Equational Reasoning on x86 Assembly Code

Kevin Coogan and Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {kpcoogan, debray}@cs.arizona.edu

***Abstract*—Analysis of software is essential to addressing problems of correctness, efficiency, and security. Existing source code analysis tools are very useful for such purposes, but there are many instances where high-level source code is not available for software that needs to be analyzed. A need exists for tools that can analyze assembly code, whether from disassembled binaries or from handwritten sources. This paper describes an equational reasoning system for assembly code for the ubiquitous Intel x86 architecture, focusing on various problems that arise in low-level equational reasoning, such as register-name aliasing, memory indirection, condition-code flags, etc. Our system has successfully been applied to the problem of simplifying execution traces from obfuscated malware executables.**

***Keywords*-equational reasoning; x86 assembly; static and dynamic analysis**

I. Introduction

The ability to analyze software for correctness, efficiency, and security is essential. In the case where the software is developed in-house using high level source code, there are myriad analysis tools available for such purposes (see [1] for an overview). However, there are many situations where high-level source code is not available. For example, large projects often use commercially available libraries that provide only the library binaries. Source code analysis must trust the library documentation regarding its behavior, and will not be aware of any bugs or security holes that may exist in the libraries themselves. Operating systems (e.g., Linux) often use inline assembly code and handwritten assembly code modules to handle architecture-specific functionality. This assembly code typically cannot be handled by source code analysis. Finally, software security companies must constantly analyze the latest viruses, worms, and other malicious code that threaten users on the Internet. Such programs are often written directly in assembly, and those that aren't typically don't provide their source code.

With respect to malware, most instances are targeted at personal and business computers. The most common instruction set architecture for such computers is the Intel x86 architecture due to its close integration with the Windows operating systems. Unfortunately, this architecture presents some unique problems that prevent the straightforward application of source code analysis techniques. The use of different parts of the same user registers to represent different sized values, and the implicit operands and side effects of x86 instructions, require modifications or extensions to current techniques for proper analysis.

Our current research uses dynamic analysis techniques to deobfuscate and understand malware that has been protected from reverse engineering using obfuscation and custom-virtualization techniques. Obfuscation masks the intent of a computation by adding a lot of code whose effects can be tedious and time-consuming to work out. Virtualization hides the control flow behavior of the original program: every byte-code instruction that is executed is reached by a jump from the virtual machine interpreter's dispatch routine, making it difficult to disentangle control transfers arising from the original program logic from those that correspond to a straight-line sequence of byte-code instructions. We address this problem by using equational reasoning to identify the origins of control transfer targets in the virtual machine.

The main contribution of this paper is the presentation of a set of guidelines for building an equational reasoning system capable of handling x86 assembly code. In addition, we implement a prototype tool that can perform equational reasoning on 32-bit x86 assembly code in the context of dynamic analysis. Equational reasoning allows us to handle all implicit functionality of x86 instructions by translating each instruction into a set of equivalent equations. Combining equational reasoning with specific knowledge of x86 user registers allows us to handle the dependencies between instructions that may be missed by typical source code analysis techniques. While virtualized malware was a motivating factor for its creation, the system we present is generic and applicable to any of the above motivating examples. We are unaware of any existing system that can perform such equational reasoning on x86 assembly code.

II. Background and Motivation

There are many applications where assembly code is the most convenient form of a program that is available for analysis. Third party software libraries, architecture-specific operating system modules, and malicious code that must be defended against are all examples of situations where there is a need for analyzing code, but there is not likely to be any high-level source code to analyze. While our own motivation

comes from the area of malware analysis, our system is generic enough to be applied to all of these applications. To that end, we present a motivating example that is outside our own area of specialization, and generic enough to be appreciated by a wider audience. The code in Figure 1 is a small snippet of handwritten assembly code used in the Linux kernel. It comes from the module that performs display adapter and video mode setup for the operating system. This particular sample is doing a check to determine if the video card belongs to a specific manufacturer or not.

```
s3_2:   movw    %cx, %ax    (1)
        orb     %bl, %ah    (2)
        call    outidx      (3)
        call    inidx       (4)
        andb    %cl, %al    (5)
        pushw   %ax         (6)
        movb    %bl, %ah    (7)
        movb    %cl, %al    (8)
        call    outidx      (9)
        popw    %ax         (10)
        cmpb    %ch, %al    (11)
        je      no_s3       (12)
```

Figure 1. Assembly code snippet from linux 2.4 kernel for determining video card manufacturer.

More specifically, the code works by testing whether or not a value can be written to a particular port on a video card. Line (3) attempts to write a value to that port, then line (4) tries to read that value back from the device. It is a known characteristic of "S3" video devices that this port cannot be written to. Hence, if the value is successfully written, then the device is not from "S3." This final check is done by line (11), when the value written to the device, is compared to the value read from the device. A bug in the code could result from the wrong initial value being loaded, or the wrong bits of a register being tested. It would be convenient for validation reasons to be able to say something about the values of `ch` and `al` at the end of this routine, to see if they match with the author's expectations.

One of the problems when trying to analyze x86 assembly code is the register conventions supported by the architecture. The standard user registers can be accessed in whole or in predefined parts, depending on what part of the value is needed. Figure 2 shows the standard layout for the x86 user registers. The effect of this layout is that parts of the register can be accessed and changed with one name, and the changed value can be used later under a different name. For example, the comparison at the end of the code uses the value stored in the `al` register. This value comes from the previous line `popw %ax`. This relationship must be accounted for by any analysis or the dependency between these instructions will be missed. One can imagine that, as the size of the code increases, the ability to account for these relationships by hand gets harder and harder.

There are also issues with implicit functionality in x86

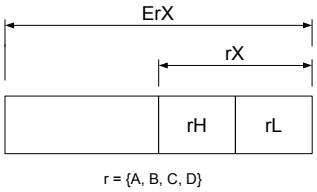

Figure 2. Register layout for x86 general purpose registers.

assembly. The last line of Figure 1 is a conditional jump statement. In x86 assembly, whether or not a conditional jump is taken is typically dependent on the value of the flags register `eflags`. In our example, the value of the flags register is set by the preceding `cmp` operation. In general, the flags register can be set by any number of instructions, including arithmetic instructions like `add` and `or` that also change other register values. These flag changing instructions don't have to immediately precede the conditional jump either. Since the conditional behavior depends only on the value of the `eflags` register, this value can be set at any time, as long as no other instruction alters the value. It could also be set at some program point, saved, then restored at a much later point.

Finally, we note that assembly code is, by nature, a very low level means of expressing a program. As a result, even simple tasks can require many, many assembly language instructions to implement. This fact is one of the primary reasons for programming in higher level languages. Hence, even if the assembly code can be adequately analyzed, there is a need to express the results of the analysis in a way that is more easily read and understood by humans.

These issues, and others associated with x86 assembly, motivate our use of equational reasoning to analyze assembly code. Equational reasoning begins with equations that express the desired functionality. We can combine the functionality expressed by the assembly instructions with our knowledge of the x86 architecture to develop sets of equations for each instruction that fully and completely define its behavior. Furthermore, we can use our knowledge of the x86 architecture and register conventions to precisely define the relationships between an instruction and any previous instructions that contributed to the values of its operands. Once a correct and complete set of equations is derived from the code, it can be simplified to give a single expression that represents the value for a particular variable at a specific point in the code.

We will return to our motivating example in our discussion of results in section V.

III. Implementation

While equational reasoning has been applied in various contexts to evaluation or analysis of computer programs, until now, it has not been applied to the specific problem

of analyzing x86 assembly code. The x86 instruction set presents two primary issues that must be addressed in order for any such system to be useful. First, x86 instructions often have implicit functionality that must be captured, and second, the ability to define parts of registers or memory locations creates dependencies that must be correctly handled. The remainder of this section is broken into two parts. Section III-A gives an overview of our system and the rules for performing the equational reasoning. Section III-B details modifications to the analysis using the concept of liveness that are needed to make computation of the problem practical.

A. System Overview

We wish to be able to generate a set of equations that are equivalent to an x86 assembly code program. With this set of equations, we can choose some point in the program that is of interest to us, and use equational reasoning to derive a simplified expression that represents the value of some variable at that program point. Our system breaks this work down into the following steps:

1) Translate instructions into equations.
2) Instrument list of equations to handle dependencies.
3) Generate simplified expressions for variables of interest.

III.A.1. Notation

x86 assembly instructions are written in the following syntax, where *opcode* denotes the instruction operation and op_i the operands:

opcode $op_1, \ldots, op_n$

Typically, $n \leq 2$, though some instructions have additional implicit source and/or destination operands. For instructions that have both source and destination operands, the first operand is typically both a source and a destination operand: for example, the instruction

```
add  eax, ebx
```

computes '`eax := eax + ebx`'. In general, an operand can be a constant value, a register, or an address expression that specifies a memory location. A special case of the latter is the indirect addressing mode, which is written as '`[`e`]`', where e is an address expression, and denotes the memory location whose address is given by the expression e. For example, an operand '`[ebx+4]`' denotes the memory location whose address is obtained by adding 4 to the contents of register `ebx`. Some instructions may also have optional modifiers, e.g., prefixes, that modify how the instructions are executed. We discuss these modifiers later in the paper.

We begin with a dynamic trace of x86 instructions that includes information about instructions as well as the values of all user registers. Our current implementation uses Ether [2], an academic project that allows great customization and configuration, but which is limited to handling code executing on 32-bit Windows XP SP2. This setup is more than adequate for our needs, due to the large amount of malicious code samples targeted at 32-bit Windows. However, we wish to emphasize that our approach generalizes in a straightforward way to 64-bit assembly code. We simply have not had the need to implement this functionality. Other tracing tools such as Qemu [3] may help with this issue, but have not been studied in depth.

Each instruction in the trace is uniquely identified by its position in the trace, which we refer to as its *order number*. Registers are treated as variables, and are given the same name as their register name. Memory locations are also treated as variables with the generic name `MLOC`: a (contiguous) range of memory locations $\{a_0, \ldots, a_n\}$ is represented as $\texttt{MLOC}[a_0 \,..\, a_n]$. Immediate values are used as is. Operations are substituted with a suitable mnemonic for understanding. The value of a (register or memory) operand *op* immediately after the execution of the instruction with order number k is denoted by op_k. We use the notation $op_\perp$ to represent the value of an operand *op* where the instruction that defined its value is not known yet. We use the notation $op_{\texttt{const}}$ when the value used has no defining instruction in the code (this can happen, e.g., for pre-initialized memory). The value stored at a memory location a is denoted by $\texttt{ValueAt}(a)$.

III.A.2. Instruction Translation

In the first step, we make a pass over the execution trace and process each instruction as follows. We use the semantic specifications for the instruction [4] to create a basic set of equations that capture the effects of that instruction, with operands represented as follows: for an instruction with order number k,

1) a destination operand *dest* is represented in the equation(s) as $dest_k$;
2) a source operand *src* is represented as $src_\perp$ (this subscript will be adjusted in the next step when dependencies are traced and the instruction that defines the value of that operand becomes known or it is found that no such instruction exists).

Figure 3(a) gives two typical x86 instructions with their order numbers in a snippet of a dynamic trace, and Figure 3(b) shows the result of instruction translation on those instructions. The code was chosen for demonstration only, and its function is irrelevant to our discussion. The "`pop`" operation, according to the Intel documentation, first moves the "popped" value from the memory location pointed at by the stack pointer `esp` to the destination location. Then, the value of the stack pointer is increased according to the size of the value moved. The `and` operation is translated according to its definition.

In addition, we must also account for other implicit functionality or side effects of the x86 instructions. For example, some x86 instructions also implicitly set or change the value of the `eflags` register. We introduce the `Flag`

```
100: pop eax
101: and al, 0x4
```

(a)

$\texttt{eax}_{100}$ = ValueAt(MLOC$[1000\,..\,1003]_{\perp}$)
$\texttt{esp}_{100}$ = $\texttt{esp}_{\perp}$ + 4
$\texttt{al}_{101}$ = $\texttt{al}_{\perp}$ & 4

(b)

Figure 3. Simple instruction translation example for a snippet of dynamic trace. (a) gives the original x86 assembly code, and (b) gives the equations derived from the instruction definitions.

operation, which takes an expression and returns the value of the eflags register that results from evaluating that expression. We then add a new equation to our set of the form eflags = Flag(expression) to capture this implicit behavior.

Finally, if the instruction being translated accesses a memory location, then additional equations are added that represent how the memory address location was calculated. These memory-location equations allow our analysis to capture and correctly handle indirection. This is a significant obstacle in analyzing x86 assembly code since indirect memory accesses are legal in nearly all x86 instructions. Figure 4 shows the result of these additional steps on our example code from Figure 3(a).

(1) MLOC$[1000\,..\,1003]_{100}$ = $\texttt{esp}_{\perp}$
(2) $\texttt{eax}_{100}$ = ValueAt(MLOC$[1000\,..\,1003]_{\perp}$)
(3) $\texttt{esp}_{100}$ = $\texttt{esp}_{\perp}$ + 4
(4) $\texttt{al}_{101}$ = $\texttt{al}_{\perp}$ & 4
(5) $\texttt{eflags}_{101}$ = Flag($\texttt{al}_{\perp}$ & 4)

Figure 4. Final translated equations as a result of step 1.

III.A.3. **Handling Dependencies**

The purpose of this step is to associate all of the source operands in the equations from step 1 with their definitions. We do this by scanning the equation list backwards, looking for where the source is defined. If the use of a variable was always preceded by an exact definition, then this step would be trivial. However, in x86 assembly, this is not always true. Consider equations (2) and (4) in Figure 4. Equation (2) sets the value of the eax register. Equation (4) uses the value of the al register. Without more information, it appears that there is no relation between these two instructions. However, this is not the case. The al register is simply the least significant 8 bits of the eax register. Thus, equation (2) is determining the value of al used in equation (4). There are several such relationships for many of the accessible registers in the x86 architecture, as was shown in Figure 2. Furthermore, the x86 architecture is what is known as "byte addressable." This means that any byte of accessible memory can be written to or read from by an instruction. The x86 architecture also allows for instructions that read more than one byte at a time. These rules and relationships may make naive attempts at equational reasoning incorrect.

We note that the definition of source operands falls into one of five cases. In case I, the destination of the equation that defines the source operand is an exact match for the source operand. That is, if the source operand is eax, our backward search will first find an equation that sets the value of eax. In case II, the first equation we find in our search completely defines the source operand through a different name or reference. For example, if our source operand is ah and our defining equation destination is eax. To put it another way, the source operand is a proper *subset* of the defining destination operand. In case III, the opposite is true. The source operand is a proper *superset* of the defining destination operand. This means that the full definition of the source operand comes from multiple previous instructions. Case IV occurs when there is an overlap between source operand and its definition, but neither is a subset of the other. As we will see, this case is not possible when dealing with user registers, but may occur for memory locations. Case V is where no definition is found.

In case I, the source operand is defined by a previous equation, and requires only that we note where the definition takes place. Consider the second equation in Figure 4. The source operand MLOC$[1000\,..\,1003]_{\perp}$ is passed to the ValueAt function. If we scan backwards in our list of equations, we see that this source is defined in the previous equation. We note the unique identifier of the defining equation, and use it to label the source operand.

In case II, the source operand is completely defined by one previous equation, but using a different name or reference. Consider the fourth equation in Figure 4 that changes the value of the al register. The source is fully defined by equation (2) in Figure 4. However, we are only using part of that definition. When we simplify instructions later in the process, we will want to replace this operand with its definition. However, we cannot substitute $\texttt{eax}_{100}$ because this is a four byte value, and the operand in equation (4) is a one byte value. For correctness, we must specify which part of the value is being used. We accomplish this by introducing a new operation called Restrict. The Restrict operation takes a variable and a byte mask representing which part of the variable is to be used, and returns that portion of the value stored in the variable.

In the example of equation (4), we begin by scanning backwards through the list of equations looking for the definition of al. When we reach equation (2), we recognize that al is a subset of the definition destination. We create a new equation (marked with $*$ in Figure 5) after equation (2) that precisely defines the variable we are searching for in terms of how it was defined in equation (2) and using the Restrict function to select the part of the value that we need. We use the identifier from the defining equation to label the destination of our new equation. Figure 5 shows

the example equations modified to handle case I and case II source definitions. The byte mask passed to `Restrict` denotes that the least significant byte (least significant 8 bits) of `eax` is used. Note that the `al` source in equation (5) changed in the same way as in equation (4). Since the same instruction calculates both `al` and `eflags`, they should appear the same in the set of equations.

It may be tempting to forgo the Restrict operation entirely for a more intuitive approach of handling all bytes of the CPU registers separately. However, careful examination of this approach reveals limitations. First, this approach will significantly increase the total number of equations for the dynamic trace. Second, later uses of a register will require concatenation of these partial values. When the value is unknown, this approach greatly increases the complexity of the resulting expression. Our experience tells us that the two main contributing factors to run time are number of equations and the complexity of the individual expressions. Hence, we favor our approach.

(1) $\texttt{MLOC}[1000\,..\,1003]_{100}$ = $\texttt{esp}_{\perp}$
(2) $\texttt{eax}_{100}$ = ValueAt(MLOC$[1000\,..\,1003]_{100}$)
(*) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
(3) $\texttt{esp}_{100}$ = $\texttt{esp}_{\perp}$ + 4
(4) $\texttt{al}_{101}$ = $\texttt{al}_{100}$ & 4
(5) $\texttt{eflags}_{101}$ = Flag($\texttt{al}_{100}$ & 4)

Figure 5. Case I and case II source definitions handled correctly.

Case III occurs when multiple previous equations contribute to the full definition of a source operand. We handle this situation by adding equations to capture the first partial definition of the source operand, then continuing the search for the remainder of the definition. We have not presented an example of case III in our examples so far because it is a little more complicated, so consider the new example equations in Figure 6. For our discussion, we are searching for the definition of the source operands in equation (4).

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
...
(4) $\texttt{ebx}_{140}$ = $\texttt{eax}_{\perp}$ + 4

Figure 6. Example of case III type source dependencies.

As we search backward for the definition of $\texttt{eax}_{\perp}$ in our equations, we find it partially defined by equation (3). We cannot ignore this partial definition, of course. However, it is also not a complete definition of the register. To handle this case, we add a new equation that precisely defines the source operand we are looking for in terms of the partial definition and the still unknown parts. For example, the value of `eax` at this point is the previous value of `eax`, combined with the changed value of `al`. We use simple bit-wise operators to accomplish this combination. Figure 7 shows this equation marked with a $*$. Note that the `eax` operand in this equation now needs a suitable definition. We recursively call our search method on this new operand until it has been fully defined. In this example, that happens when we reach equation (1), so we assign it identifier 100.

(1) $\texttt{eax}_{100}$ = 1000
(2) $\texttt{al}_{100}$ = Restrict($\texttt{eax}_{100}$, 0001)
...
(3) $\texttt{al}_{120}$ = $\texttt{al}_{100}$ + 4
(*) $\texttt{eax}_{120}$ = ($\texttt{eax}_{100}$ & 1110) | $\texttt{al}_{120}$
...
(4) $\texttt{ebx}_{140}$ = $\texttt{eax}_{120}$ + 4

Figure 7. Case III dependencies from Figure 6 handled correctly.

Case IV occurs when the source operand is partially defined by a previous destination operand, and these two operands overlap but neither is a subset of the other. For the user registers, this case is impossible. If we look again at Figure 2, we see that in no case does a legal register name overlap another name in this way. However, in the case of memory operations, this can occur. To handle this case, we combine our solutions from Cases II and III. First, we `Restrict` the destination operand so that the destination is a subset of the source. Then we recursively call our function to look for the remainder of the definition. Due to space constraints we do not offer any Case IV examples.

Case V is a trivial case where no definition of a value is found. This situation can arise for memory accesses when the memory location holds a pre-initialized constant that does not change throughout the execution of the program. It has also been seen in obfuscated code, where the code uses uninitialized register values in junk code that do not affect the results of the execution to try to make program analysis more difficult. In all instances of Case V, we simply label these values as constants. That is, a source operand *src* with no explicit definition will be labeled $src_{\texttt{const}}$.

III.A.4. Expression Simplification

We note that at this stage, all source operands have a unique and precise definition previously in the equation list. Equational reasoning then becomes a straightforward operation. We begin by identifying what values are of interest. This could be the value of a register used in an instruction, or the parameter of a function call that has been pushed onto the stack, etc.... For each variable of interest, we create a trivial equation that assigns its value to itself at a point in the program where we want to analyze it (e.g., $\texttt{eax}_{103}$ = $\texttt{eax}_{\perp}$). We then calculate the source operand definitions for the equation, as before.

Now, we substitute the right hand side of the equation with the expression that defines it previously in the list of equations, so that the right hand side of the equation is the root of a syntax tree structure that represents the expression. We can simplify our syntax tree using a set of mathematical

rules, as desired. We continue this process, substituting operands with their defining expressions and simplifying, until no more changes can be made. At this point, we have a simplified expression that represents the variable of interest. We note that there are existing equation solving tools that could be made to work with our resulting equations, but we implement this simplification step ourselves.

B. Run Time Considerations

While the solutions presented to cases II and III (and by extension Case IV) above are mathematically sound, they present other problems during the equational reasoning portion of our system. Mainly, it is possible for these approaches to lead to an exponential increase in the size of the equations. Consider the example equations given in Figure 8. This example is similar to the one presented in Figure 6, except there are multiple changes to the value of `al`. Here, the ellipses indicate other instructions that are irrelevant to our analysis, except that they prevent the simplification of equations (2), (3), and (4) to avoid this problem. If we apply our approach to handling source definitions, we get the equations in Figure 9, because at each step we are adding a new equation with an `eax` term, then looking for the definition of that term.

```
(1) eax100 = 1000
    ...
(2) al120 = al⊥ + 4
    ...
(3) al140 = al⊥ + 4
    ...
(4) al160 = al⊥ + 4
    ...
(5) ebx180 = eax⊥ + 4
```

Figure 8. Example leading to exponential explosion of equations.

However, many of these equations are redundant. Examining the original equations shows that we don't care about the full value of `eax` anywhere except at the beginning and the end. If we try to simplify equation (9) by substituting the term $\texttt{eax}_{160}$ with its definition, then repeat this process, we must handle many `eax` terms that are not contributing to the value of the equation. In our example, this is not too important, because the original value of `eax` is constant and can be simplified, but if this value were unknown, then our simplified equation would become longer and more complex. In severe cases, this added complexity can adversely effect run time to the point of being impractical.

To handle this problem, we use the concept of liveness to modify the search algorithm that we use to find the source definitions. Simply put, a variable in a program is "live" if its value may be used later. We can use this idea to deal with unnecessary instances of a variable by tracking which parts of the variable are live. In our example, we saw that we had to create new equations that used `eax` each time

```
(1) eax100 = 1000
(2) al100 = Restrict(eax100, 0001)
    ...
(3) al120 = al100 + 4
(4) eax120 = (eax100 & 1110) | al120
    ...
(5) al140 = al120 + 4
(6) eax140 = (eax120 & 1110) | al140
    ...
(7) al160 = al140 + 4
(8) eax160 = (eax140 & 1110) | al160
    ...
(9) ebx180 = eax160 + 4
```

Figure 9. Exponential explosion of equations resulting from equations in Figure 8.

we encountered a partial definition. This happened several times since each new equation had its own `eax` source operand. However, the identification of some of these partial definitions was incorrect, since our operand used the three leftmost bytes of `eax`, and the definitions defined only the rightmost byte. In truth, these equations do not define the relevant part of the source operand at all. To capture this idea, we assign a liveness value to all operands when we search for their definitions. In the general case, the entire variable is live, but in the case of our added equations, we use our knowledge of what part of the variable is used to assign a more precise value.

```
(1) eax100 = 1000
(2) al100 = Restrict(eax100, 0001)
    ...
(3) al120 = al100 + 4
    ...
(4) al140 = al120 + 4
    ...
(5) al160 = al140 + 4
(6) eax160 = (eax100 & 1110) | al160
    ...
(7) ebx180 = eax160 + 4
```

Figure 10. More precise results that avoid problem of exponential explosion.

Applying this approach to our original example in Figure 8, when we search for the definition of `eax` in equation (5), we find the partial definition in equation (4). We add our extra equation as before, but when we recursively search for the definition, we tell the search algorithm that only the leftmost three bytes of `eax` are live. Thus, continuing to search backwards, we see equations (2) and (3), and recognize that they do not affect the value of `eax` at all because they do not define any of the live parts of the variable. Only when we reach equation (1) do we find a definition of our variable. The final set of equations given in Figure 10 produces better, more simplified, results.

We observe that our analysis uses a byte mask to mark which parts of a variable are live. Registers are of fixed size, and so a finite sized byte mask makes sense. In x86,

most memory accesses are also of fixed size. However, in the case of string operations that carry a `rep`, `repz`, or `repnz` prefix, this is not the case. These prefixes can cause a string instruction to effectively access any sized memory block, by repeating the string operation. To allow for our fixed size byte masks, we handle each repeat of these string operations as separate instructions. Hence, all memory accesses can access a maximum of 4 bytes for a 32-bit program.

IV. Results

We begin with a small toy example to demonstrate our tool. We wrote a sample program that calculates the factorial of an input value, then prints the value to the screen. We captured a dynamic trace of the execution of our program with the input value 4, and used that trace as input to our system. Using our equational reasoning system to analyze the arguments passed to the print routine, the value that is printed out is determined to be the expression:

$$\texttt{ValueAt}(\texttt{MLOC}[\texttt{12ff78}\,..\,\texttt{12ff7b}]_{312}) = ((((1*(1))*(1+1))*(1+2))*(1+3))$$

This indicates that the value passed to the print routine was the result of multiplying $1 \times 1 \times 2 \times 3 \times 4$, or $4!$, which is correct. The result also indicates that there is a redundancy in our implementation. It appears that our code is looping 4 times, and is calculating the initial value times 1. We checked our source code and confirmed that this is the case.

Next, to illustrate the use of our tool on more practical, real world situations, we present a couple of examples from our current area of research. Our ongoing research project is to deobfuscate malicious code to make it easier for security analysts to understand. One such obfuscation that we have concentrated on is a technique called virtualization, where instructions from a program are translated into virtual instructions, and a custom virtual machine is built into the code to interpret these instructions. If we analyze the control flow of the resulting program, we see that execution of the virtualized code is really just some dispatch routine that is called over and over again. This dispatch routine serves a simple function–to fetch the address of the next instruction to execute, then to jump to that instruction.

Despite being a very simple concept, the implementation of the dispatch routine is often very complex and confusing. Consider the example in Figure 11. This example was generated by first compiling a sample program written in the C programming language. This executable was then virtualized using the commercially available software CodeVirtualizer [5]. (We conjecture that the code inserted by CodeVirtualizer for its virtual machine is handwritten assembly code.) The example begins with the first instruction from the dispatch routine at order number 297. This instruction loads a byte from memory into the `al` register. The instructions at order numbers 305 through 312 do some calculation, then combine this calculation with the value in the `al` register. The instructions at order numbers 369 and 379 move this value onto the stack, then retrieve it later. Finally, the example ends with the actual jump to the virtual instruction. Notice that the jump occurs at order number 381, indicating that the dispatch routine consists of 85 instructions. As an analyst, we may be interested to know *how* the address of the virtual instruction is calculated. This may give us insight into what all of those instructions are doing, and why they are needed.

```
297: lodsb
...
305: mov bh, 0xa4
306: xor bh, 0x25
307: or bh, 0x7d
308: sub bh, 0x72
309: shr bh, 0x7
...
312: add al, bh
...
369: mov [esp], eax
...
379: pop eax
...
381: jmp dword near [edi+eax*4]
```

Figure 11. Sample code from CodeVirtualizer dispatch routine.

We can use our equational reasoning system to generate an expression for `eax` and `edi` at the point of the jump to the virtual instruction. First, we present an intermediate result in Figure 12. For the purpose of discussion, we have performed our analysis for the `eax` register, and suppressed any expression simplification except for simple substitution. The resulting expression is too long to reproduce completely, but we have included those parts relevant to the code in Figure 11. We see that the value of `eax` starts with the value from memory location `0x4090f4`. This is the memory location implicitly accessed by the instruction at order number 297. We also see that this value is a constant, indicating that the value in this memory location has not changed since the program began execution. Next, we see that the arithmetic operations of instructions 305 through 309 are captured by the expression. These operations are mainly combining immediate values, which can be simplified away. We also see that the moves of the value to and from the stack have disappeared through simple substitution.

$$\begin{aligned}\texttt{eax} = \texttt{ValueAt}(&\texttt{MLOC}[\texttt{4090f4}\,..\,\texttt{4090f4}]_{\texttt{const}})\\ &- \ldots\\ &+ (((\texttt{0xa4} \;\hat{}\; \texttt{0x25}) \mid \texttt{0x7d}) - \texttt{0x72})\\ &\quad \gg \texttt{0x07}\\ &+ \ldots\end{aligned}$$

Figure 12. Unsimplified partial expression for `eax` register (arithmetic operators are as in C or Java).

Figure 13 shows the results of our full analysis with simplification on the registers `eax` and `edi` at the point

of the dispatch jump. Here we see that the manipulation of the value read from memory is much simpler than appeared at first. This appears to be a simple decryption where we add some immediate value, then xor the result with another immediate value. The value of eax then is just a value read from memory, and decrypted using a hard-coded decryption key. We also see that the value for edi is a constant value. By looking at some other instances of the dispatch routine, we see that edi holds the address of a table, and eax indexes into that table to retrieve the instruction addresses.

$$\begin{aligned} \texttt{eax}_{381} &= (\texttt{ValueAt}(\texttt{MLOC}[\texttt{4090f4}..\texttt{4090f4}]_{\texttt{const}}) \\ &\quad + \texttt{0x75}) \ \hat{} \ \texttt{0x6c} \\ \texttt{edi}_{381} &= \texttt{0x00404200} \end{aligned}$$

Figure 13. Simplified expressions for eax and edi registers.

Next, we move on to a slightly more complicated example. The code in Figure 14 shows the implementation of conditional control flow logic in another virtualization product known as VMProtect [6]. VMProtect makes deobfuscation difficult in several ways. Here, it has implemented a conditional branch statement without the use of the standard x86 branch instructions. Instead, instructions 1438-1440 and 1450-1452 load the addresses of the if and else instructions, respectively, onto adjacent locations on the stack. The address of the smaller memory address is also saved to memory (not shown). Next, some statement is executed that calculates the value for the eflags register. That value is masked so that all that remains is the flag that is needed (not shown) leaving the result in eax of instruction 2017. This flag value is bit-shifted to the right, so that if the flag value was true, the resulting value in eax is 4. Otherwise, the resulting value in eax is 0. Then the value in eax is added to the smaller address that was saved earlier. The effect is that if the flag was false, the smaller address (i.e., the stack address of the if clause) will be used in instruction 2057 to load a value into eax. If the flag was true, then the address of the else clause will be used. This stack address is then used in instruction 2310 to load the table index of the next instruction to execute.

One of the tasks that we need to perform when deobfuscating such code is to identify the conditional control flow logic of the original program. Here, that logic has been hidden, and simple analysis to see how the target address of the jump was calculated only reveals the use of a constant address value loaded at instruction 2314. Figure 15 shows results we obtained from our equational reasoning on the sample code from Figure 14. We see that the value of eax used to index into the table in instruction 2317 is simply a constant value. If we follow the indirection in instruction 2314 to see how the value of esi is calculated, we see that it is also a constant value. However, if we follow the indirection in instruction 2057 to see how eax was calculated we get a very complex expression that is

```
1438: mov eax, [esi]
1439: sub ebp, 0x4
1440: mov [ebp+0x0], eax
...
1450: mov eax, [esi]
1451: sub ebp, 0x4
1452: mov [ebp+0x0], eax
...
2017: shr eax, cl
...
2039: add [ebp+0x4], eax
...
2056: mov eax, [ebp+0x0]
2057: mov eax, [ss:eax]
...
2058: mov [ebp+0x0], eax
2310: mov esi, [ebp+0x0]
...
2314: mov al, [esi]
2315: inc esi
2316: movzx eax, al
2317: jmp dword near [eax*4+0x405091]
```

Figure 14. Sample code from VMProtect implementation of conditional control flow.

conditionally dependent on the return value of a call to the "atoi" function, which is then bit-shifted to the right and added to a stack address. The complete expression is too complicated to reproduce here, however, we can confirm that this is exactly the calculation we described above.

$$\begin{aligned} \texttt{eax}_{2056} &= \texttt{0x0012ff74} + \\ &\quad ...(\ \texttt{Flag}(...\texttt{atoi}(\texttt{0x003548e1})...) \\ &\quad \gg \texttt{0x04}) \\ \texttt{esi}_{2314} &= \texttt{0x00405765} \\ \texttt{eax}_{2317} &= \texttt{0x0000002d} \end{aligned}$$

Figure 15. Conditional control flow instructions from VMProtect example.

This analysis proves that the calculation of the address of the next instruction at this point in the code is conditionally dependent. How and why to follow the indirections in the code is not discussed here, and is the subject of our ongoing research into the deobfuscation of malicious code. However, we do see from this example how difficult the task would be without some form of simplification such as that given to us by equational reasoning. In this example, the code spans 880 instructions in the dynamic trace. At each instance of indirection, we can use the simplified expression to determine if there is a dependency that we care about.

Our system is not limited to simply analyzing hand-picked locations in the dynamic trace. Since each location typically depends on one or more previous locations in the trace, and those previous locations depend on previous locations, we found it easy to implement a general solution that performs equational reasoning to calculate a simplified expression for every single destination operand in every generated equation from the trace. Thus, examining a particular location is done

by simply looking the expression up in the list of results.

To give an example of the run time of this "all destinations" approach, we executed the code on a sample dynamic trace of 106407 instructions, which generated a total of 261977 equations. Full expression simplification for all destination operands ran in 489.0 seconds, on average. This time corresponds to approximately 218 instructions per second or 536 equations per second. Tests were run on an Intel Core 2 Duo E6600 2.4GHz CPU with 4GB RAM.

V. Discussion and Future Work

A. Applications to Static Analysis

We have implemented our dynamic system because of an immediate need for such a system in our ongoing research work. However, the techniques we used are fairly general and should be applicable in a static context as well. In the static analysis case, we assume a complete and correct static disassembly of the program such that a control flow graph of the program can be built. This assumption may not always hold in general, but the problem of static disassembly of assembly code is outside the scope of our work. For example, our current research focuses on malware. However, many instances of malware incorporate self-modifying code into their protections, which render standard static analysis ineffective.

An important requirement of our system is the ability to have distinct names for the values of a variable at different points in the program. In the dynamic analysis case, we did this using the order number of each instruction. For the static analysis case, we can obtain a similar effect by converting the code to static single assignment (SSA) form [7]. SSA form creates a unique instance of a variable for each definition, so that in the code, the location of the definition of every use of a variable is known. This is typically accomplished in SSA by assigning a variable name for each definition that consists of the variable name from the original code concatenated with a unique number. So, just as we used the order number of the dynamic trace to allow for unique equations in the dynamic implementation, we use SSA form, and its unique variable names to guarantee that our set of equations is referentially transparent.

A second requirement is the need to determine the possible referents of indirect memory accesses. In the dynamic case discussed here, the runtime register values associated with each instruction in the trace allow us to identify the specific memory locations accessed by any instruction. In the static analysis case, the corresponding information has to be obtained using pointer analysis [8].

We return to our motivating example from section II. We argued that it may be helpful to know something about the values that were being compared at the end of the routine in Figure 1. If we apply SSA form to the code in our example, and perform our analysis by hand, we get the following results. At the point of conditional jump, `ch` is a constant value. That is, it has not been changed by the code we are analyzing. For `al` at the point of the conditional branch, we get the expression

$$\texttt{Restrict(inidx(}\mathtt{dx}_{\mathtt{const}}\texttt{), 01) \& }\mathtt{cl}_{\mathtt{const}}.$$

In short, it is the lower 8 bits of the value read from the device (i.e., the `inidx` function call) `and`'ed with the value of the `cl` register at the begininng of the code. In other words, if the least significant 8 bytes we read from the device are equal to the value we started with in the `cl` register, it is not an "S3" device. This matches very closely with the apparent intent of the code provided by the programmer in the code comments.

B. Other Considerations

The discussion of expression simplification in Section III.A.4 effectively describes a term-rewriting system. It is therefore reasonable to ask whether the result of simplifying an expression using our equations is guaranteed to be unique—in other words, whether the rewrite system is Church-Rosser, or *confluent*. The focus of the work described in this paper is on low-level issues specific to reasoning about assembly code, such as register-name aliasing, indirect memory accesses, condition-code flags, etc.; issues of confluence are somewhat orthogonal to this focus and so are not addressed here. In principle, one could apply Knuth-Bendix completion [9] to the rewrite rules we use in order to obtain a confluent rewriting system. This issue has been discussed by Walenstein *et al.* [10].

VI. Related Work

Equational reasoning has been applied to many problems in software analysis, such as certifying properties of a functional program [11], linking first class primitive modules [12], and rewriting Haskell fragments [13].

The analysis of assembly code is well studied. Walenstein, *et al.* used term rewriting for normalizing metamorphic variants of viruses and other malicious code [10]. This work deals specifically with identifying semantics-preserving x86 code transformations performed by the malware and does not address analysis of the code in general. Leroy employed equational reasoning to verify PowerPC assembly generated by CompCert compiler [14]. Similar RTL-style representation has been used [15] to improve disassembly by identifying indirect jumps to function calls. There are other examples as well [16]–[19]. However, we are not aware of any such work using equational reasoning techniques.

Our work is similar in some aspects to the formal verification work of Magnus Myreen [20], though his work is concerned with verification of code, and ours is geared towards understanding behavior. Our approach is also similar, in principle, to symbolic execution techniques [21], [22]. However, symbolic execution typically does not address low level issues such as x86 register-level aliasing.

VII. Conclusions

Library binaries, handwritten assembly for architecture-specific needs, and malicious code are just a few examples of cases where there is a need for program understanding, but where no high-level source is available for analysis. We present an equational reasoning system specifically designed to handle the unique issues associated with x86 assembly – a common instruction set architecture – and can be used in static or dynamic contexts. Our system can create an expression to represent any variable at any point in the code, and can simplify this equation for the purpose of making it more readable by humans.

We have implemented an automated version of our dynamic tool for analyzing 32-bit x86 traces, and have applied this tool to several dynamic examples. We have also applied the rules of our system by hand to a static code sample with known values to demonstrate the possibility of extending the system to static analyses. The current tool is capable of processing approximately 218 instructions per second, or 536 equations per second on our test setup. Our system does not solve all analysis problems, but is one more tool in the toolbox of researchers and engineers whose job is to analyze and understand assembly code and executable binaries.

Acknowledgement

This work was supported in part by the National Science Foundation via grant no. CNS-1016058, as well as by a GAANN fellowship from the Department of Education award no. P200A070545.

References

[1] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*, ser. FOSE '07, 2007, pp. 104–119.

[2] A. Dinaburg, P. Royal, M. I. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, 2008, pp. 51–62.

[3] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 2005, pp. 41–46.

[4] *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, Intel Corp.

[5] Oreans Technologies, "Code virtualizer: Total obfuscation against reverse engineering," Dec. 2008, `http://www.oreans.com/codevirtualizer.php`.

[6] VMProtect Software, "Vmprotect software protection," Dec. 2008, `http://vmpsoft.com/`.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[8] G. Balakrishnan, "Wysinwyx: What you see is not what you execute," Ph.D. dissertation, Computer Science Department, University of Wisconsin, Madison, 2007.

[9] D. E. Knuth and P. Bendix, "Simple word problems in universal algebras," in *Computational Problems in Abstract Algebra*, J. Leech, Ed. Pergamon Press, 1970, pp. 263–297.

[10] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia, "Normalizing metamorphic malware using term rewriting," in *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, 2006, pp. 75–84.

[11] J. S. Jorge, V. M. Gulias, and J. L. Freire, "Certifying properties of an efficient functional program for computing Gröbner bases," *J. Symb. Comput.*, vol. 44, pp. 571–582, May 2009.

[12] J. B. Wells and R. Vestergaard, "Equational reasoning for linking with first-class primitive modules," in *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ser. ESOP '00, 2000, pp. 412–428.

[13] A. Gill, "Introducing the haskell equational reasoning assistant," in *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, ser. Haskell '06, 2006, pp. 108–109.

[14] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, pp. 107–115, July 2009.

[15] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proceedings of the 20th international conference on Computer Aided Verification*, ser. CAV '08, 2008, pp. 423–427.

[16] L. Djoudi and L. Kloul, "Assembly code analysis using stochastic process algebra," in *Proceedings of the 5th European Performance Engineering Workshop on Computer Performance Engineering*, ser. EPEW '08, 2008, pp. 95–109.

[17] J. Brauer, B. Schlich, T. Reinbacher, and S. Kowalewski, "Stack bounds analysis for microcontroller assembly code," in *Proceedings of the 4th Workshop on Embedded Systems Security*, ser. WESS '09, 2009, pp. 5:1–5:9.

[18] R. Venkitaraman and G. Gupta, "Static program analysis of embedded executable assembly code," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES '04, 2004, pp. 157–166.

[19] A. Wolfram, P. Braun, F. Thomasset, and E. Zehendner, "Data dependence analysis of assembly code," *Int. J. Parallel Program.*, vol. 28, pp. 431–467, October 2000.

[20] M. Myreen, *Formal Verification of Machine-code Programs*, ser. Distinguished Dissertation Series. British Computer Society, The, 2011. [Online]. Available: http://books.google.com/books?id=jyDvtgAACAAJ

[21] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[22] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé, "Using symbolic execution for verifying safety-critical systems," in *Proceedings of the 8th European software engineering*, ser. ESEC/FSE-9, 2001, pp. 142–151.