Trudio: Dynamic Analysis Approach for Obfuscated Programs

***Abstract*—When a new malware is found, security researchers should first understand the contents of the malware to prepare countermeasure. However, this is a difficult task since the malware writers distort the contents of malwares using code obfuscation techniques. Nowadays, it is getting more dfficult to understand a malware since the obfuscation techniques are getting more sophisticated and diverse. Thus, it is infeasible to develop reversing techniques for every obfuscation technique. Nonetheless, since an obfuscated program inevitably has the same semantics with the original program, monitoring the dynamic behaviors of the program will help the analyzers to figure out the inside. In other word, to disclose the semantics of an obfuscated program, it will be useful to analyze its run trace, which is the record of executed instructions and changes in status. We presents a program analysis tool Trudio, which analyzes a program based on its run traces and transforms the program into more analyzable shape using the information acquired from the run traces. With our tool the malware analyzers can save time to understand an obfuscated malware. This paper presents the ideas and algorithms that are implemented in Trudio and proves that these methods can help analysis by experiments.**

***Keywords*-security, reverse engineering, program analysis**

I. Introduction

When a new malware is introduced, security researchers should first understand the contents of the malware to prepare countermeasure. This is a difficult task since the malware writers distort the contents of malwares to mislead the analyzers.

The methods to distort the contents of a program are called code obfuscation. Traditional code obfuscation includes dead code insertion, control flow distortion, instruction substitution, and data encoding. These methods increase the analysis time of the target program, but since they typically preserve the overall structure of the program, experienced analyzers can understand the program in short time.

However, the obfuscation techniques applied to the malwares become more sophisticated and diverse. Nowadays, the malware writers are even using the techniques called virtualization obfuscation. A program obfuscated with the virtualization obfuscation contains a virtual machine running a random instruction set, and the original program code encoded into byte codes of the instruction set of the embedded virtual machine. The obfuscated program will run the virtual machine with the byte codes, so the obfuscated program is semantically equivalent to the original program. However, the structure of an obfuscated program will be completely different from the original code because the obfuscated program runs the virtual machine.

Several researches have tried to reverse obfuscation techniques [1], [2]. The reversing methods developed for a specific obfuscation have strict assumptions on the obfuscation technique. Those methods are generally effective for the obfuscated programs fitted into the assumptions of the reversing method. However, those reversing algorithms cannot be applied to a revised obfuscation technique that does not meet the assumptions. For instance, after Rolles introduced a reversing method for virtualization obfuscation [1], obfuscator VMProtect [3] was updated to avoid the reversing method.

Moreover, the number of malwares is exponentially increasing. In the first half of the year 2011, more than one million malwares running on Windows are newly identified in the market [4]. Thus, it is infeasible to develop reversing techniques for every obfuscation technique applied in each malware.

Even if a program is heavily obfuscated, the program inevitably has the same semantics with the original program. From this fact, we noticed that monitoring the dynamic behaviors of an obfuscated program can help to figure out the inside of the program.

Inspired by this observation, we designed and implemented Trudio, which is a program analysis tool based on dynamic approach. Trudio has three purposes: structure analysis, semantic analysis, and optimization. The structure analysis reveals how the program is organized. Through this analysis, analyzers can find the actual control flow of the program and figure out which obfuscation techniques are applied to the program. The semantic analysis aims to show the semantics of the program. Analyzers can see the algorithms in the target program via this analysis. The optimization transforms the target program into more analyzable shape by removing unnecessary instructions.

Trudio does not aim to automatically reverse a specific obfuscation technique, but it is rather a general purpose program analysis tool. However, Trudio will save the analysis time of the malware analyzers since they expect to understand the malwares not to have the source code or unobfuscated version of the malwares.

This paper presents the ideas and algorithms embodied in Trudio. However, the methods presented in this paper can be considered as general approaches to understand of a program based on dynamic analysis. This paper also presents the experimental proof that Trudio appropriately reveals the structure and the semantics of obfuscated programs and optimization is effective.

This paper is organized as follows. Section II describes the overview of dynamic analysis approach and terminologies. Section III defines the structure of run trace and the terminologies. Section IV introduces the structure analysis methods in detail. Section V discusses the details of the semantic analysis methods. Section VI explains the methods to optimize the obfuscated program into more analyzable shape. Section VII describes the implementation detail of the trace tool and Trudio. Section VIII shows the experimental results. Section IX compares our work to the previous works. Section X concludes the paper with its limitations and possible solutions.

II. Overview

A. Dynamic Approach

Even for a heavily obfuscated executable program must have the eventually equivalent behavior with the original program. Thus, the semantics of the program can be revealed if we observe the dynamic behaviors of the program. By analyzing the dynamic behaviors, we can acquire more accurate information about the structure of the program and transform the program into more analyzable shape.

The dynamic analysis comprises the following steps:

1) Extract the dynamic behaviors and generate a run trace from the target program. The run trace will save the executed machine instructions and the logs accessing registers or memory.
2) Analyze and visualize the extracted run trace to figure out the structure of the target program.
3) Analyze and visualize the run trace to understand the semantics of the target program.
4) Optimize the target program into more analyzable shape.

B. Trudio

We designed and implemented Trudio, which is the program analysis tool based on dynamic approach. This tool includes the implementations of the analysis and optimization algorithms presented in this paper.

Trudio does not have the tracing function, and the tracing tool is separately developed. Thus, Trudio takes a run trace extraceted from the tracing tool as input, and it covers step 2 through step 4 in the dynamic analysis workflow defined above. Therefore, you can investigate a run trace of the target program in detail.

Trudio targets on the executable programs for Microsoft Windows on Intel Architecture 32-bit (IA32). Though Trudio targets on a specific environment, most discussions in this paper are written not assuming a specific environment. However, some discussions mention about the architecture assumptions when it heavily depends on the implementation characteristic of the environment.

Trudio is available at [5].

C. Motivating Example

Before beginning technical discussions, let us show a motivating example. We wrote a program, which calculates and prints a fibonacci number using recursion. We applied virtualization obfuscation on the recursion function using VMProtect. The obfuscated program is much more complicated than the original program, and the obfuscated program was more than 100 times slower than the original program. It seemed to be almost impossible to analyze statically.

Nevertheless, we could observe that the semantics extracted by the semantic analysis from two programs were identical. Expression tree is the data structure designed to show the calculation process of a specific value. Figure 1 shows the expression trees generated by automatic tracking from the printed value of the program, and you can find out Figure 1a and 1b are identical. It means the extracted semantics of two programs were identical.

The expression trees are introduced in section V-C, and the detailed procedure to generate these expression trees are presented in section VIII-C.

III. Run Trace and Terminologies

A run trace is the record of an execution of the target program. A run trace contains the information of executed machine instructions and the change of status, i.e. the values in registers and memory.

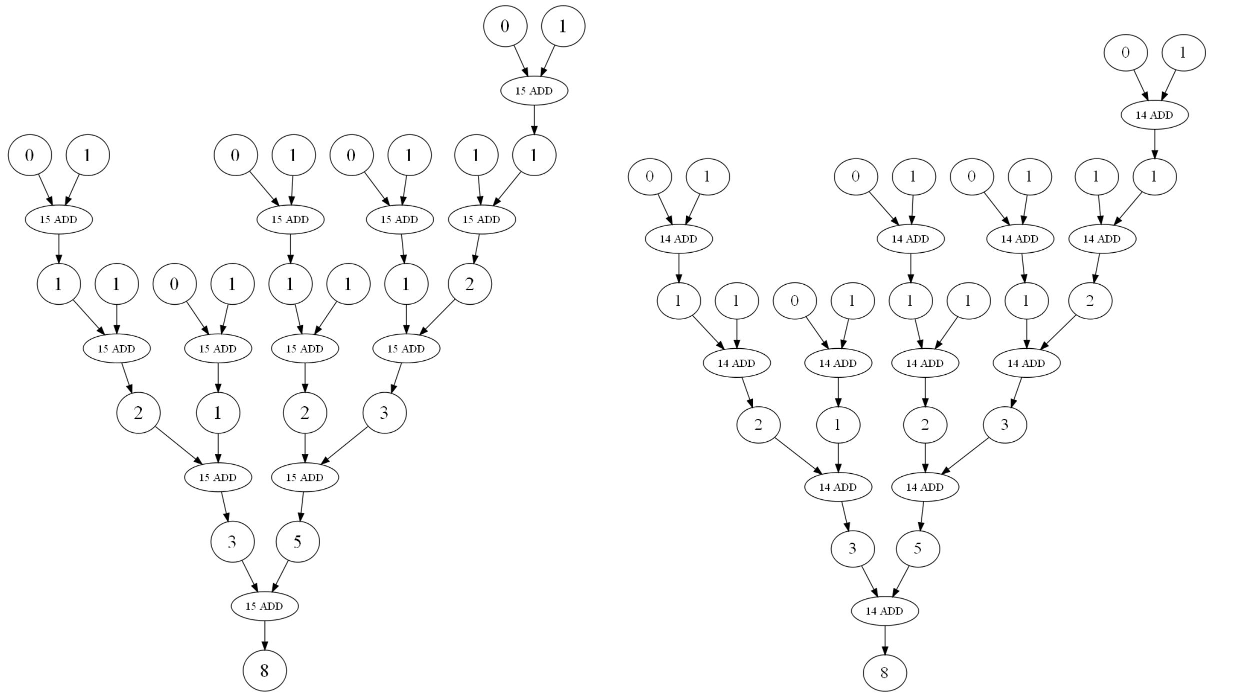
Basically, a run trace has a list of executed instructions. We call each of these executed instructions an instance of the trace, and the list is named as the full trace instance list. Those are called instances to differentiate from instructions of the trace. The full trace instance list may contain several instances of the same instruction. In other word, an instruction generates one or more instances, and an instance belongs to an instruction.

Each instruction is either inner or external. Inner instruction is from the target program, and external instruction is from modules loaded from other than the target program. An inner instruction represents one machine instruction, but an external instruction represents one external routine. It infers that an external instruction may represent multiple machine instructions.

An inner instruction has the information of the memory address where the instruction was placed and the byte sequence of the machine instruction. An external instruction saves the external routine name with the file path where the routine was loaded from.

An instance has the pointer to the instruction which it belongs and the index in the full trace instance list. Each instance also has a list of instance accesses.

The instance accesses represent the status accessed by the instance. The status of the system means the values in registers and memory. An instance access is tagged as one



(a) Original (b) Obfuscated

Figure 1. Expression trees of recfibo

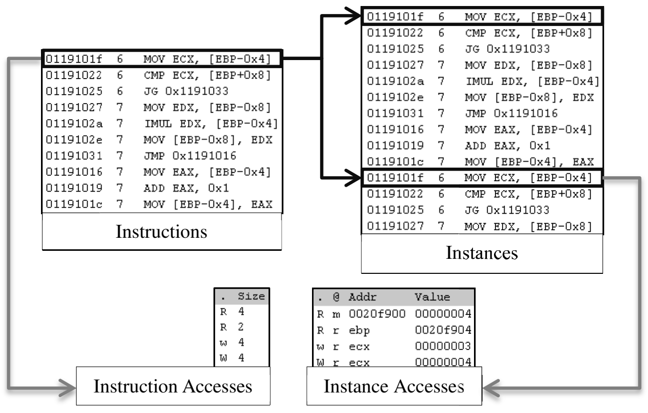
of read, writing, or written, and we call it as the type of the instance access. If an access is tagged as read, it indicates the instance has read where the instance access points out. When an access is tagged as writing or written, it means that the instance has written to register or memory where the access indicates. A writing access must be paired to another written access in the same instance, and vice versa. A writing access saves the value at the indicated location before the instance wrote a new value, and a written access has the value the instance has written.

An inner instruction also has a list of instruction accesses. Since each inner instruction represents its own machine instruction, the instance accesses of instances of an instruction must have the same format, that is, each instance must have the same number of read, writing, and written accesses. An instruction access is the abstraction of the matched instance accesses among those belonging to the instances of the instruction. An external instruction can cover multiple machine instructions, so the format of instance accesses of the instances may not be consistent. Therefore, external instructions do not have instruction accesses.

Figure 2 shows the structure of run trace.

In order to describe the algorithms using the run trace, we defined some functions about instances, instructions, and their accesses.

- $trace(k : Integer)$ returns the k^{th} instance in the full trace instance list assuming the full trace instance list has zero-based index.



```
0119101f  6   MOV ECX, [EBP-0x4]
01191022  6   CMP ECX, [EBP+0x8]
01191025  6   JG 0x1191033
01191027  7   MOV EDX, [EBP-0x8]
0119102a  7   IMUL EDX, [EBP-0x4]
0119102e  7   MOV [EBP-0x8], EDX
01191031  7   JMP 0x1191016
01191016  7   MOV EAX, [EBP-0x4]
01191019  7   ADD EAX, 0x1
0119101c  7   MOV [EBP-0x4], EAX
```

```
0119101f  6   MOV ECX, [EBP-0x4]
01191022  6   CMP ECX, [EBP+0x8]
01191025  6   JG 0x1191033
01191027  7   MOV EDX, [EBP-0x8]
0119102a  7   IMUL EDX, [EBP-0x4]
0119102e  7   MOV [EBP-0x8], EDX
01191031  7   JMP 0x1191016
01191016  7   MOV EAX, [EBP-0x4]
01191019  7   ADD EAX, 0x1
0119101c  7   MOV [EBP-0x4], EAX
0119101f  6   MOV ECX, [EBP-0x4]
01191022  6   CMP ECX, [EBP+0x8]
01191025  6   JG 0x1191033
01191027  7   MOV EDX, [EBP-0x8]
```

```
.  Size
R  4
R  2
w  4
W  4
```

```
. @ Addr      Value
R m 0020f900  00000004
R r ebp       0020f904
w r ecx       00000003
W r ecx       00000004
```

Figure 2. Structure of run trace

- $index(i : Instance)$ returns the index of i in the full trace instance list.
- $instruction(i : Instance)$ returns the instruction where i belongs.
- $place(I : Instruction)$ returns where the instruction I is placed, either $inner$ or $external$.
- $instances(I : Instruction)$ returns the set of instances belonging to I.
- $opcode(I : Instruction)$ returns the operation code of the machine instruction of I. I must be an inner instruction.
- $accesses(i : Instance)$ returns the list of instance

accesses belonging to i.

- $accesses(I : Instruction)$ returns the list of instruction accesses belonging to I. I must be an inner instruction.
- $instance(a : InstanceAccess)$ returns the instance where a belongs.
- $instruction(A : InstructionAccess)$ returns the instruction where A belongs.
- $instructionaccess(a : InstanceAccess)$ returns the instruction access where a belongs.
- $instanceaccesses(A : InstructionAccess)$ returns the set of instance accesses of A.

An instance access is always consecutive, so it always reads or writes a continuous array of bytes from the address space. Thus, an instance access saves the address of the first item and the size of the accessing array. We defined the functions to retrieve the information of instance access.

- $type(a : InstanceAccess)$ returns the type of a, one of $read$, $writing$, $written$.
- $size(a : InstanceAccess)$ returns the accessing length of a in integer.
- $address(a : InstanceAccess)$ returns the first accessing address of a.
- $addresses(a : InstanceAccess)$ returns the set of accessing addresses. Since an instance access must access address space consecutively, $addresses(a) = \{address(a) + k | 0 \leq k < size(a)\}$.
- $value(a : InstanceAccess)$ returns value of a as an integer. The implementation of this function may be different by the endian setting of the target computer architecture.
- $value(a : InstanceAccess, b : Address)$ returns the value of the byte read from or written to address b by instance access a. b must be in $addresses(a)$.
- $written(a : InstanceAccess)$ returns the written instance access paired to a. a must be writing instance access.
- $writing(a : InstanceAccess)$ returns the writing instance access paired to a. a must be written instance access.
- The functions $type$, $size$, $written$, $writing$ are similarly defined for instruction accesses.

An address is defined as a pair of address space and integer number representing the location in the address space. An address space is a consecutive array of bytes, so a location pointed by an address has a byte value. Address space is either register or memory, and each space is assumed to be separated. To fetch the property of an address, we defined the following functions.

- $space(a : Address)$ returns the address space of a, either register or memory.
- $location(a : Address)$ returns an integer representing the location of a in the address space.
- For an address a and an integer k, $a + k$ means $(space(a), location(a) + k)$.

In some computer architectures, the registers may not be directly mapped into an integer. In that case, you can map a register into an arbitrary integer number. Of course, different registers should not be overlapped. For instance, Trudio maps EAX register into 0 through 3 and EBX into 4 through 7. To improve the readability, we may use the notation like $(register, eax)$, and this address indicates the first location of the register EAX, that is, $(register, eax) = (register, 0)$.

The following names are associated with corresponding meanings throughout this paper.

- T is the full trace instance list.
- T_n is the length of T.
- Z is the set of all instructions, that is,
  $Z = \bigcup_{i \in T} \{instruction(i)\}$
- Z_i is the set of all inner instructions, that is,
  $Z_i = \{I \in Z | place(I) = inner\}$

In the algorithms listed in this paper, we will use the notation $A \Leftarrow B$ for two sets A and B. This means adding the items in B into A. In other word, $A \Leftarrow B$ is equivalent to $A \leftarrow A \cup B$.

IV. Structure Analysis of Obfuscated Program

The first step to analyze a program is understanding its structure. This section introduces the methodologies to understand the structure of the target program by the dynamic approach.

A. Control Flow Graph

Control flow graph (CFG) is a graph to show the structure of a program. In general term, control flow graph consists of vertices corresponding to basic blocks in the program and the edges indicating the possible control flows from block to block.

You can generate a control flow graph from a program using static analysis. However, there are some problems in static control flow analysis.

First of all, static control flow analysis may not detect indirect branches correctly. An indirect branch instruction jumps to the address that may be different in every execution. Since the target address of an indirect branch is decided while it runs, static analysis may not properly predict the target addresses.

Secondly, statically generated CFG will be distracted by control flow obfuscation. Control flow obfuscation inserts pointless control flow instructions including the branches that always jump to the same location. A basic block in the typical definition of CFG is a sequence of instructions whose instructions inside are neither jump instruction nor jump target, so a control flow instruction can only be located at the end of a basic block. Therefore, statically generated CFG has many dispensable vertices.

Dynamic approach is based on its real run trace, so it already knows the jump targets of indirect jumps. Moreover, it can figure out whether a branch is meaningless because it already knows that the branch actually has changed the control flow or not.

Inspired by this observation, we made a new definition of control flow graph for dynamic analysis. The new definition is different from the original in two aspects. First, a basic block is a sequence of instructions that have actually run in sequential order without jumping out or in. Second, an edge in the original CFG represents the possibility of control transfer, but an edge of the new definition means that the program control has really been transferred in that way.

To describe the control flow graph generation algorithm, we need to define two functions in and out.

- $in(I : Instruction)$ returns the set of instructions that have been executed right before I, that is, $in(I) = \bigcup_{k \in g(I)} \{instruction(trace(k-1))\}$ where $g(I) = \left(\bigcup_{j \in instances(I)} \{index(j)\}\right) - \{0\}$
- $out(I : Instruction)$ returns the set of instructions which have been executed right after I, that is, $out(I) = \bigcup_{k \in g(I)} \{instruction(trace(k+1))\}$ where $g(I) = \left(\bigcup_{j \in instances(I)} \{index(j)\}\right) - \{T_n - 1\}$

A new CFG contains vertices and edges. Each vertex will have a basic block, and an edge is represented as a pair of two vertices. The following describes the functions to retrieve the information about the CFGs.

- $block(i : Instruction)$ returns the block to which instruction i belongs.
- $instructions(b : BasicBlock)$ returns the list of instructions in b.

Algorithm 1 shows the pseudocode to generate the control flow graph.

Algorithm 1 Control flow graph generation

```
$CFG_V \leftarrow \{\}$
$CFG_E \leftarrow \{\}$
$tmp \leftarrow \{\}$
for $i \leftarrow 0$ to $T_n - 1$ do
  $I \leftarrow instruction(trace(i))$
  if $i = 0$ then
    $new \leftarrow a\,new\,block$
    $CFG_V \Leftarrow \{new\}$
    Append $I$ into $new$
  else
    $I_p \leftarrow instruction(trace(i-1))$
    if $I \notin tmp$ then
      if $|in(I)| \neq 1$ or $|out(I_p)| \neq 1$ then
        $new \leftarrow a\,new\,block$
        $CFG_V \Leftarrow \{new\}$
        Append $I$ into $new$
      else
        Append $I$ into $block(I_p)$
      end if
    end if
    if $block(I) \neq block(I_p)$ then
      $CFG_E \Leftarrow \{(block(I_p), block(I))\}$
    end if
  end if
  $tmp \Leftarrow \{I\}$
end for
$CFG = (CFG_V, CFG_E)$
```

B. Virtual Machine

This section discusses the case of the virtualization obfuscation technique. A program obfuscated using the virtualization obfuscation technique contains a virtual machine (VM) running a random instruction set and the original program code encoded into byte codes for embedded VM's instruction set. The obfuscated program virtually runs the encoded code of the original program, so the behavior will be identical to the original program. However, the shape will be completely different since the obfuscated program actually runs a virtual machine.

Virtual machines written for obfuscation purpose usually have simple structures. A virtual machine has a virtual program counter (VPC), a main VM routine, and several handlers. VPC indicates the address where next byte code to execute is located, and the main routine of VM reads a byte code using VPC and finds the memory address where the handler for the byte code is placed.

Main routines of VMs are generally have no branches inside, so a main routine typically appears in one block, and we call it main block of VM. We can find the main block and handlers of a VM based on the following characteristics of the main block.

- The number of execution is bigger than the other blocks.
- The number of out edges and in edges in CFG is bigger than the other blocks.
- The out edges from the main block indicate the byte code handlers.

After finding the main block of a virtual machine, we can find the handlers from its out edges. The instructions fetching and decoding the byte codes can be found in the block by tracking the instructions influencing the last instruction of the main block.

C. Access Map

Access map shows the memory area where the selected instruction has read or written.

This is particularly useful for a virtualization obfuscated program to find the memory area where the byte codes are located. If you find the instruction fetching the byte codes in the main block, you can seek the byte code area by tracking the memory area that the instruction accesses.

We can define the function $access$ for this purpose.

- $access(i : Instance)$ returns the set of addresses where i accesses, that is,
  $access(i) = \bigcup_{a \in accesses(i)} addresses(a)$
- $access(I : Instruction)$ is the abstraction of $access$ from instance to instruction, that is,
  $access(I) = \bigcup_{i \in instances(I)} access(i)$

Reversely, you also can find the instances or instructions accessing a memory address.

- $accessat(a : Address)$ returns the set of instructions which access at a, that is,
  $accessat(a) = \{I \in Z_i | a \in access(I)\}$

V. Semantic Analysis of Obfuscated Program

Trudio has the tool showing a memory map highlighting the addresses $access(i)$ for an instruction i selected by user.

A. Value History

Most instruction set architectures are designed to be imperative, so the machine instructions change the status of the system. Thus, the status is constantly changing while a program is executed. And the changes are recorded in run trace as instance accesses. So we can reenact the status between instances.

For an address a and an integer $0 \leq n < T_n$, V_n^a has the value at address a right after the $trace(n)$ executed. V_n is the set of the available addresses right after the $trace(n)$ executed. Algorithm 2 shows the algorithm to calculate V. U_n is the set of addresses updated while processing the n^{th} trace.

Algorithm 2 Value history

```
for $n \leftarrow 0$ to $T_n - 1$,
    $U_n \leftarrow \{\}$
    for $a \in accesses(trace(n))$ do
        if $type(a) = written$ then
            for $k \in addresses(a)$ do
                $V_n^k \leftarrow value(a, k)$
                $U_n \Leftarrow \{k\}$
            end if
        end if
    end for
    if $n > 0$ then
        for $k \in V_{n-1} - U_n$,
            $V_n^k \leftarrow V_{n-1}^k$
        end for
    end if
end for
```

B. Dependency Tracker

If the value read by an instance access a_1 is written by another instance access a_0, we say that a_1 depends on a_0 and a_0 influences a_1. We can represent these relationships as a graph among the instance accesses. This graph shows the data flow and the calculating process, that is, the semantics or algorithm. We call this graph a dependency graph. The dependency graph is a great tool to understand the behavior of the program.

Let D be the dependency graph of the run trace.

- $D = (D_V, D_E)$
- D_V is the set of all instance accesses
- $D_V = \bigcup_{i \in T} accesses(i)$
- $D_E : \{(InstanceAccess, InstanceAccess)\}$
- $\forall (f, s) \in D_E$, $type(f) = written$ and $type(s) = read$
- $(f, s) \in D_E$ means that f influences s, i.e. s depends on f.

We also need to define a overwriting graph W.

- $W : \{(InstanceAccess, InstanceAccess)\}$
- $\forall (f, s) \in W$, $type(f) = type(s) = written$
- $(f, s) \in W$ means that s has overwritten the value written by f.

Before we define the algorithm to calculate D_E and W, we should define a dictionary named O first. For an address a and an integer $0 \leq n < T_n$, O_n^a returns the instance access which has most recently written to the address a before $trace(n)$. O_n is the set of the addresses where O_n^a is available. Assume $O_{-1} = \{\}$. Let O be called overwritten dictionary. Algorithm 3 shows the algorithm to calculate the overwritten graph W and the overwritten dictionary O. Algorithm 4 shows the algorithm to generate dependency graph using the calculated overwritten dictionary O.

Algorithm 3 Overwritten dictionary and graph generation

```
for $n \leftarrow 0$ to $T_n - 1$ do
    for $a \in accesses(trace(n))$ do
        if $type(a) = written$ then
            for $k \in addresses(a)$ do
                if $O_n^k$ is available,
                    $W \Leftarrow \{(O_n^k, a)\}$
                end if
                $O_n^k \leftarrow a$
            end for
        end if
    end for
    for $k \in O_{n-1} - O_n$ do
        $O_n^k \leftarrow O_{n-1}^k$
    end for
end for
```

We can define some functions to use D and W more easily.

Algorithm 4 Dependency graph generation

```
$D_E \leftarrow \{\}$
for $n \leftarrow 0$ to $T_n - 1$ do
  for $a \in accesses(trace(n))$ do
    if $type(a) = read$ then
      for $k \in addresses(a)$ do
        if $k \in O_n$ then
          $D_E \Leftarrow \{(O_n^k, a)\}$
        end if
      end for
    end if
  end for
end for
```

- $depends(a : InstanceAccess)$ returns the set of instance accesses that influences a, that is, $depends(a) = \bigcup_{(k,a)\in D_E}\{k\}$. a should be a read access and assume $depends(a) = \{\}$ if a is not a read access.
- $forwards(a : InstanceAccess)$ returns the set of instance accesses that depends on a, that is, $forwards(a) = \bigcup_{(a,k)\in D_E}\{k\}$. a should be a written access and assume $forward(a) = \{\}$ if a is not a written access.
- $overwrites(a : InstanceAccess)$ returns the set of instance accesses that have overwritten the value written by a, that is, $overwrites(a) = \bigcup_{(a,k)\in W}\{k\}$. a should be a written instance access and assume $overwrites(a) = \{\}$ if a is not a written access.

We can define the abstraction functions for instruction accesses.

- $depends(A : InstructionAccess) =$
  $\bigcup_{a\in instanceaccesses(A)} [g(a)]$ where
  $g(a) = \bigcup_{d\in depends(a)} (instructionaccess(d))$.
- $forwards$ and $overwrites$ are similarly defined.

For real application, the whole dependency graph is too huge for an analyzer to understand. Moreover, most analyzers do not concern about the every dependency relation, but they only want to know the process to calculate a specific value. For example, the instance accesses used as the arguments for an external routine generally decide the behavior of the program. Therefore, analyzers will want to know how the argument values have been calculated.

Thus we need to extract a subset from dependency graph illustrating the relations connected to the user specified value. We call the subsets of dependency graph the subdependency graphs. And a relevant subdependency graph is a subdependency graph which extracts only the relations directly and indirectly connected to the value designated by the user.

A subdependency graph S has the following characteristics.

- S is a subset of dependency graph D, that is, $S = (S_V, S_E)$ where $S_V \subseteq D_V$ and $S_E \subseteq D_E$.
- $S_V = \bigcup_{(f,s)\in S_E}\{f, s\}$

C. Expression Tree

An obfuscated program generally has more complicated calculation process than the original program. For example, an obfuscated calculation procedure may have unnecessary move instructions. The analyzers may not be interested in the meaningless instructions such as pointless moves.

Expression tree is the graph generated from a relevant subdependency graph, skipping the instructions that the analyzer does not interest in. Since an expression tree does not show the uninteresting behaviors, you can see the semantic of the program more vividly.

To generate an expression tree, you should first set a subdependency graph S and the set of interesting instructions N. An interesting instruction is the instruction that you want to include in the expression tree.

The fundamental difference of expression tree from the subdependency graph is the existence of bypassing edges. Suppose that there are three different instance accesses a, b, and c, and a influences b and b influences c. If b belongs to an instance of uninteresting instruction, you want to skip the edges going through b and want a bypassing edge (a, c) instead. Because of bypassing edge, an expression tree is not necessarily a subset of dependency graph.

An expression tree E is defined as follows:

- $E = (E_V, E_E)$
- $E_V : \{InstanceAccess\} \subset S_V$
- $E_E : \{(InstanceAccess, InstanceAccess)\}$
- E_E is not necessarily a subset of S_E because E_E can have the bypassing edges. However, still $\forall(f, s) \in E_E$, $type(f) = written$ and $type(s) = read$.

The key point of expression tree generation algorithm is finding the bypassing edges. The function $forwardbypass$ can be used to find bypassing edges.

- $forwardbypass(i : InstanceAccess)$ returns the set of read instance accesses which should replace i. i must be a read instance access, and every item in the returned set of $forwardbypass$ will be read instance access.
- $backwardbypass$ can be defined similarly, but we only use $forwardbypass$.

Algorithm 6 shows the algorithm to generate the expression tree E from S and N, using $forwardbypass$ function.

The implementation of expression tree in Trudio uses a set of interesting operation codes instead of a set of interesting instructions. In other word, for a designated set of interesting operation codes P, $N = \{i \in Z_i | opcode(i) \in P\}$.

VI. Optimization of Obfuscated Program

A. Optimization

The optimization in this paper aims to transform an obfuscated program into more analyzable shape. An obfuscated

Algorithm 5 $forwardbypass$ function

```
def $forwardbypass(i : InstanceAccess)$
  if $instruction(instance(i)) \in N_V$ then
    return $\{i\}$
  else
    $result \leftarrow \{\}$
    for $j \in forwards(i)$ do
      for $k \in accesses(j)$ do
        if $type(k) = read$ and $k \in S_V$ then
          $result \Leftarrow forwardbypass(k)$
        end if
      end for
    end for
    return $result$
  end if
end def
```

Algorithm 6 Expression tree generation

```
$E_V \leftarrow \{\}$
$E_E \leftarrow \{\}$
for $(f, s) \in S_E$ do
  if $instruction(instance(f)) \in N_V$ then
    $bypasses \leftarrow forwardbypass(s)$
    $E_V \Leftarrow \{f\} \cup bypasses$
    $E_E \Leftarrow \bigcup_{k \in bypasses}\{(f, k)\}$
  end if
end for
```

program will be easier to analyze if unnecessary instructions and decoding routines are eliminated.

Therefore, our optimization is basically finding the inner instructions to be dropped from the program. The external instructions cannot be dropped, but they are considered important throughout the whole optimization process. In addition, removal of an instruction may requires some modifications on other instructions or data in memory. Thus, an optimization is defined as a 3-tuple of (dropped instructions, modified instructions, modified data).

- $Optimization = (Dropped, Modified, Data)$
- $Dropped$: $\{Instruction\}$ is the set of dropped instructions. Every instruction in $Dropped$ must be inner instruction.
- $Modified : \{Instruction \rightarrow Instruction\}$ is the dictionary mapping from original instruction to modified instruction.
- $Data : \{Address \rightarrow Byte\}$ is the dictionary mapping from memory address to modified byte(integer).

Our optimization is in seven steps:

1) Milestone establishment
2) Trace pruning by dependency analysis
3) Removal of meaningless instructions
4) Removal of effectless instructions
5) Removal of useless stack operations
6) Instruction chain reduction
7) Executable patch

Steps 1 and 7 are necessary, and steps 2 through 6 are optional. Step 7 is technically not an optimization step, but it is generating an executable program reflecting the optimization.

B. Milestone Establishment

Some instructions in the program must not be deleted, and those instructions are called milestones. Milestone establishment finds and sets the milestones, and it is the fist and mandatory step of optimization.

Milestones are searched by three criteria: instruction calling an external routine, relevant instructions to external instruction, and the last instructions in basic blocks. We assumed every external instruction is important, so the instructions which call or directly affect an external instruction should be milestones. And the last instruction of a basic block has actually changed the control flow, so they should not be eliminated.

For the simplicity of writing, let us define a function $influences$.

- $influences(I : Instruction)$ returns the set of instructions on which an instruction access of I influences, that is, $influences(I) = \bigcup_{A \in g(I)}\{instruction(A)\}$ where $g(I) = \bigcup_{k \in accesses(I)} forwards(k)$

Then, the set of milestones M is
$M = \{k \in Z_i | p_1(k)\ or\ p2(k)\ or\ p_3(k)\}$ where $p_1(k) = (out(k) \not\subseteq Z_i)$, $p_2(k) = (\{j \in influences(k) | j \notin Z_i\} \neq \{\})$, and $p_3(k) = (k = the\ last\ item\ of\ instructions(block(k)))$.

C. Trace Pruning by Dependency Analysis

The instructions directly or indirectly influencing milestones are called relevant instructions. Since the relevant instructions also affect to the behavior of the program, they should not be removed.

Algorithm 7 shows how to find the set of relevant instructions. R is the set of instances belonging to relevant instructions.

D. Removal of Meaningless Instructions

An instruction is meaningless if it has never affected the status of the system throughout the whole run trace. There are two cases: meaningless branch and instructions unchanging the status.

If a control flow instruction is not at the end of the block, the instruction is meaningless. The only instructions at the end of a block have effectively changed the control flow. The control flow instructions which has never changed the control flow cannot be meaningful.

Another case is instructions that have not changed the status. In other word, for an inner instruction I, I is

Algorithm 7 Trace pruning by dependency analysis

```
def $relevant(i : Instance, R : \{Instance\})$
  if $i \notin R$ and $place(instruction(i)) = inner$ then
    $R \Leftarrow \{i\}$
    for $a \in accesses(i)$ do
      for $d \in depends(a)$ do
        $R \Leftarrow relevant(d, R)$
      end for
    end for
  end if
  return $R$
end def

$R \leftarrow \{\}$
for $I \in M$ do
  for $i \in instances(I)$ do
    $R \leftarrow relevant(i, R)$
  end for
end for
$Dropped \Leftarrow \{I \in Z_i | I \notin \{instruction(i) | i \in R\}\}$
```

meaningless if $\forall i \in instances(I)$, $\forall a \in accesses(i)$, $type(a) = written$ and $value(writing(a)) = value(a)$.

E. Removal of Effectless Instructions

An instruction is effectless if and only if the instruction influences only the dropped instructions. Therefore, this optimization step is dependent on the optimization progress. The set of all effectless instructions is

$\{k \in Z_i | influences(k) \subseteq Dropped\}$

F. Removal of Useless Stack Operations

This section describes removal of needless stack operations. The discussions in this section is heavily dependent on the implementation of the stack operations. Thus, this section is written assuming IA32.

IA32 has stack operations such as PUSH and POP. A portion of memory is assigned as a stack. The memory address of the top item of the stack is saved in ESP register, and the bottom item in EBP register. The stack is grown to the lower address, that is, the value of ESP is decreased when a new item is pushed into the stack.

You can consider the stack operations in IA32 are just wrapping instructions. For example, PUSH instruction is a composite of MOV the value to the stack and SUB to the ESP register. In other word, there are no fine restrictions in using stack such as a rule that the stack must be accessed only with PUSH and POP instructions. For example, you can access the 4th value from top in the stack using MOV EAX, [ESP+0x10].

Hence, to remove a stack operation, some instructions run between PUSH and POP may need modification. Suppose you want to delete a pair of PUSH and POP, but there is MOV EAX, [ESP+0x10] instruction right after PUSH. Then this instruction should be modified to MOV EAX, [ESP+0xC] since ESP will be larger by 4 if the PUSH is removed.

While the removal of stack operation is tricky, finding useless stack operation is rather easy. A stack operation can be used in two ways: value and position. A normal PUSH should be used to save a meaningful value to the stack. In this case, some instructions will use the value pushed into the stack by PUSH instruction. Such PUSH and corresponding POP instruction pairs are value-meaningful. A stack operation is called position-meaningful if there is an instruction that overwrites to the location where the pushed value is located. If a stack operation is not neither value-meaningful nor position-meaningful, we can remove the stack operation pair.

The first step to find the removable stack operations is to find pairs of push and pop. In IA32, after a PUSH was executed, if ESP goes higher than the value of ESP before the PUSH executed, we can consider the pushed value is now popped from the stack. Therefore, we can find the pop instruction of PUSH traversing the instances following the instances of PUSH.

While we find the push and pop pairs, we will also traverse the instructions between push and pop. We call these instructions the insider instructions. We will also find the instructions that need modification among the insiders. These instructions are called affected instructions.

Some push and pop pairs are not valid. If the value pushed to the stack by a PUSH instruction is popped by multiple instructions, we consider these stack operations are improper. If the insiders and affected instructions of a push pop pair are not consistent for each instance of push and pop, this stack operation is also inappropriate.

Algorithm 8 shows the algorithm to find proper stack operation pairs. SO is the set of 4-tuples of push instruction, corresponding pop instruction, size of pushed value, and affected instructions.

- $memop(i : Instance)$ returns a read or written instance access of i accessing the memory. $instruction(i)$ must be inner instruction. If there is no memory operand, this function will returns $\{\}$. The returned set of $memop$ will have maximum one item since the instructions of IA32 have one memory operand at most. That is, $memop(i) = \{a \in accesses(i) | space(address(a)) = memory\}$
- $espaccess(i : Instance)$ returns the set of instance accesses reading from the ESP register. That is, $espaccess(i) = \{a \in accesses(i) | type(a) = read, address(a) = (register, esp)\}$
- $pushedvalue(i : Instance)$ returns the written instance access belonging to i writing the value into the stack. $instruction(i)$ must be inner push instruction.

Algorithm 8 Push and pop pairing

```
$SO \leftarrow \{\}$
for $I \in Z_i$ do
  if $opcode(I) \in \{PUSH, PUSHF, PUSHA\}$ then
    $pop \leftarrow \{\}$
    $psz \leftarrow nil$      ; pushed size
    $ins \leftarrow nil$      ; insiders
    $aff \leftarrow nil$      ; affected instructions
    for $i \in instances(I)$ do
      $esp \leftarrow espaccesses(i)$
      ; $|esp|$ must be 1
      $k \leftarrow index(i)$
      $psz_i \leftarrow size(pushedvalue(i))$
      $ins_i \leftarrow \{\}$
      $aff_i \leftarrow \{\}$
      do
        $k \leftarrow k + 1$
        if $k \geq T_n$ then goto $break$
        $ins_i \Leftarrow \{instruction(trace(k))\}$
        $mem \leftarrow memop(trace(k))$
        if $mem \neq \{\}$ and
           $address(mem)$ is in stack area and
           $address(mem) \geq esp$ then
         $aff_i \Leftarrow \{instruction(trace(k))\}$
        end if
        $esp_n \leftarrow espaccess(trace(k))$
      while $|esp_n| = 1$ and $value(esp_n) \geq value(esp)$
      if ($psz \neq nil$ and $psz \neq psz_i$) or
         ($ins \neq nil$ and $ins \neq ins_i$) or
         ($aff \neq nil$ and $aff \neq aff_i$) then
        goto $break$
      end if
      $psz \leftarrow size(pushedvalue(i))$
      $ins \leftarrow ins_i$
      $aff \leftarrow aff_i$
      $pop \Leftarrow \{instruction(trace(k))\}$
    end for
    if $|pop| \neq 1$ then goto $break$
    $SO \Leftarrow \{(I, pop, psz, aff)\}$
  end if
$break$:
end for
```

After we got the set of stack operation pairs, we must find the useless stack operations. We first find value-meaningful or position-meaningful stack operations. If both are empty, we could choose the stack operations to remove. Algorithm 9 shows the algorithm to find useless stack operations.

Algorithm 10 shows a brief sketch of stack operation removal algorithm. Removal of stack operations is heavily dependent on the detail of the instruction set architecture, hence we do not discuss the details in this paper.

Algorithm 9 Search of useless stack operation

```
$SO_{useless} \leftarrow \{\}$
for $so \in SO$ do
   $(push, pop, psz, aff) \leftarrow so$
   $valmean \leftarrow \{\}$
   $posmean \leftarrow \{\}$
   for $i \in instances(push)$ do
      $pushed \leftarrow pushedvalue(i)$
      $valmean \Leftarrow forwards(pushed)$
      $posmean \Leftarrow overwrites(pushed)$
   end for
   if $valmean = \{\}$ and $posmean = \{\}$ then
      $SO_{useless} \Leftarrow so$
   end if
end for
```

Algorithm 10 Stack operation removal

```
for $so \in SO_{useless}$ do
   $(push, pop, psz, aff) \leftarrow so$
   if $\exists m \in aff$, $m$ is not supported then
      goto $break$
   end if
   $Dropped \Leftarrow \{push\}$
   Drop or modify $pop$
   for $m \in aff$ do
      Move displacement of $m$ by $-psz$
   end for
$break$:
end for
```

G. Instruction Chain Reduction

Data encoding is a common obfuscation technique. A program obfuscated with this technique saves the data in encoded form and has routines to decode them. For example, some programs obfuscated with virtualization obfuscation have the decoding routines for byte codes. When the main block of VM fetches a byte code, the loaded value is not really a byte code but it requires decoding process. This is why the previous reversing techniques are barely possible to apply to the programs obfuscated by recent obfuscators. However, if we can remove decoding routines and make the program directly fetch the decoded byte code from memory, then those works may be applicable for the optimized programs. Moreover, the executable program without decoding routines and encoded data will be much simple to analyze.

This optimization technique is aimed to remove the decoding routines. We observed two facts: the encoded data must be decoded before they are used, and the data in the middle of decoding process are used only by the decoding routine. Inspired by these observations, we defined reducible instruction chain and designed the algorithms to find and remove reducible instruction chains.

A reducible instruction chain is composed of source, destination, member instructions, and replacing instruction. Source of a chain is the instruction access that first fetches the encoded data. And then, the value goes through the member instructions, and finally decoded value is saved into the destination of chain. A chain can be reduced in the following steps:

1) Save the value of chain's destination into the location of chain's source.
2) Replace the instruction of chain's destination with replacing instruction.
3) Drop all member instructions other than replaced instruction.

Through this procedure, we can remove the decoding routines and optimize the program to directly fetch the decoded data from memory.

However, some instruction chains cannot be reduced even if we found them. For example, on the architectures whose instructions have different lengths(CISC), if the length of the instruction of chain's destination is shorter than replacing instruction, then it is not possible to replace a shorter instruction with a longer one.

Formally, a reducible chain can be represented as a 4-tuple: (source, destination, members, replacing)

- $source$: $InstructionAccess$ is the instruction access where encoded data are first loaded. $source$ must be read from memory.
- $destination$: $InstructionAccess$ is the instruction access where decoded data are finally saved. $destination$ must be written access.
- $members$: $\{Instruction\}$ is the set of instructions in the reducible chain. The instructions of $source$ and $destination$ should be in $members$. All instructions in $\{I \in members | I \neq instruction(destination)\}$ will be dropped in the reduction procedure.
- $replacing$: $OPCODE$ is used to reduce the instruction chain. In the reduction process, the instruction of $destination$ is replaced by $replacing$ to let the program directly get the decoded data into $destination$. However, you may notice that $replacing$ is just an operation code, not a full instruction. A replacing instruction is always a move instruction from source to destination, so we already know the operands and the action. But move operation can be various as Intel architecture has the plain move, move with sign extension, move with zero extention, etc. Thus $replacing$ must indicate one of move operation codes.

When we find recudible instruction chains, we first find the starting points of candidate chains and propagate them. To find the starting points of reducible chains, we assumed encoded data are read only by one instruction and the addresses where the encoded data are saved will not be accessed by other instructions. This assumption means that an obfuscated program will not have more than one decoding routines of the same algorithm.

Algorithm 11 shows the mechanism to find the starting points of reducible chains.

Algorithm 11 Search of starting points of reducible instruction chains

```
$RC_{first} \leftarrow \{\}$
for $I \in Z_i$ do
  for $A \in accesses(I)$ do
    if $type(a) \neq read$ then goto $break$
    for $a \in instanceaccesses(A)$ do
      if $space(address(a)) \neq memory$ then goto $break$
      for $j \in addresses(a)$ do
        if $accessat(j) \neq \{I\}$ then goto $break$
      end for
    end for
  end for
  $RC_{first} \Leftarrow \{I\}$
$break$:
end for
```

From the starting points RC_{first} found in Algorithm 11, we can propagate the reducible chain when an instruction in the chain influences exactly one instruction. Algorithm 12 shows the reducible chain propagation algorithm. The following shows the definition of functions about instructions used in Algorithm 12.

- $source(i : Instruction)$ returns the read instruction access that is the source operand of trace instruction i. This function will return nil if there is no source operand.
- $destination(i : Instruction)$ returns the written instruction access that is the destination operand of trace instruction i. This function will return nil if there is no destination operand.

Algorithm 13 shows the algorithm reducing an instruction chain. If any assertion in the for loop is violated, the reducible chain cannot be reduced, and the progresses for the chain should be restored. Assume that the function $keys$ appeared in Algorithm 13 returns the key set of the dictionary.

H. Executable Patching

Executable patching is the final step of optimization. This step generates an optimized version of the obfuscated program using the information derived from the optimization procedure. This step includes replacing the dropped instructions into no operation(NOP) and applying the modifications on instructions and data.

However, it requires attention while handling control instructions. If a control instruction is removed, the instruction which should be executed after the control instruction may

Algorithm 12 Propagation of reducible instruction chains

```
$RC \leftarrow \{\}$
for $f \in RC_{first}$ do
    if $source(f) \neq nil$ then
        $next \leftarrow f$
        $chn \leftarrow \{\}$
        $rpl \leftarrow nil$
        repeat
            if $opcode(next)$ is move opcode then
                $rpl \leftarrow opcode(next)$
            end if
            $chn \Leftarrow next$
            if $|influences(next)| \neq 1$ then
                break repeat
            end if
            $next \leftarrow influences(next)$
        end repeat
        if $rpl \neq nil$ then
            $RC \Leftarrow (source(f), destination(next), chn, rpl)$
        end if
    end if
end for
```

Algorithm 13 Instruction chain reduction

```
for $rc \in RC$ do
    $(src, dst, chn, rep) \leftarrow rc$
    $Modified \Leftarrow \{(instruction(dst) \rightarrow "rep\, dst\, src")\}$
    $srcinst \leftarrow instanceaccesses(src)$
    $dstinst \leftarrow instanceaccesses(dst)$
    assert $|srcinst| = |dstinst|$
    for $k \leftarrow 0$ to $|srcinst|$ do
        $s \leftarrow srcinst[k]$
        $d \leftarrow dstinst[k]$
        assert $size(s) = size(d)$
        $a_s \leftarrow address(s)$
        $a_d \leftarrow address(d)$
        for $l \leftarrow 0$ to $size(s) - 1$ do
            assert $a_s + l \notin keys(Data)$
            $Data \Leftarrow \{((a_s + l) \rightarrow value(d, a_d + l))\}$
        end for
    end for
    $Dropped \Leftarrow (chn - \{instruction(dst)\})$
end for
```

not follow the control instruction. This will make an error while running the optimized program, so we have to preserve such control instructions.

VII. Implementation Detail

A. Target Architecture

Intel architecture in 32-bit, IA32 is the most widely used instruction set architecture, and Microsoft Windows is the most popular personal computer operating system in the market. Thus our implementation targets to the programs for Microsoft Windows running on IA32.

B. Tracing

Prior to the implementation of analysis techniques, we should have developed a tracing tool. At first, we had considered using debuggers suchas IDA Pro [6] or OllyDbg [7]. But these debuggers use the single step exception to trace, thus they could not properly trace some obfuscated executable programs. Secondly, we had examined emulators. There are several good emulators such as QEMU [8] or Bochs [9]. But these emulators run the whole system whereas we needed an emulator running on an application program. So, we chose Pin [10].

We wrote a tracing tool using the APIs of Pin. The tracing tool was designed to record and monitor every executed instruction(instance) and all read or write accesses on register or memory. On the other hand, this tracing tool does not track the instructions in external modules since they are out of scope of the analysis. The tracing tool records only the first instruction of called external module as external instruction. However, instance accesses are still recorded for an external instruction.

This tool is designed to be separated from the analysis process. Therefore, if you write a new tracing tool, you can use it only if it generates run traces of the same format.

The tracing tool is available at [11].

C. Trudio

The analysis algorithms described so far are implemented and integrated into one analysis tool named Trudio. The name Trudio is the compound word of trace and studio. It shows the characteristic of this tool, which does not automatically reverse the obfuscation, but provides the analysis methods and optimizing functionality to help analyzing the obfuscated software.

Trudio is developed in Java language with Swing GUI toolkit, distorm3 [12] as disassembler, and Netwide Assembler [13] as assembler. The total lines of code is over 13000 lines.

Trudio is available at [5].

VIII. Experiments and Results

A. Experiment Overview

To test the effectiveness of the approach presented in this paper, we designed experiments examining the methodologies for structure analysis, semantic analysis, and optimization. For the experiments, we wrote sample programs: imperative factorial(impfact), imperative fibonacci(impfibo), recursive factorial(recfact), and recursive fibonacci(recfibo). recfibo was used as a motivating example. Every sample program calculates a factorial or fibonacci number, prints out the value, and terminates.

We intentionally designed a sample program to get no input from user and has one execution path since the problem of generating test cases covering many execution paths is out of scope of this paper.

We obfuscated the executable programs using VMProtect, and the obfuscation was applied on the main calculation routine of each sample program.

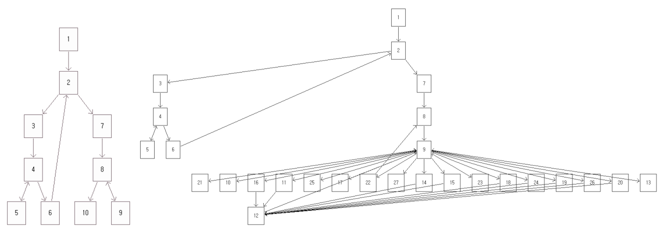
The fact that we chose VMProtect does not mean Trudio is limited to the programs obfuscated using virtualization. We chose VMProtect just because we supposed that this is one of the most advanced obfuscators in the market.

Table I shows the summary of each sample program.

Name	Original		VMProtect	
	Instructions	Instances	Instructions	Instances
impfact	322	407	896	222036
impfibo	324	465	899	294173
recfact	316	428	952	248648
recfibo	328	734	994	784807

Table I
SUMMARY OF SAMPLE PROGRAMS

Figure 3 shows the examples of control flow graphs generated from impfact. Figure 3a shows the CFG of original impfact, and Figure 3b shows the CFG of obfuscated impfact. The obfuscation was applied to the codes at the block 8 and 9 in the original program. You can easily see that the structure of original program and obfuscated program are completely different. The obfuscated program has a main routine of virtual machine to read, decode and call the handlers at the block 9 with many handlers.



(a) Original (b) Obfuscated

Figure 3. Control flow graphs of impfact

The sample programs and experimental results are available at [14].

B. Structure Analysis

All the control flow graphs generated from the obfuscated programs look similar to Figure 3b; one main block and many handlers. Static analyzers could not find the edges from the main block to handlers since the indirect calls are used when the main block calls handlers. On the contrary, we could figure out all the edges from the main block to handlers.

We could evaluate the utility of the schemes to find the main block of virtual machine. As we previously discussed, an executable program obfuscated using virtualization obfuscation has its main block, and the main block is called the most times and has the most number of out edges.

Table II shows the name of the block executed the most and the second most times. The table also enumerates the name of the blocks that have the most and the second most out edges with the number of out edges. You can see the block that is executed most times has the largest number of out edges. Those blocks were actually the main blocks of the embedded virtual machines. You also find that the big gap exists between the most block and the second most block, and it proves that the property we guessed about main blocks were correct.

Name	Execution count		Out edges	
	1st	2nd	1st	2nd
	Block / #	Block / #	Block / #	Block / #
impfact	9 / 3062	12 / 1168	9 / 17	2, 4 / 2
impfibo	9 / 3755	13 /1424	9 / 16	2, 4, 10, 13 / 2
recfact	10 / 3239	13 / 1274	10 / 16	2, 4, 25 / 2
recfibo	10 / 10769	13 / 4271	10 / 15	30 / 3

Table II
VIRTUAL MACHINE MAIN BLOCK

C. Semantic Analysis

A sample program calculates an integer value, prints out the value via printf function, and terminates. The argument delivered to printf function is pushed into stack right before printf function is called, and the value is the result of the calculation routine. Thus, the pushed value is a good starting point to generate a relevant subdependency graph to find out how the printed value has been calculated. We generated expression trees from the extracted subdependency graph and the set of operation codes appeared in the expression tree of original programs excluding move and stack instructions.

You have already seen the visualized expression trees generated from original recfibo and obfuscated recfibo as a motivating example in Figure 1a and Figure 1b. Both expression trees are generated from the relevant subdependency graph extracted from the printed value, and shows only the instructions of operation code ADD. Operation codes ADD, MOV, POP, PUSH, XOR appeared in the relevant subdependency graph of original recfibo, and MOV, POP, PUSH were excluded because they are move or stack operations, and XOR was used to compare a value to 0. Thus we chose ADD to filter the instructions that appeared in the expression tree.

Those two graphs are identical in structure and values in vertices. The only difference is that the ADD instruction belongs to block 15 in the original program whereas it belongs to block 14 in the obfuscated program. You can infer block 14 is the VM handler for addition.

Similarly, we generated and compared the expression trees of original and obfuscated versions of each sample program. Table III shows the operation codes we used for expression tree filtering. Ori indicates the operation codes appeared in original version, Obf means the obfuscated version, and Use lists the operation codes filtering the expression tree. As we stated before, the operation codes were chosen from the OPCODEs appeared in the expression tree of original version excluding move and stack operations. While we obfuscate recfact, however, VMProtect substitutes SUB instruction by ADD instruction, so we produces the expression tree with ADD instructions for obfuscated version instead of SUB of original program.

Name		OPCODEs
impfact	Ori	ADD, IMUL, MOV, PUSH
	Obf	ADD, CWDE, IMUL, MOV, POP, PUSH
	Use	ADD, IMUL
impfibo	Ori	ADD, MOV, PUSH
	Obf	ADD, CWDE, MOV, POP, PUSH
	Use	ADD
recfact	Ori	IMUL, MOV, PUSH, SUB
	Obf	ADD, AND, CWDE, IMUL, MOV, NOT, POP, PUSH
	Use	IMUL, SUB / ADD, IMUL
recfibo	Ori	ADD, MOV, POP, PUSH, XOR
	Obf	ADD, AND, CWDE, MOV, NOT, PUSH
	Use	ADD

Table III
SEMANTIC EXTRACTION SCORE

The expression trees of original program and obfuscated version were identical for impfact, impfibo, and recfibo. Since the obfuscated recfact used ADD instruction to decrement a variable instead of SUB instruction as in the original program, the graphs of recfact were slightly different.

The result shows the semantic analysis approach presented in this paper properly discloses the behavior of the obfuscated program.

D. Optimization

We applied the proposed optimization techniques to the obfuscated program of each sample. Every optimized executable program runs properly and generates the identical result with the obfuscated program before it was optimized.

The basic measure of the optimization is the number of dropped instructions and the number of instances belonging to the survived instructions. The number of survived instances is calculated with the following equation:

$$\left|\bigcup_{I\in\{k\in Z_i|k\notin Dropped\}} instances(I)\right|$$

Table IV shows the comparisons of the numbers of survived instructions and instances before and after optimization. Ori means the numbers of original programs while Opt indicates the numbers of optimized versions.

You can see that the number of instructions are reduced about 42 to 45%, and the number of instances are reduced by 50 to 63%. You can see that the decreasing rate of instances is bigger than the rate of the instructions. The reason is that many of the dropped instructions are placed in the virtual machine main block or handlers, which run multiple times.

Name	Instructions			Instances		
	Ori	Opt	%	Ori	Opt	%
impfact	896	500	-44.2%	222036	87322	-60.7%
impfibo	899	489	-45.6%	294173	108589	-63.1%
recfact	952	533	-44.0%	248648	123569	-50.3%
recfibo	994	576	-42.1%	784807	353445	-55.0%

Table IV
OPTIMIZATION SCORE

Note that the executable patching step may insert some jump instructions to preserve the address of each instruction, and the number of optimized instances may be larger if you count the number of instructions really executed from the patched executable program.

As the number of instructions are decreased and decoding routines are eliminated from VM main block and handlers, we can expect for the analysis of optimized program to be easier than analysis the original obfuscated program.

IX. Related Work

G. Wroblewski systemized and described the traditional obfuscation techniques for executable programs, including dead code insertion, reordering of control flow, and data obfuscation in his PhD thesis [15]. C. Linn and S. Debray [16] introduced an obfuscation technique to improve resistance to static disassembly.

S. Udupa et al. [17] suggested an automatic deobfuscation technique. This technique is, however, a static approach, so it is totally different from our approach.

R. Rolles [1], M. Sharif [2], and N. Falliere et al. [18] formerly discussed the reversing techniques for the virtualization obfuscated programs. However, as previously discussed, these techniques have their own assumptions on the obfuscation techniques, so they are inappropriate for the modified obfuscation techniques that are not fit to the assumptions. On the contrary, our work assumes nothing about the obfuscation applied to the target program, it is applicable for any obfuscation technique.

K. Coogan et al. [19] tried to deobfuscate the softwares obfuscated by virtualization obfuscation techniques with dynamic approach. Their work finds the instructions relevant to the external routine calls and slice the program to contain only the relevant instructions. Their work seems to be similar to our trace pruning optimization with milestone establishment. However, their ultimate goal is to present an fully automatic program slicer for deobfuscation, but our goal is to provide a sound and general dynamic analysis tool.

X. Conclusion

In this paper, we presented the dynamic program analysis tool Trudio, which has three analysis purposes: structure analysis, semantic analysis, and optimization. Trudio has the implementation of several algorithms for these functionalities. This paper describes the details of the algorithms implemented in Trudio. Furthermore, we verified the effectiveness of this approach by experiments.

Using the analysis methods provided by Trudio, you can reveal the structure and semantic of obfuscated programs and transform an obfuscated program into more analyzable shape. It is expected for the security experts to save time for analyzing the obfuscated malwares.

However, our work has some weaknesses. First of all, we have not proved the safety of the optimization techniques. In other word, an optimized program is not guaranteed to be able to imitate the execution paths appeared in the run trace. The proof of the safety of each optimization technique must be studied further.

Secondly, dynamic approach has its inherent limitation: coverage problem. Dynamic analysis is based on the run traces of the target program, and the algorithms assume the program only runs the execution paths appeared in the run traces. Hence, the analysis results acquired from dynamic approach may not be true for the other execution paths. To complement the limitation of the dynamic approach, we may need to use multiple run traces extracted from several executions of the same program.

References

[1] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX conference on Offensive technologies*, ser. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1.

[2] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 94–109.

[3] VMProtect Software, "Vmprotect software protection," http://vmpsoft.com/, 2011.

[4] R. BenzmÃijller and S. Berkenkopf, "G data malware report, half-yearly report january - june 2011," G Data SecurityLab, Tech. Rep., 2011.

[5] http://trudio.googlecode.com/files/Trudio.zip, 2011.

[6] Hex-Rays SA, "IDA Pro," http://www.hex-rays.com/products/ida/index.shtml, 2011.

[7] O. Yuschuk, "OllyDbg," http://ollydbg.de/, 2011.

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

[9] K. Lawton, "bochs: The open source ia-32 emulation project," http://bochs.sourceforge.net/, 2011.

[10] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 190–200.

[11] http://trudio.googlecode.com/files/tracing.zip, 2011.

[12] G. Dabah, "diStorm3," http://code.google.com/p/distorm/, 2011.

[13] J. H. Simon Tatham, "The netwide assembler," http://www.nasm.us/, 2011.

[14] http://trudio.googlecode.com/files/experiments.zip, 2011.

[15] G. Wroblewski, "General method of program code obfuscation," Ph.D. dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[16] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299.

[17] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," *Reverse Engineering, Working Conference on*, vol. 0, pp. 45–54, 2005.

[18] N. Falliere, P. Fitzgerald, and E. Chien, "Inside the jaws of trojan.clampi," Symantec Corp., Tech. Rep., 2009.

[19] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 275–284.