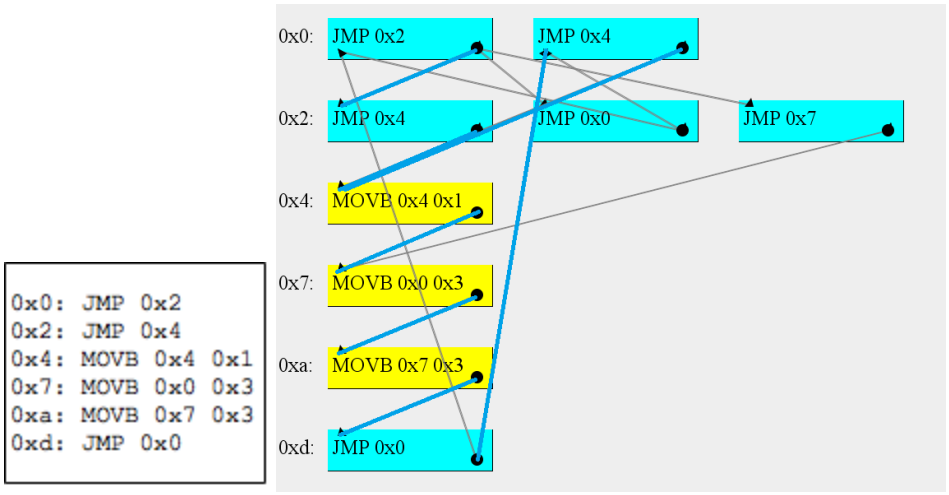



Figure 14: Another example program and its SE-CFG. The edges highlighted in blue indicate actual path of execution.

`VIRUS` code is reachable, by comparing the initial state of byte 0x1, 0x3, to the latter state of 0x1, 0x2. The target of the final jump instruction, 0x0, has two instructions. Two arrows are drawn because of how we define our conservative estimate: `JUMP` instructions do not consider the current state of instruction memory to determine their possible targets. Rather, they consider all possible states of instruction memory, as determined by AMB building the CodeBytes and InstructionBytes.

Next, consider Figure 14. AMB adds three nodes and five edges. Its actual path of execution is much easier to work out than Figure 12, and this is because the complexity of the `MOVB` instructions is quite low. When `MOVB` instructions do not change other `MOVB` instructions, the number of outer while loop iterations is significantly lower. This is the factor that ramps up the complexity of the graphs and exploits the conservatism of AMB.

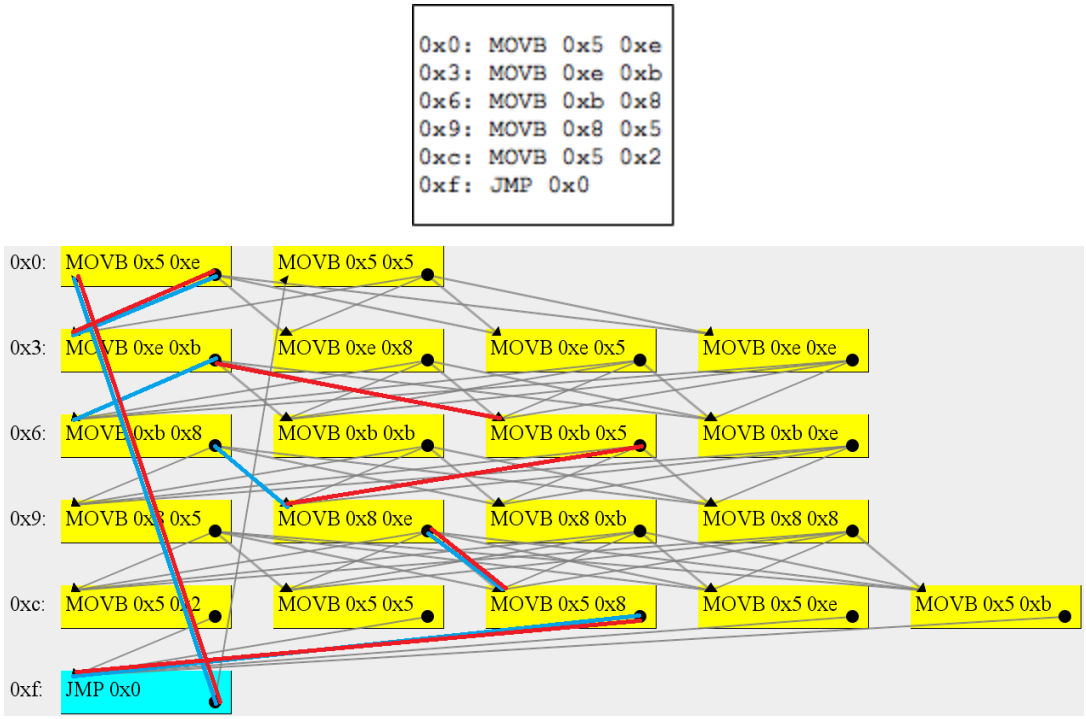



Figure 15: A slightly modified and more complex version of Figure 12. The actual path of execution is indicated in Blue and Red. Blue is the path of the first iteration, and the Red is the path that then iterates infinitely.

Figure 15, which is only different from Figure 12 because of the final `JUMP` instruction, The added instruction does not add significant complexity to the graph. AMB adds one node and

six edges, but this does not require the outer while loop to iterate anymore than it did for Figure 12. Adding the `JUMP` instruction also does not significantly complicate the path through the program. The path of the first iteration remains unchanged, and the second iteration is the same as all future iterations.

```
0x0:  MOVB 0x5  0x11
0x3:  MOVB 0x11 0xe
0x6:  MOVB 0xe  0xb
0x9:  MOVB 0xb  0x8
0xc:  MOVB 0x8  0x5
0xf:  MOVB 0x5  0x2
```

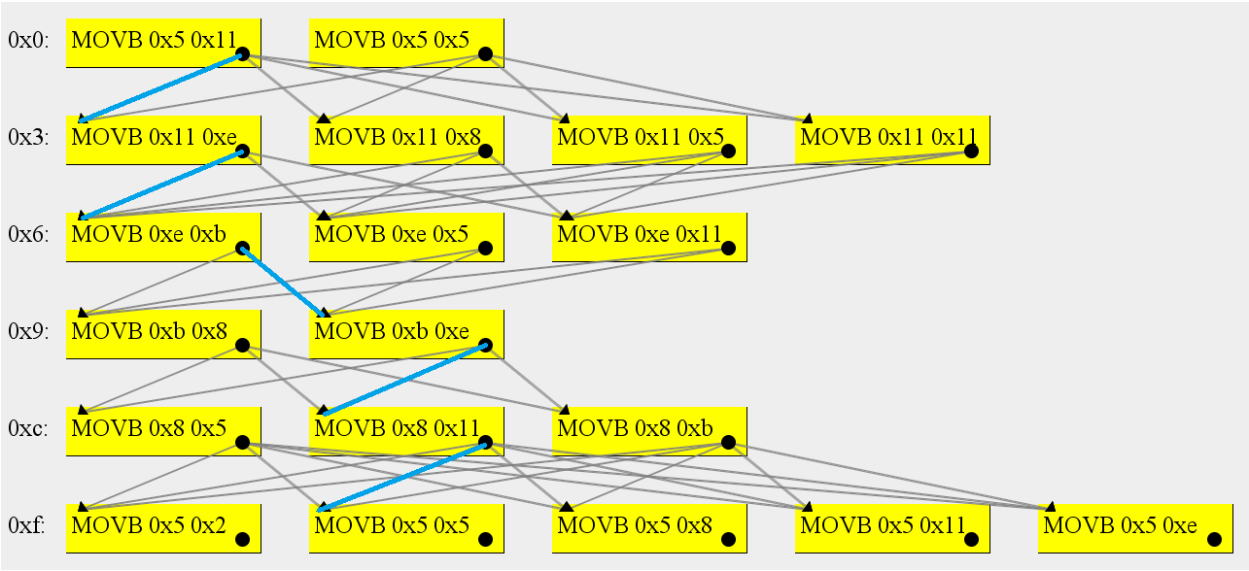



Figure 16: A final example program and SE-CFG. By adding another `MOVB` instruction to Figure 12 and shifting all other instructions, we have decreased the number of nodes and edges.

In Figure 16, our final example program, we demonstrate that simply adding `MOVB` instructions does not increase the complexity of the SE-CFG. Figure 12 has one fewer instructions, and AMB calculates 19 nodes and 60 edges. Figure 16 has an extra `MOVB` instruction, and AMB calculates 19 nodes and 47 edges. This supports our claim that `MOVB` instructions affect the complexity of the graph that AMB calculates, but it indicates that the complexity of the self-modification done by the `MOVB` instructions has a stronger influence than the raw number of `MOVB` instructions. The complexity of self-modification is a result of how many versions of each `MOVB` instruction can exist. If a more versions of a `MOVB` instruction, A, can exist, then each of those versions of instruction A can then create more versions of other `MOVB` instructions. This causes the outer while loop of AMB to do more iterations.

## 5. Conclusion and Future Work

We have implemented Anckaert et al's algorithm to perform static analysis on binary files containing self-modifying code. We have used the results of this algorithm to construct a new type of control flow graph, a state-enhanced control flow graph, which accounts for all possible states of instruction memory as well as all possible data paths through the program. We have tested our implementation on several example programs and demonstrated our implementation to produce accurate results which reveal positive and negative aspects of Anckaert et al's algorithm. On the positive side, our visualizations have demonstrated successful visualizations of programs that would otherwise be difficult to represent with a control flow graph, such as Figures 12, 13, 14, 15, and 16. On the negative side, our visualizations have demonstrated the extreme growth to the graph that the conservative decisions made by the

algorithm, as shown in Figures 12, 14, 15, and 16. We have concluded that the conservative estimate taken is ideal only when simpler self-modification occurs. Programs in which `MOVB` instructions modify other `MOVB` instructions, thereby causing other `MOVB` instructions to be modified, create SE-CFGs which obscure the data path more than reveal it.

Future work will include improvements to Anckaert et al's conservative estimate, with the goal of removing the vast majority of impossible edges and nodes, as in Figure 12. We hope to apply optimization to remove code that is either unreachable from the onset, or code that is created through write instructions but cannot ever be executed. This can be accomplished by always keeping track of the current state of instruction memory as the algorithm traverses the program. In a way, our proposed solution here is a synthesis of static and dynamic analysis. Being able to perform both, and any level of synthesis between the two would also be hugely beneficial. This will allow for the targets of jump instructions to always be identified statically. Additionally, future work should include expanding our instruction sets to the point where analysis and SE-CFG representation of actual binary files (such as malware) code can be done.

## 6. References

[1] B. Anckaert, M. Madou, and K. De Bosschere. 2006. A model for self-modifying code. In Proceedings of the 8th international conference on Information hiding (IH'06), Jan L. Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee (Eds.). Springer-Verlag, Berlin, Heidelberg, 232-248.

[2] The Leprosy-B virus (1990) http://familycode.atspace.com/lep.txt. Visited May 10, 2014.

[3] D. Aucsmith. Tamper resistant software: an implementation. Information Hiding, LNCS, 1174:317–333, 1996

[4] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting selfmodification mechanism for program protection. In Proc. of the 27th Annual International Computer Software and Applications Conference, pages 170–181, 2003.

[5] K Roundy and B. Miller. 2013. Binary-code obfuscations in prevalent packer tools.ACM Comput. Surv. 46, 1, Article 4 (July 2013), 32 pages. DOI=10.1145/2522968.2522972 http://doi.acm.org/10.1145/2522968.2522972

[6] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44, 2, Article 6 (March 2008), 42 pages. DOI=10.1145/2089125.2089126 http://doi.acm.org/10.1145/2089125.2089126

[7] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In Proceedings of the 15th ACM conference on Computer and communications security (CCS '08). ACM, New York, NY, USA, 51-62. DOI=10.1145/1455770.1455779 http://doi.acm.org/10.1145/1455770.1455779

[8] S. Chae, A. Majumder, and M. Gopi. 2012. HD-GraphViz: highly distributed graph visualization on tiled displays. In Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing (ICVGIP '12). ACM, New York, NY, USA, , Article 43 , 8 pages. DOI=10.1145/2425333.2425376 http://doi.acm.org/10.1145/2425333.2425376

## 7. Appendix

```
/**
 * Paints a control flow graph given a Program analyzed by the AMB
Algorithm
 *
 * @author Drew Ivarson
 * @version May 21, 2015
 */

import java.awt.Graphics2D;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Rectangle;
import java.awt.Color;
import java.awt.RenderingHints;
import java.util.ArrayList;
import java.awt.geom.AffineTransform;
import static java.awt.geom.AffineTransform.*;
```

```
import java.lang.Math.*;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class GraphPainter extends JPanel
{

    private Program p;
    private ArrayList<InstructionByte> inBytes;


    public static final int VERTICAL_CENTER = 15;
    public static final int HORIZ_CENTER = 70;
    public static final int VERTICAL_START = 70;
    public static final int HORIZONTAL_START = 100;
    public static final int VERTICAL_DIST = 100;
    public static final int HORIZ_DIST = 250;

    public GraphPainter(Program p1, ArrayList<InstructionByte> insBytes)
    {
        p = p1;
        inBytes = insBytes;
    }

    public void paint(Graphics g)
    {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
        Font font = new Font("Serif", Font.PLAIN, 25);
        g2.setFont(font);
        g.setColor(Color.BLACK);
        ArrayList<ArrayList<ArrayList<Integer>>> points =
makeLinks(inBytes);
        for (int i = 0; i < inBytes.size(); i++)
        {
            int vertPos = 70+ i*VERTICAL_DIST;
            InstructionByte cur = inBytes.get(i);
            Integer address = cur.getAddress();
            g2.drawString("0x" + Integer.toHexString(address) + ":", 10,
70+ i*VERTICAL_DIST);
            ArrayList<ConcIns> curConcs = cur.getInstructions();
```

```
            for (int j = 0; j < curConcs.size(); j++)
            {
                int horizPos = 100 + j*HORIZ_DIST;
                g.setColor(Color.BLACK);
                g.drawRect(horizPos - 30, vertPos - 30, 200, 50);
                if (curConcs.get(j).isJMP())
                    g.setColor(Color.CYAN);
                else if (curConcs.get(j).isMod())
                    g.setColor(Color.YELLOW);
                else
                    g.setColor(Color.LIGHT_GRAY);
                g.fillRect(horizPos - 30, vertPos - 30, 200, 50);
                g.setColor(Color.BLACK);
                /**
                for (int k = 2; k < points.get(i).get(j).size(); k+=2)
                    {
                        g.setColor(Color.BLACK);
                        /**
                        System.out.println("points.get(" + i + ").get(" +
                                           j + ").get(" + k
                                           + ")" +
points.get(i).get(j).get(k));

                        int y1 = points.get(i).get(j).get(0);
                        int x1 = points.get(i).get(j).get(1);
                        int y2 = points.get(i).get(j).get(k) + 5;
                        int x2 = points.get(i).get(j).get(k+1) + 10;
                        /**
                        System.out.println("\nDrawing: " + i +
                                    "\nx1 = " + x1 +
                                    "\ny1 = " + y1 +
                                    "\nx2 = " + x2 +
                                    "\ny2 = " + y2);

                        g2.drawOval(x1 - 1, y1 - 1, 15, 15);
                        g.setColor(Color.GRAY);
                        g2.drawLine(x1 + 6, y1 + 6, x2, y2);
                        g2.drawLine(x1 + 5, y1 + 7, x2 - 1, y2 + 1);
                        g.setColor(Color.BLACK);
                        //g2.fillOval(x2 - 1, y2 - 1, 15, 15);
                        g.setColor(Color.GRAY);
                     }
```

```
                */
            }

        }



        //System.out.println("\nPoints: " + points);

        //for each instructionByte
        for (int i = 0; i < points.size(); i++)
        {
            g.setColor(Color.BLACK);
            int vertPos = 70+ i*VERTICAL_DIST;
            InstructionByte cur = inBytes.get(i);
            //g2.drawString("0x" + cur.getAddress() + ":", 10, 70+
i*VERTICAL_DIST);
            ArrayList<ConcIns> curConcs = cur.getInstructions();
            //System.out.println("\npoints.get(" + i + ")" +
points.get(i));
            //for each concrete instruction
            for (int j = 0; j < points.get(i).size(); j++)
            {
                int horizPos = 100 + j*HORIZ_DIST;
                /**
                g.setColor(Color.BLACK);
                g.drawRect(horizPos - 30, vertPos - 30, 200, 50);
                if (curConcs.get(j).isJMP())
                    g.setColor(Color.CYAN);
                else if (curConcs.get(j).isMod())
                    g.setColor(Color.YELLOW);
                else
                    g.setColor(Color.LIGHT_GRAY);
                g.fillRect(horizPos - 30, vertPos - 30, 200, 50);
                g.setColor(Color.BLACK);
                */
                //System.out.println("points.get(" + i + ").get(" +
                                        // j + ")" + points.get(i).get(j));


                for (int k = 2; k < points.get(i).get(j).size(); k+=2)
                    {
                        g.setColor(Color.BLACK);
```

```
                        /**
                        System.out.println("points.get(" + i + ").get(" +
                                        j + ").get(" + k
                                        + ")" +
points.get(i).get(j).get(k));
                        */
                        int y1 = points.get(i).get(j).get(0);
                        int x1 = points.get(i).get(j).get(1);
                        int y2 = points.get(i).get(j).get(k) + 5;
                        int x2 = points.get(i).get(j).get(k+1) + 10;

                        //int arrowHeadx1, arrowHeady1, arrowHeadx2,
arrowHeady2;
                        if (y2 < y1)
                        {
                            y2+= 35;
                        }

                        int triSize = 8;
                        int[] xPoints = new int[3];

                        int[] yPoints = new int[3];

                        if (x1 < x2 && y1 < y2)
                        {
                            xPoints[0] = x2 + 3;
                            yPoints[0] = y2 + 3;
                            xPoints[1] = x2;
                            yPoints[1] = y2 - triSize;
                            xPoints[2] = x2 - triSize;
                            yPoints[2] = y2 + 3;
                            //arrowHeadx1 = x2;
                            //arrowHeady1 = y2 - triSize;
                            //arrowHeadx2 = x2 - triSize;
                            //arrowHeady2 = y2;
                        }
                        else if (x1 > x2 && y1 < y2)
                        {
                            xPoints[0] = x2 - 3;
                            yPoints[0] = y2 + 3;
                            xPoints[1] = x2 + triSize;
                            yPoints[1] = y2 + 3;
                            xPoints[2] = x2;
```

```
    yPoints[2] = y2 - triSize;
    //arrowHeadx1 = x2;
    //arrowHeady1 = x2 - triSize;
    //arrowHeadx2 = x2 + triSize;
    //arrowHeady2 = y2;
}
else if (x1 > x2 && y1 > y2)
{
    xPoints[0] = x2 - 3;
    yPoints[0] = y2 - 3;
    xPoints[1] = x2 + triSize;
    yPoints[1] = y2 + 3;
    xPoints[2] = x2;
    yPoints[2] = y2 + triSize;
    //arrowHeadx1 = x2 + triSize;
    //arrowHeady1 = y2;
    //arrowHeadx2 = x2;
    //arrowHeady2 = y2 + triSize;
}
else //x1 < x2, y1 > y2
{
    xPoints[0] = x2 + 3;
    yPoints[0] = y2 - 3;
    xPoints[1] = x2 - triSize;
    yPoints[1] = y2 + 3;
    xPoints[2] = x2;
    yPoints[2] = y2 + triSize;
    //arrowHeadx1 = x2 - triSize;
    //arrowHeady1 = y2;
    //arrowHeadx2 = x2;
    //arrowHeady2 = y2 + triSize;
}
/**
System.out.println("\nDrawing: " + i +
            "\nx1 = " + x1 +
            "\ny1 = " + y1 +
            "\nx2 = " + x2 +
            "\ny2 = " + y2);
*/
//g2.fillOval(x1 - 1, y1 - 1, 15, 15);
```

```
                        g.setColor(Color.GRAY);
                        g2.drawLine(x1 + 6, y1 + 6, x2, y2);
                        g2.drawLine(x1 + 5, y1 + 7, x2 - 1, y2 + 1);

                        g.setColor(Color.BLACK);

                        g2.fillPolygon(xPoints,yPoints,3);

                        //g2.fillOval(x2 - 1, y2 - 1, 15, 15);
                        g.setColor(Color.GRAY);


                     }
                int y1 = points.get(i).get(j).get(0);
                int x1 = points.get(i).get(j).get(1);
                g.setColor(Color.BLACK);
                g2.fillOval(x1 - 1, y1 - 1, 15, 15);
                g2.drawString(curConcs.get(j).toString(), horizPos - 25,
vertPos);
                g.setColor(Color.GRAY);
            }
        }
        /**
        for (int i = 0; i < points.size() ; i++)
        {
            for (int j = 2; j < points.get(i).size()-1; j++)
            {
                int ARR_SIZE = 4;
                int x1 = points.get(i).get(1);
                int y1 = points.get(i).get(0);
                int x2 = points.get(i).get(j+1);
                int y2 = points.get(i).get(j);

                System.out.println("\nDrawing: " + i +
                                   "\nx1 = " + x1 +
                                   "\ny1 = " + y1 +
                                   "\nx2 = " + x2 +
                                   "\ny2 = " + y2);
                g2.drawLine(x1, y1, x2, y2);
                //double dx = x2 - x1;
                //double dy = y2 - y1;
                //double angle = Math.atan2(dy, dx);
```

```
                //int len = (int) Math.sqrt(dx*dx + dy*dy);
                //g2.drawLine(0, 0, len, 0);
                //AffineTransform at =
AffineTransform.getTranslateInstance(x1, y1);

                //at.concatenate(AffineTransform.getRotateInstance(angle));
                //g2.transform(at);



                //g2.fillPolygon(new int[] {len, len-ARR_SIZE,
len-ARR_SIZE, len},
                //      new int[] {0, -ARR_SIZE, ARR_SIZE, 0}, 4);
            }
        }
        */

    }

    public void paintGraph()
    {
        JFrame f = new JFrame();
        f.getContentPane().add(new GraphPainter(this.p, this.inBytes));
        f.setSize(1000, 200 + 150*inBytes.size());
        f.setVisible(true);
    }

    private ArrayList<ArrayList<ArrayList<Integer>>>
makeLinks(ArrayList<InstructionByte> inBytes)
    {
        ArrayList<ArrayList<ArrayList<Integer>>> points = new
ArrayList<ArrayList<ArrayList<Integer>>>();
        //Make an index for each instructionbyte
        for (int i = 0; i < inBytes.size(); i++)
        {
          ArrayList<ArrayList<Integer>> instrBytes = new
ArrayList<ArrayList<Integer>>();
          //Make an entry within each instructionbyte for each
          //Concrete Instruction
          for (int j = 0; j < inBytes.get(i).getInstructions().size(); j++)
            {
             ArrayList<Integer> targets = new ArrayList<Integer>();
             //Entries within each ConcIns for each target
```

```
            //First two are the start (x,y), rest are targets
            for (int k = 0; k < inBytes.get(i).getTargets().size(); k++)
               {
                   targets.add(VERTICAL_START + (15 - VERTICAL_CENTER) +
i*VERTICAL_DIST);
                   targets.add(HORIZONTAL_START + HORIZ_CENTER + 75 +
j*HORIZ_DIST);
                   //System.out.println("\naPoints: " + aPoints);

                   //now find all targets
                   for (int l = 0; l < inBytes.size(); l++)
                   {
                       for (int p = 0; p <
inBytes.get(l).getInstructions().size(); p++)
                       {
                           if
(inBytes.get(l).isTargetOf(inBytes.get(i).getTargets().get(k)))
                           {
                               targets.add(VERTICAL_START + (-5 -
VERTICAL_CENTER) + l*VERTICAL_DIST - 10);
                               targets.add(HORIZONTAL_START - 25 +
p*HORIZ_DIST);
                               //System.out.println("aPoints: " +
aPoints);
                           }
                       }
                   }
               }
             instrBytes.add((ArrayList)targets);
           }
           points.add(instrBytes);
       }
       //System.out.println("\npoints: " + points);
       return points;
   }



   private void drawArrow(Graphics g1, int x1, int x2, int y1, int y2)
   {
       Graphics2D g = (Graphics2D)g1.create();

       int ARR_SIZE = 4;
```

```
        double dx = x2 - x1;
        double dy = y2 - y1;
        double angle = Math.atan2(dy, dx);

        int len = (int) Math.sqrt(dx*dx + dy*dy);

        AffineTransform at = AffineTransform.getTranslateInstance(x1, y1);

        at.concatenate(AffineTransform.getRotateInstance(angle));
        g.transform(at);

        g.drawLine(x1, x2, y1, y2);

        g.fillPolygon(new int[] {len, len-ARR_SIZE, len-ARR_SIZE, len},
                      new int[] {0, -ARR_SIZE, ARR_SIZE, 0}, 4);


    }

}

import java.util.ArrayList;
/**
 * Contains static methods for most parts of the AMB algorithm
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class AMB
{
    private InstructionSet inSet;
    private Program p;
    private ArrayList<InstructionByte> insBytes;

    public AMB(Program p, InstructionSet inSet)
    {
        this.p = p;
        this.inSet = inSet;
        insBytes = new ArrayList<InstructionByte>();
    }

    public Program getProgram()
    {
```

```
    return p;
}

public InstructionSet getInstructionSet()
{
    return inSet;
}

public void analyze()
{
    int oldAmountOfData = getAmountOfData();
    int newAmountOfData = oldAmountOfData;
    do
    {
        p.markAllUnvisited();
        oldAmountOfData = newAmountOfData;
        //System.out.println("In the outer loop");
        recurse(0);
        newAmountOfData = getAmountOfData();

    } while (newAmountOfData > oldAmountOfData);

    //System.out.println(p);
    parseAllInstructionBytes();
    printInsBytes();

    GraphPainter painter = new GraphPainter(p, insBytes);
    painter.paintGraph();

}
private int getAmountOfData()
{
    ArrayList<CodeByte> bytes = p.getBytes();
    int data = 0;
    for (int i = 0; i < bytes.size(); i++)
    {
        data += bytes.get(i).getValues().size();
    }
    return data;
}


/**
```

```
     *
     */
    public void recurse(int begin)
    {
        if (begin == -1)
            return;
        if (begin >= p.getBytes().size())
            return;


        if (p.getBytes().get(begin).visited())
            return;


        p.getBytes().get(begin).markVisited();
        for (int i = 0; i < p.getBytes().get(begin).getValues().size();
i++)
        {
            handleInstruction(begin,
p.getBytes().get(begin).getValues().get(i));
        }
        for (int i = 0; i < p.getBytes().get(begin).getTargets().size();
i++)
        {
            recurse(p.getBytes().get(begin).getTargets().get(i));
        }
    }

    private void handleInstruction(int base, int op)
    {
        AbstractInstruction abs = inSet.getAbstractSyntaxFromOP(op);
        if (abs == AbstractInstruction.WRITECONST)
        {
            for (int k = 0; k <
p.getBytes().get(base+1).getValues().size(); k++)
            {
                   int value = p.getBytes().get(base+1).getValues().get(k);
                   for (int l = 0; l <
p.getBytes().get(base+2).getValues().size(); l++)
                   {
                        int addr =
p.getBytes().get(base+2).getValues().get(l);
```

```
                        if
(!p.getBytes().get(addr).getValues().contains(value))
                        {
                            p.getBytes().get(addr).addValue(value);
                        }
                    }

            }
            int target = base+3;
            if (target >= p.getBytes().size())
            {
                if (p.getBytes().get(base).notATarget(-1))
                    p.getBytes().get(base).addTarget(-1);
            }
            else
            {
                if (p.getBytes().get(base).notATarget(target))
                    p.getBytes().get(base).addTarget(target);
            }
        }
        else if (abs == AbstractInstruction.WRITEREL)
        {
            for (int k = 0; k <
p.getBytes().get(base+1).getValues().size(); k++)
            {
                   int value = p.getBytes().get(base+1).getValues().get(k);
                   for (int l = 0; l <
p.getBytes().get(base+2).getValues().size(); l++)
                   {
                       int offset =
p.getBytes().get(base+2).getValues().get(l);
                       p.getBytes().get(base+offset).addValue(value);
                   }

            }
            int target = base+inSet.getNumBytesFromOpcode(op);
            if (target >= p.getBytes().size())
            {
                if (p.getBytes().get(base).notATarget(-1))
                    p.getBytes().get(base).addTarget(-1);
            }
            else
            {
```

```
                if (p.getBytes().get(base).notATarget(target))
                    p.getBytes().get(base).addTarget(target);
            }
        }
        else if (abs == AbstractInstruction.GOTOCONST)
        {
             for (int k = 0; k <
p.getBytes().get(base+1).getValues().size(); k++)
             {
                 int target = p.getBytes().get(base+1).getValues().get(k);
                 if (target > p.getBytes().size())
                 {
                     System.out.println("Invalid jump attemptint got goto:
" + target);
                     continue;
                 }
                 else if (p.getBytes().get(base).notATarget(target))
                    p.getBytes().get(base).addTarget(target);
             }
        }
        else if (abs == AbstractInstruction.GOTOREL)
        {
            for (int k = 0; k <
p.getBytes().get(base+1).getValues().size(); k++)
            {
                 int target = base +
p.getBytes().get(base+1).getValues().get(k);
                 if (target > p.getBytes().size())
                 {
                     System.out.println("Invalid jump attemptint got goto:
" + target);
                     continue;
                 }
                 else if (p.getBytes().get(base).notATarget(target))
                 {
                     p.getBytes().get(base).addTarget(target);
                 }
            }
        }
        else if (abs == AbstractInstruction.SKIP)
        {
            int target = base + inSet.getNumBytesFromOpcode(op);
            if (target >= p.getBytes().size())
```

```
            {
                if (p.getBytes().get(base).notATarget(-1))
                    p.getBytes().get(base).addTarget(-1);
            }
            else
            {
                if (p.getBytes().get(base).notATarget(target))
                    p.getBytes().get(base).addTarget(target);
            }
        }
        else
        {}
    }
    /**
     * Takes an arraylist of Codebytes that represents a set of data bits
for an instruction.
     *
     * Using the size of the list of codebytes, it returns a list of a list
of integers.
     * The outermost list represents all possible combinations of the
codebyte values.
     * The innter lists are the specific combinations
     *
     */
    private ArrayList<ArrayList<Integer>> getDataBits()
    {
        ArrayList<CodeByte> bytes = p.getBytes();
        ArrayList<ArrayList<Integer>> toRet = new
ArrayList<ArrayList<Integer>>();
        switch (bytes.size())
        {
            case 1:
            {
              for (int i = 0; i < bytes.get(0).getValues().size(); i++)
                    toRet.add(new
ArrayList<Integer>(bytes.get(0).getValues().get(i)));
            };
            case 2:
            {
                ArrayList<Integer> current = new ArrayList<Integer>();
                current.add(0);
                current.add(0);
                for (int i = 0; i < bytes.get(0).getValues().size(); i++)
```

```
                {
                    for (int j = 0; j < bytes.get(1).getValues().size();
j++)
                    {
                        current.set(0, bytes.get(0).getValues().get(i));
                        current.set(1, bytes.get(1).getValues().get(j));
                        toRet.add(new ArrayList<Integer>(current));
                    }
                }
            };
            case 3:
            {
                ArrayList<Integer> current = new ArrayList<Integer>();
                current.add(0);
                current.add(0);
                current.add(0);
                for (int i = 0; i < bytes.get(0).getValues().size(); i++)
                {
                    for (int j = 0; j < bytes.get(1).getValues().size();
j++)
                    {
                        for (int k = 0; k <
bytes.get(2).getValues().size(); k++)
                        {
                            current.set(0,
bytes.get(0).getValues().get(i));
                            current.set(1,
bytes.get(1).getValues().get(j));
                            current.set(2,
bytes.get(2).getValues().get(k));
                            toRet.add(new ArrayList<Integer>(current));
                        }
                    }
                }
            };
            default:
            {}
        }
        return toRet;
    }

    /**
     * Given:
```

```
     * A base address, an opcode, a list of CodeBytes, and an instruction
set, this function:
     *
     * Produces an InstructionByte which encapsulates all possible
combinations of the given opcode
     * with the data codebytes.
     */

    private InstructionByte parseDataBits(int baseAddr, int opcode)
    {
        ArrayList<CodeByte> bytes = p.getBytes();
        InstructionByte ins = new InstructionByte(baseAddr);
        ArrayList<ArrayList<Integer>> allData = getDataBits();
        for (int i = 0; i < allData.size(); i++)
        {
            ins.addInstruction(new ConcIns(opcode, allData.get(i)));
        }
        return ins;
    }

    /**
     * Produces a list of instructionbytes based on the instruction set and
list of codebytes
     * associated with this object
     *
     * @return list of instructionbytes
     */

    private void parseAllInstructionBytes()
    {
        ArrayList<CodeByte> bytes = p.getBytes();
        for (int i = 0; i < bytes.size(); i++)
        {
            //System.out.println("Now looking at CodeByte: " +
bytes.get(i));
            if (bytes.get(i).isInstructionBase())
            {
                InstructionByte current = new InstructionByte(i);
                insBytes.add(current);
                current.setTargets(bytes.get(i).getTargets());

                for (int j = 0; j < bytes.get(i).getValues().size(); j++)
                {
```

```
                        int op = bytes.get(i).getValues().get(j);
                        int numBytes = inSet.getNumBytesFromOpcode(op);
                        if (numBytes == 1)
                        {
                            current.addInstruction(new ConcIns(op));
                        }
                        else if (numBytes == 2)
                        {
                            for (int k = 0; k <
bytes.get(i+1).getValues().size(); k++)
                            {
                                ConcIns cCurrent = new ConcIns(op,
bytes.get(i+1).getValues().get(k));

cCurrent.addDisasm(inSet.disassemble(cCurrent));
                                current.addInstruction(cCurrent);
                                //current.addInstruction(cCurrent);
                            }
                        }
                        else if (numBytes == 3)
                        {

                            for (int k = 0; k <
bytes.get(i+1).getValues().size(); k++)
                            {
                                //System.out.println("K: " + k);
                                for (int l = 0; l <
bytes.get(i+2).getValues().size(); l++)
                                {
                                    ArrayList<Integer> data = new
ArrayList<Integer>();
                                    //System.out.println("L: " + l);
                                    //System.out.println("i + 1: " +
bytes.get(i+1).getValues());
                                    //System.out.println("I + 2: " +
bytes.get(i+2).getValues());
                                    data.add(0,
bytes.get(i+1).getValues().get(k));
                                    data.add(1,
bytes.get(i+2).getValues().get(l));
                                    ConcIns cCurrent = new ConcIns(op,
data);
```

```
cCurrent.addDisasm(inSet.disassemble(cCurrent));
                                    current.addInstruction(cCurrent);
                                }
                            }
                        }
                        else
                        {
                            System.out.println("Apparently we've
encountered a > 3 bytes instruction.   Interesting");
                        }
                    }
                }
            else
            {}
        }
    }




    /**
     * Recursively goes through the algorithm
     *
     * @param the program
     * @param the instruction set
     * @return a list of instruction bytes
     */
    public static ArrayList<InstructionByte> recAlgorithm(Program p,
InstructionSet inSet)
    {
        ArrayList<InstructionByte> ins = new ArrayList<InstructionByte>();
        ArrayList<CodeByte> bytes = p.getBytes();
        for (int i = 0; i < bytes.size(); i++)
        {
            bytes.get(i).markUnvisited();
        }
        int next = nextInstruction(bytes, 0, inSet);
        return ins;
    }

    private static int nextInstruction(ArrayList<CodeByte> b, int start,
InstructionSet inSet)
```

```
    {
        b.get(start).markVisited();
        ArrayList<ConcIns> instructions = new ArrayList<ConcIns>();
        int next = start;
        for (int i = 0; i < b.get(start).getValues().size(); i++)
        {
            int numBytes =
inSet.getNumBytesFromOpcode(b.get(start).getValues().get(i));
            InstructionByte ins = disassembleInstruction(next, i, b, inSet,
numBytes);
            System.out.println(ins);
        }
        return next;
    }

    private static InstructionByte disassembleInstruction(int start, int
startVal, ArrayList<CodeByte> b, InstructionSet inSet, int numBytes)
    {
        InstructionByte ins = new InstructionByte(start);
        int opcode = b.get(start).getValues().get(startVal);
        ArrayList<Integer> data = new ArrayList<Integer>();
        data.add(0);
        data.add(0);
        if (numBytes == 3)
        {
            for (int i = 0; i < b.get(start+1).getValues().size(); i++)
            {
                data.set(0, b.get(start+1).getValues().get(i));
                for (int j = 0; j <  b.get(start+2).getValues().size();
j++)
                {
                    System.out.println("Second data bit: " +
b.get(start+2).getValues().get(j));
                    data.set(1, b.get(start+2).getValues().get(j));
                    ins.addInstruction(new ConcIns(opcode, data));

                }
            }
        }
        else if(numBytes == 2)
        {
            for (int i = 0; i < b.get(start+1).getValues().size(); i++)
            {
```

```
                data.add(b.get(start+1).getValues().get(i));
                ins.addInstruction(new ConcIns(opcode, data));
                data = new ArrayList<Integer>();
            }
        }

        return ins;
    }

    /**
     * Fills in CodeBytes of program p by searching for rewrites
     *
     * @param  p the Program
     * @param inSet the instruction set
     */
    public void calcByteVals()
    {
        int stepper = 1;
        for (int i = 0; i < p.getBytes().size(); i+= stepper)
        {
            for (int j = 0; j < p.getBytes().get(i).getValues().size();
j++)
            {
                int opcode = p.getBytes().get(i).getValues().get(j);
                stepper = inSet.getNumBytesFromOpcode(opcode);
                String current = opcode + " ";
                current += p.getBytes().get(i+1).getValues().get(0);

                if (stepper == 3)
                {
                    current += " " +
p.getBytes().get(i+2).getValues().get(0);
                }
                //System.out.println(current);
                AbstractInstruction abs =
inSet.getAbstractSyntaxFromOP(opcode);
                if (abs == AbstractInstruction.WRITECONST)
                {
                    for (int k = 0; k <
p.getBytes().get(i+1).getValues().size(); k++)
                    {
                        int value =
p.getBytes().get(i+1).getValues().get(k);
```

```
                        for (int l = 0; l <
p.getBytes().get(i+2).getValues().size(); l++)
                        {
                            int addr =
p.getBytes().get(i+2).getValues().get(l);
                            p.getBytes().get(addr).addValue(value);
                        }

                    }
                }
                else if (abs == AbstractInstruction.WRITEREL)
                {
                    for (int k = 0; k <
p.getBytes().get(i+1).getValues().size(); k++)
                    {
                        int value =
p.getBytes().get(i+1).getValues().get(k);
                        for (int l = 0; l <
p.getBytes().get(i+2).getValues().size(); l++)
                        {
                            int offset =
p.getBytes().get(i+2).getValues().get(l);
                            p.getBytes().get(i+offset).addValue(value);
                        }

                    }
                }
                else if (abs == AbstractInstruction.GOTOCONST)
                {
                }
                else if (abs == AbstractInstruction.GOTOREL)
                {
                }
                else
                {}
            }
        }
    }

    private void printInsBytes()
    {
        for (int i = 0; i < insBytes.size(); i++)
        {
```

```
            System.out.println(insBytes.get(i) + "\n\n");
        }
    }




}

import java.util.ArrayList;
/**
 * An instruction byte has:
 *
 * a base address
 * a list of instructiosn starting at that base address
 * @author Drew Ivarson
 * @version 2/9/2015
 */
public class InstructionByte
{
    // instance variables - replace the example below with your own
    private int address;
    private ArrayList<ConcIns> instructions;
    private ArrayList<Integer> targets;


    /**
     * Constructor for objects of class InstructionByte
     *
     * @param base the base address
     */
    public InstructionByte(int base, ConcIns ins)
    {
        address = base;
        targets = new ArrayList<Integer>();
        instructions = new ArrayList<ConcIns>();
        instructions.add(ins);
    }

    public InstructionByte(int base)
    {
        targets = new ArrayList<Integer>();
        address = base;
```

```
    instructions = new ArrayList<ConcIns>();
}


public boolean isTargetOf(int target)
{
    return (this.address == target);
}

/**
 * Add an instruction to the set of instructions
 *
 * @param  ins the new instruction
 */
public void addInstruction(ConcIns ins)
{
    instructions.add(ins);
}

public void setTargets(ArrayList<Integer> tars)
{
    targets = tars;
}

public ArrayList<Integer> getTargets()
{
    return targets;
}


/**
 * Gets the list of instructions
 *
 * @return the list of instructions
 */
public ArrayList<ConcIns> getInstructions()
{
    return instructions;
}

/**
 * Gets the base address
 *
```

```
     * @return the address
     */
    public int getAddress()
    {
        return address;
    }

    public String toString()
    {
        String toRet = "";
        for (int i = 0; i < instructions.size(); i++)
        {
            if (i > 0)
                {
                    toRet+= " | ";
                }
            toRet += instructions.get(i).toString();
        }

        toRet+= "\nTargets: ";
        String targetAddage = "";
        for (int i = 0; i < targets.size(); i++)
        {
            if (targets.get(i) == -1)
            {
            }
            else
            {
                if (i > 0)
                    targetAddage += ", ";
                targetAddage+= targets.get(i);
            }
        }
        if (targetAddage.equals(""))
            toRet+= "None, end of program.";
        else
            toRet+= targetAddage;

        return "Base Address: " + address + "\n" + toRet;
    }

}
```

```
import java.util.ArrayList;
/**
 * Contains an address and list of possible values
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class CodeByte
{
    private boolean visited;
    private int address;
    private ArrayList<Integer> values;
    private ArrayList<Integer> targets;

    /**
     * Constructor for objects of class CodeBytes
     */
    public CodeByte(int add, int val)
    {
        visited = false;
        address = add;
        values = new ArrayList<Integer>();
        values.add(val);
        targets = new ArrayList<Integer>();
    }

    public void addTarget(int addr)
    {
        targets.add(addr);
    }

    public ArrayList<Integer> getTargets()
    {
        return targets;
    }
    public void markVisited()
    {
        visited = true;
    }

    public void markUnvisited()
    {
        visited = false;
```

```
    }

    public boolean visited()
    {
        return visited;
    }
    public void addValue(int val)
    {
        if (!values.contains(val))
            values.add(val);
        else
            return;

    }
    /**
     * Getter method for a codebyte, gives the list of values
     *
     * @return  values  ArrayList<Integer> of values
     */
    public ArrayList<Integer> getValues()
    {
       return values;
    }

    public boolean notATarget(int addr)
    {
        return !targets.contains(addr);
    }

    public boolean isInstructionBase()
    {
        return !targets.isEmpty();
    }

    public String toString()
    {
        String toRet = "CodeByte @ : " + Integer.toHexString(address) +
"\nValues: ";
        for (int i = 0; i < values.size(); i++)
        {
            toRet += Integer.toHexString(values.get(i)) + ", ";
        }
        if (!targets.isEmpty())
```

```
        {
            toRet += "\nTargets: ";
            for (int i = 0; i < targets.size(); i++)
            {
                toRet += Integer.toHexString(targets.get(i)) + ", ";
            }
        }
        toRet += "\n";
        return toRet;
    }
}

import java.util.ArrayList;
/**
 * A concrete instruction has an opcode, data bytes, and a disassembly
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class ConcIns
{
    // instance variables - replace the example below with your own
    private int opcode;
    private ArrayList<Integer> data;
    private String disassembly;

    /**
     * Constructor for objects of class ConcIns
     */
    public ConcIns(int op, ArrayList<Integer> dataStuff)
    {
        if (dataStuff.isEmpty())
        {
            data = new ArrayList<Integer>();
        }
        else
        {

            data = dataStuff;

        }
        opcode = op;
        disassembly = "";
```

```
}

public ConcIns(int op)
{
    data = new ArrayList<Integer>();
    opcode = op;
    disassembly = "";
}

public ConcIns(int op, int nData)
{
    data = new ArrayList<Integer>();
    data.add(nData);
    opcode = op;
    disassembly = "";
}

public int getOP()
{
    return opcode;
}

public boolean isJMP()
{
    String op = "";
    try
    {
        op = disassembly.substring(0, disassembly.indexOf(" "));
    }
    catch (Exception E)
    {
        return false;
    }
    //System.out.println("isJMP, op: " + op);
    return op.equals("JMP");
}

public boolean isMod()
{
    String op = "";
    try
    {
        op = disassembly.substring(0, disassembly.indexOf(" "));
```

```
        }
        catch (Exception E)
        {
            return false;
        }
        //System.out.println("isMod, op: " + op);
        return op.equals("MOVB");
    }

    public ArrayList<Integer> getData()
    {
        return data;
    }

    public void addDisasm(String dis)
    {
        disassembly = dis;
    }

    public String toString()
    {
        if (opcode == 3)
            return "VIRUS";
        else if (disassembly.equals("") && !data.isEmpty())
            return opcode + " " + data;
        else if (disassembly.equals(""))
            return opcode + " " + data;
        else
        {
            return disassembly;
        }
    }
}

import java.util.ArrayList;
/**
 * A program is a list of codebytes
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class Program
{
```

```
private ArrayList<CodeByte> bytes;
/**
 * Constructor for objects of class Program
 */
public Program(ArrayList<CodeByte> cb)
{
    // initialise instance variables
    bytes = cb;
}

public void markAllVisited()
{
    for (int i = 0; i < bytes.size(); i++)
    {
        bytes.get(i).markVisited();
    }
}
public void markAllUnvisited()
{
    for (int i = 0; i < bytes.size(); i++)
    {
        bytes.get(i).markUnvisited();
    }
}
/**
 * Gets the list of codebytes
 *
 * @return     the list of codebytes
 */
public ArrayList<CodeByte> getBytes()
{
    return bytes;
}


public String toString()
{
    String toRet = "";
    for (int i = 0; i < bytes.size(); i++)
    {
        toRet += bytes.get(i).toString() + "\n";
    }
```

```
        return toRet;
    }
}

import java.nio.file.*;
import java.util.ArrayList;
/**
 * Given the name a file, parses it into an ArrayList of codebytes
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class InputParser
{
    public static Program parseProgram(String name)
    {
        ArrayList<String> fileDump = new ArrayList<String>();
        Path path = FileSystems.getDefault().getPath(name);
        System.out.println("Path: " + path);

        try
        {
              fileDump = (ArrayList<String>)Files.readAllLines(path);
               System.out.println(fileDump);
        }

        catch (Exception E)
        {
               System.out.println(fileDump + "\nException: " + E);
               System.out.println("File reader error");
        }

        int addrCount = 0;
        String row = "";
        ArrayList<CodeByte> program = new ArrayList<CodeByte>();
        for (int i = 0; i < fileDump.size(); i++)
        {
            row = fileDump.get(i);
            row = row.replaceAll(" ", "");
            //System.out.println(row);
```

```
            while (!row.equals(""))
            {
                CodeByte current = new CodeByte(addrCount,
                            Integer.parseInt(row.substring(0, 2), 16));
                program.add(current);
                row = row.substring(2, row.length());
                addrCount++;
            }
        }

        Program p = new Program(program);
        return p;
    }
}


/**
 * Write a description of class main here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
import java.util.ArrayList;
import java.util.Scanner;
public class main
{
    public static void main(String[] args)
    {

            Scanner sc = new Scanner(System.in);
            String name = "blahhhh";
            System.out.println("Welcome to the CFG Maker (someday...)");
            System.out.println("The current instruction set is:\n\n");
            InstructionSet inSet =
InstructionSetParser.parseInstructionSet("InstructionSets/InstrSet0.txt");
            System.out.println("\n\n");
            System.out.print("Enter file name, new to change instruction
sets, or quit to quit: ");
            //System.out.println(inSet);
            name = sc.nextLine();
            System.out.println();
            while (!name.equals("quit"))
            {
```

```
            if (name.equals("new"))
            {
                System.out.print("Enter new instruction set name,\nor
nothing if you don't want to change: ");
                name = sc.nextLine();
                inSet = InstructionSetParser.parseInstructionSet(name);
                System.out.println(inSet);
                System.out.print("Enter file name, new to change
instruction sets, or quit to quit: ");
                name = sc.nextLine();
                System.out.println();
            }
            else
            {
                Program p = InputParser.parseProgram(name);
                  //                  System.out.println(p);

                   //                System.out.println("---Calculating
CodeByte Variants---\n\n");
                  //                  AMBAlgorithm.calcByteVals(p,
inSet);
                  //                ArrayList<InstructionByte> instr =
AMBAlgorithm.recAlgorithm(p, inSet);
                  //              System.out.println("---CodeByte
Variants:\n");
                //            System.out.println(p);
                AMB thing = new AMB(p, inSet);
                thing.analyze();
                //System.out.println(thing.getProgram());


                System.out.print("Enter file name, new to change
instruction sets, or quit to quit: ");
                name = sc.nextLine();
                System.out.println();
            }
        }
    }
}

import java.nio.file.*;
import java.util.*;
/**
```

```
 * Given a file, gives back an InstructionSet object
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */


public class InstructionSetParser
{
    /**
     * Given a file, returns an instruction set
     *
     * @param  filename   the name of the file
     * @return     the instrucion set described by the file
     */
    public static InstructionSet parseInstructionSet(String filename)
    {
       InstructionSet ins = new InstructionSet();
       ArrayList<String> file = new ArrayList<String>();
       Path path = FileSystems.getDefault().getPath(filename);
       try
       {
             file = (ArrayList) Files.readAllLines(path);
       }
       catch (Exception e)
       {
             System.out.println("File reader error");
       }

       for (int i = 0; i < file.size(); i++)
       {
           System.out.println(file.get(i));
           int opcode;
           int numBytes;
           String name;
           AbstractInstruction abs;

           String line = file.get(i);
           opcode = Integer.parseInt(line.substring(0, line.indexOf(" ")),
16);
           line = line.substring(line.indexOf(" ") + 1, line.length());
           numBytes = Integer.parseInt(line.substring(0, line.indexOf("
")));
```

```
            line = line.substring(line.indexOf(" ") + 1, line.length());
            name = line.substring(0, line.indexOf(" "));
            line = line.substring(line.indexOf(" ") + 1, line.length());
            String absStr = line.substring(0, line.length());

            if (absStr.equals("WRITECONST"))
                abs = AbstractInstruction.WRITECONST;
            else if (absStr.equals("WRITEREL"))
                abs = AbstractInstruction.WRITEREL;
            else if (absStr.equals("GOTOCONST"))
                abs = AbstractInstruction.GOTOCONST;
            else if (absStr.equals("GOTOREL"))
                abs = AbstractInstruction.GOTOREL;
            else if (absStr.equals("SKIPORGOTOCONST"))
                abs = AbstractInstruction.SKIPORGOTOCONST;
            else if (absStr.equals("SKIPORGOTOREL"))
                abs = AbstractInstruction.SKIPORGOTOREL;
            else
                abs = AbstractInstruction.SKIP;

            ins.add(new Instruction(opcode, numBytes, name, abs));
        }

        return ins;
    }
}


/**
 * This represents the 6 possible abstract behaviors of any assembly
language
 *
 * WRITECONST - Write constVal to constAddr
 * WRITEREL - Write cosntVal to OFFSET
 * GOTOCONST - Go to constAddr
 * GOTOREL - Go to OFFSET
 * SKIPORGOTOCONST - Branch statement that is either a SKIP or is a
GOTOCONST
 * SKIPORGOTOREL - Branch statement that is either a SKIP or is a GOTOREL
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */
```

```
public enum AbstractInstruction
{
    WRITECONST, GOTOCONST, WRITEREL, GOTOREL, SKIP, SKIPORGOTOCONST,
SKIPORGOTOREL
}


/**
 * An instruction is an assembly instruction.   It has 4 fields:
 *
 * 1.   int opcode - binary representation of its name
 * 2.   int number_of_bytes - the number of bytes long it is including the
opcode
 * 3.   String name - its disassembled name
 * 4.   AbstractInstructrion absSyntax - its representation given the
abstract syntax defined in the file
 * @author Drew Ivarson
 * @version 2/8/2015
 */
public class Instruction
{
    // instance variables - replace the example below with your own
    private int opcode;
    private int number_of_bytes;
    private String name;
    private AbstractInstruction absSyntax;

    /**
     * @param op opcode
     * @param numBytes number of bytes
     * @param name the name
     * @param absIns the abstract instruction
     */
    public Instruction(int op, int numBytes, String name,
AbstractInstruction absIns)
    {
        opcode = op;
        number_of_bytes = numBytes;
        this.name = name;
        absSyntax = absIns;
    }

    /**
```

```
     * Gets the opcode
     *
     * @return  opcode the opcode
     */
    public int getOP()
    {
        return opcode;
    }

    /**
     * Gets the number of bytes
     *
     * @return int number of bytes
     */
    public int getNumBytes()
    {
        return number_of_bytes;
    }

    /**
     * Gets the name
     *
     * @return the name
     */
    public String getName()
    {
        return name;
    }

    /**
     * Gets the abstract syntax
     *
     * @return the AbstractInstruction
     */
    public AbstractInstruction getAbsSyntax()
    {
        return absSyntax;
    }

    /**
     * ToString in the form of:
     * name: NAME, opcode: xx, number of bytes: x, Abstract Syntax:
absSyntax
```

```
     */
    public String toString()
    {
        return "Name: " + name + ", opcode: " + Integer.toHexString(opcode)
            + ", number of bytes: " + number_of_bytes
            + ", AbstractSyntax: " + absSyntax;
    }
}


/**
 * An instruction set is an object that defines the instructions that any
program can have.
 *
 * @author Drew Ivarson
 * @version 2/8/2015
 */

import java.util.ArrayList;

public class InstructionSet
{
    private ArrayList<Instruction> instructions;


    /**
     * Constructor for objects of class InstructionSet
     */
    public InstructionSet()
    {
        instructions = new ArrayList<Instruction>();
    }

    /**
     * Add an instruction
     *
     * @param  Instruction ins the instruction
     */
    public void add(Instruction ins)
    {
        instructions.add(ins);
    }
```

```
public String getInstrNameFromOP(int op)
{
    String toRet = "not found";
    for (int i = 0; i < instructions.size(); i++)
    {
        if (instructions.get(i).getOP() == op)
            toRet = instructions.get(i).getName();
    }

    return toRet;
}

/**
 * Given the opcode, gets the number of bytes
 *
 * @param opcode
 * @return numBytes
 */
public int getNumBytesFromOpcode(int opcode)
{
    if (instructions.isEmpty())
        return 0;
    else
    {
        for (int i = 0; i < instructions.size(); i++)
        {
            if (instructions.get(i).getOP() == opcode)
                return instructions.get(i).getNumBytes();
        }

        return 0;
    }
}

public String disassemble(ConcIns ins)
{
    String toRet = "" + getInstrNameFromOP(ins.getOP());

    for (int i = 0; i < ins.getData().size(); i++)
    {
        toRet+= " 0x"  + Integer.toHexString(ins.getData().get(i));
    }
```

```
    return toRet;
}

/**
 * Given an opcode, gets the name
 *
 * @param opcode the opcode
 * @return name the name
 */
public String getNameFromOpcode(int opcode)
{
    if (instructions.isEmpty())
        return "OPCODE NOT FOUND";
    else
    {
        for (int i = 0; i < instructions.size(); i++)
        {
            if (instructions.get(i).getOP() == opcode)
            {
                return instructions.get(i).getName();
            }
        }
    }

    return "OPCODE NOT FOUND";
}

/**
 * Given the opcode, gets the abstract form
 *
 * @param the opcode
 * @return AbstractInstruction
 */
public AbstractInstruction getAbstractSyntaxFromOP(int opcode)
{
    if (instructions.isEmpty())
        return AbstractInstruction.SKIP;
    else
    {
        for (int i = 0; i < instructions.size(); i++)
        {
```

```
                if (instructions.get(i).getOP() == opcode)
                {
                    return instructions.get(i).getAbsSyntax();
                }
            }
        }

        return AbstractInstruction.SKIP;
    }

    public String toString()
    {
        String toRet = "";
        for (int i = 0; i < instructions.size(); i++)
        {
            toRet += instructions.get(i).toString();
            toRet += "\n";
        }
        return toRet;
    }
}
```