Identifying Input-Dependent Jumps from Obfuscated Execution using Dynamic Data Flow Graphs

Joonhyung Hwang
Korea Advanced Institute of Science and Technology
envia@envia.pe.kr

Taisook Han
Korea Advanced Institute of Science and Technology
han@cs.kaist.ac.kr

ABSTRACT

A method to identify input-dependent jumps from the execution of obfuscated machine code is presented. Input-dependent jumps, which are defined as jumps whose target addresses can be changed depending on the input, correspond to decision points in program execution. By investigating how a target address is calculated, it is possible to pinpoint the triggering conditions of a given behavior, and new execution paths can be discovered by finding input values that change the target address. Obfuscators hinder such analysis by inserting numerous artificial jumps that use opaque predicates with constant values into the code. One important obfuscation approach is virtualization-obfuscation, in which entire blocs of control flow information are replaced with bytecode interpreter code. Using the fact that the semantics of the original program must be preserved under obfuscation, we propose an obfuscation mitigation approach that exploits the relationship between the original and obfuscated executions using dynamic data flow graphs that represent output computation using concrete and symbolic information. These graphs are generated from execution traces that are recorded using dynamic binary instrumentation and simplified using pattern-based rules based on algebraic identities and the general properties of well-behaved programs. To identify input-dependent jumps, a dynamic data flow graph is generated and simplified for each write access to the program counter; if the node for the target address is reachable from a node for an input value in the resulting graph, the jump is input-dependent. Experimental application of the proposed approach to code treated with various obfuscators successfully revealed the relationship between input-dependent jumps in the original and obfuscated executions, confirming that information obtained from dynamic data flow graphs is useful in understanding branch conditions.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

input-dependent jump; dynamic data flow graph; deobfuscation; reverse engineering

SSPREW-8, December 3–4, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6096-8/18/12…$15.00
https://doi.org/10.1145/3289239.3291460

ACM Reference Format:
Joonhyung Hwang and Taisook Han. 2018. Identifying Input-Dependent Jumps from Obfuscated Execution using Dynamic Data Flow Graphs. In *Software Security, Protection, and Reverse Engineering Workshop (SSPREW-8), December 3–4, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3289239.3291460

1 INTRODUCTION

Obfuscation [6, 20] is a technique for transforming program code to make it more difficult to follow. An obfuscator hides the structure of an original program while leaving its observable behavior unchanged, enabling the use of the obfuscated program in place of the original program. Although the obfuscation approach is useful for protecting secrets contained in program code and preventing unwanted modifications, obfuscation is also used in malicious software to hinder analysis and detection. Without proper tools and skills, it is easy to waste effort on working with maliciously obfuscated code with irrelevant information and missing important behavior. It is therefore useful to develop tools that can mitigate the effects of obfuscation and identify original program behavior to facilitate the analysis of potentially malicious obfuscated executables.

An important approach to gaining a better understanding of the structure and behavior of a program is identifying the branch conditions of jumps in program execution. Conditions that trigger specific behaviors can be used to discover input values for new execution paths through the use of methods such as symbolic execution [2]. Unfortunately, the control flow information of an obfuscated program will not be related to that of the original program. A powerful example of control flow transformation is virtualization-obfuscation [8, 11, 18, 19, 26], which is also known as emulation-based obfuscation [21] or table interpretation [6]. Virtualization-obfuscation transforms an original program into a bytecode program to be executed by an interpreter. Although this results in a program with a control flow information that is related solely to the structure of the interpreter, jumps in the execution of the obfuscated program will correspond to jumps in execution of the original program because the observable behaviors of the obfuscated and original programs should be same.

We propose a method for using input-dependent jumps to identify the relationship between the execution of an obfuscated program and that of its original. Here, we define an input-dependent jump as a jump whose target address can be changed depending on the input. In both an obfuscated and original program, such jumps will be related to decision points in program execution; by contrast, jumps with a predetermined target address will likely be related to obfuscation constructs.

Identification of input-dependent jumps in obfuscated code is not trivial because execution is obfuscated by the use of constants

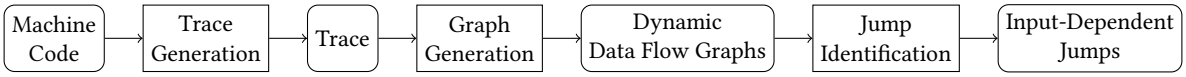

Figure 1: Overall process overview.

and algebraic identities. If branch conditions are generated without proper simplification, the results are usually too complex to understand, both for humans and machines. For example, Yadegari et al. [25] reports cases where naive application of symbolic execution to obfuscated code fails or times out even if the original code is very simple. After observing the execution of obfuscated binaries, we realized that such programs access large numbers of constant values embedded in the binaries. Computation using constants can be simplified because its results are already known. On the other hand, computation that depends on input values will be related to the semantics of the original program.

To identify input-dependent jumps, the proposed method generates dynamic data flow graphs from dynamically generated traces. These graphs, which capture the computation of the target addresses of jumps, are simplified to eliminate the effects of obfuscation using graph rewriting rules based on algebraic identities and the general properties of well-behaved programs. A jump will then be identified as input-dependent if the node for the target address is reachable from a node for an input value in the simplified graph for the jump. Figure 1 shows the overall process.

Testing of the proposed identification method against the commercial obfuscators Code Virtualizer [15], Themida [16], and VMProtect [22] revealed that the number of input-dependent jumps in the obfuscated executions did not differ significantly from the number of jumps in the original execution. It suggests that the proposed method successfully identified most of the jumps introduced by the obfuscators as non-input-dependent. Experiments with Tigress challenges [5] shows the efficacy of our method. Many of the simplified branch conditions are able to be understood by human with reasonable effort. In this paper, we focus on the assessment of the proposed method against the virtualization-obfuscation tools for x86 and x64 platforms. Our approach, however, can be applied in other situations because no special assumptions are made regarding the obfuscation approach; indeed, the tested obfuscators use combinations of various obfuscation techniques.

The rest of this paper is organized as follows. Section 2 provides motivating examples. Section 3 introduces the dynamic data flow graphs used by the proposed approach. Section 4 describes how the traces and graphs are generated, while Section 5 describes how the graphs are simplified and how input-dependent jumps are identified. Sections 6 and 7 discuss the experimental results and review related work, respectively, and, finally, Section 8 concludes the paper.

2 MOTIVATING EXAMPLES

In this paper, we use a factorial program (Figure 2) and a bubble sort program (Figure 3) as examples. Although these are relatively simple programs, they are difficult to analyze following obfuscation, which in this case is done using Code Virtualizer 1.3.9.10 and 2.2.2.0, Themida 2.4.6.0, and VMProtect 2.13.6 and 3.1.2.830.

Approaches to automated program analysis can be classified into two categories: static and dynamic. In static analysis, program

```
fac = 1;
for (i = 1; i <= n; i++)
{
        fac *= i;
}
```

Figure 2: Factorial program.

```
for (i = 1; i < n; i++)
{
        for (j = i; j > 0; j--)
        {
                if (x[j] < x[j-1])
                {
                        t = x[j];
                        x[j] = x[j-1];
                        x[j-1] = t;
                }
        }
}
```

Figure 3: Bubble sort program.

```
941911 @ 0x8b1030 : mov dword ptr [ebp-0x8], 0x1
941912 @ 0x8b1037 : mov dword ptr [ebp-0x4], 0x1
941913 @ 0x8b103e : jmp 0x8b1049
941914 @ 0x8b1049 : mov ecx, dword ptr [ebp-0x4]
941915 @ 0x8b104c : cmp ecx, dword ptr [ebp-0xc]
941916 @ 0x8b104f : jnle 0x8b105d
```

Figure 4: Execution of factorial of zero.

behavior is investigated without executing the program. It is easy to lose precision under static analysis because the approach considers all possible execution paths together. Most obfuscators in practice are strong enough to defeat static analysis. Although there is some work on static analysis of obfuscated machine code [11], it requires understanding of the obfuscation structure, which usually requires dynamic or manual analysis.

Here, we apply a dynamic approach that uses traces from program execution. Under dynamic analysis, precise results can be obtained using values from actual execution of the program. One limitation of this approach is that only executed code is analyzed, and therefore a malicious program can be classified as benign if the malicious part of the program is not executed. This problem can be solved by increasing coverage through the discovery of new paths.

We first look at the unobfuscated execution of the factorial program shown in Figure 2 with the variable `n` set to zero. Because `i <= n` is not satisfied, the body of the loop is not executed; only by setting the input variable `n` to a value greater than or equal to one will the body be executed. Figure 4 shows the information

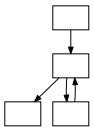

Figure 5: Control flow graph of factorial program (original).

extracted from a trace of the execution. The first, second, and third columns show the identification numbers of instruction execution instances, the addresses of the instructions, and the instructions in assembly language, respectively. Variables `i` and `fac` occupy memory locations pointed by `ebp-0x4` and `ebp-0x8`, respectively, while the input variable `n` is set to zero and stored in a memory location pointed by `ebp-0xc`. After initialization of variables `fac` and `i`, `i` and `n` are compared, the conditional branch is taken, and the loop body is skipped.

We assume that the original binary executables are usual application programs compiled using a typical compiler, in which case any unusual behavior observed in the execution of obfuscated programs can be taken to reflect code inserted by obfuscators.

We also assume that only the essential parts of the program are obfuscated, and program execution starts from the unobfuscated area before entering the obfuscated area. Applying expensive transformation such as virtualization-obfuscation to entire programs generally produces undesirable results because it significantly slows down the execution. Identification of obfuscated areas is an interesting research subject. For example, Xu et al. [23] covers a method to detect boundaries of obfuscating virtual machines. However, the proposed approach simply inserts markers into samples to identify the beginning and end of the obfuscated area because it is not the topic of this work. Obfuscation boundaries of Tigress challenges are determined by heuristics.

In the obfuscation process, various control and data transformations are performed to increase the analysis workload through the insertion of numerous instructions that do not change the final results and the conversion of simple computations to more complex variants using the laws of arithmetic and logic. Obfuscation also alters program control flow information to mislead the analysis process. For example, virtualization-obfuscation converts an original program into a bytecode program and its interpreter. Figures 5 and 6 show the control flow graphs of the original factorial program in Figure 2 and the program obfuscated by Code Virtualizer 1, respectively. It is seen that the two graphs are different, with only the structure of the interpreter visible in Figure 6. In this case, the structure of the virtual machine and interpreter is determined by the obfuscator. As a result, automated analysis of obfuscated code is challenging even if a dynamic approach is used.

It is possible to use input-dependent jumps to find the relationship between the obfuscated and original execution. A jump that depends on an input value in the original execution must have a corresponding jump in the obfuscated execution because the semantics of the original program are preserved in the obfuscation process. Thus, an analysis of the input-dependent jumps in the obfuscated execution can reveal decisions made in the original execution.

Table 1: Memory location statistics.

Obfuscator	Total	Main	Constant
Original	12	0	0
Code Virtualizer 1	749	685	640
Code Virtualizer 2	1722	1626	876
Themida 2	1184	1076	932
VMProtect 2	1611	1355	1355
VMProtect 3	2123	1995	1995

To use input-dependent jumps, they must be identified from the program execution. In Figure 4, `jmp 0x8b1049` of execution instance 941913 is not input-dependent because its target address is predetermined. By contrast, `jnle 0x8b105d` of execution instance 941916 is input-dependent because its target address is determined based on the result of comparing the variable `i` with the input variable `n`. The jump will be considered non-input-dependent if the value of `n` is predetermined.

In this work, input values are required to satisfy two conditions. First, they are used in the code of interest but defined outside. Secondly, they can be changed by the user. The second condition is checked with a negative list approach, in which a set of conditions to be a non-input value are provided; for example, a constant value is a non-input value. In this scheme, a value is considered as an input value if it does not satisfy any of the conditions. Although most previous work [8, 11, 19, 26] requires manual identification of input values, our experiment indicates that such effort is not always necessary.

Identification of input-dependent jumps is complicated under obfuscated execution, partly because obfuscated programs use embedded constant values. Table 1 lists statistics on memory location usage from the execution of the factorial program with the input variable `n` set to two. The first and second columns show, respectively, the obfuscation tools and the total sizes in bytes of the memory locations accessed during their execution. Some of the locations are in the stack, while others are in the memory regions allocated for the main executable image. The third column shows the sizes in bytes of the memory locations accessed in the main executable region. Some of these memory locations contain constant values inserted by the obfuscator. The values of the constants' locations are taken directly from the executable image, and there are no write operations to these locations. The fourth column shows the sizes in bytes of the constants' memory locations. It is seen that the obfuscated programs use large numbers of constants.

To mitigate the effects of obfuscation, the results of computation using multiple constants can be simplified to a single constant. Obfuscated computation can also be simplified using the laws of arithmetic and logic. Such simplification, however, requires the use of a data structure to represent the computational process; as discussed in the following section, such data structures can be generated using dynamic data flow graphs.

3 DYNAMIC DATA FLOW GRAPHS

The proposed approach uses dynamic data flow graphs to represent how output values are computed from input values. Note that our

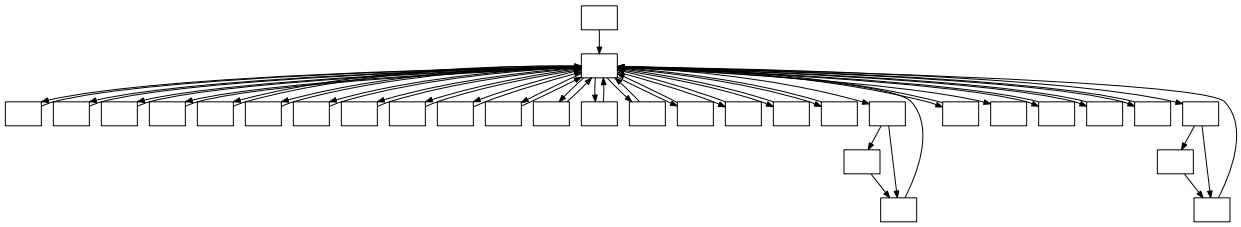

Figure 6: Control flow graph of factorial program (obfuscated by Code Virtualizer 1).

notion of a flow graph is different from some literature [10], where a flow graph is defined as a directed graph with a node that can reach every node in the graph.

A dynamic data flow graph is a directed acyclic multigraph, whose edges are labeled by numbers. It is defined as a 5-tuple (N, E, def, use, pos), where N and E are disjoint sets for nodes and edges and $def\colon E \to N$, $use\colon E \to N$, and $pos\colon E \to \mathbb{N}$ are functions on E. The nodes correspond to values of expressions calculated during execution. Each access to memory or a register is considered separately because a given location can be used for completely unrelated purposes under obfuscated execution. If a node $n \in N$ represents a result of an operation, its operands are represented with other nodes. For each operand, there is an edge $e \in E$ directed from the operand node to the operation node. In this case, we call the operand node as the definition node of e denoted by $def(e)$, and the operation node as the use node of e denoted by $use(e)$. If the operation of node n takes i operands, there should be i edges directed to n. The map pos labels these edges by assigning position numbers from 1 to i according to the order of the operands. There should be no cycle in a dynamic data flow graph, because it is not possible to use a value before definition. A dynamic data flow graph can be generated for a single output or multiple output values; additionally, a dynamic data flow graph for a single output value can be extracted as a subgraph from a graph for multiple output values.

The dynamic data flow graph nodes are designed to provide both symbolic and concrete information on the computational process. A node is defined as a 3-tuple $(id, type, info)$, where id is a node identification number, $type$ is a node type corresponding to the type of expression it represents, and $info$ is information for type-dependent analysis. The concrete value of each expression and its size in bits are stored for all nodes. They are used when a node for a symbolic variable or a result of an operation is converted to a node for a constant. They are also used to check the consistency of the graph.

The types of nodes and their corresponding information are listed in Table 2. The nodes of a dynamic data flow graph can be categorized into the following types: Access, Integer, Operation, Symbol, and Variable. An Access node represents the read or write access of an execution instance of an instruction to a target. In this case, the address of the target and the identification number of the execution instance are stored in the node along with the mode of access, which is `r` for a read or `w` for a write. An Integer node represents a constant. An Operation node represents an operation

Table 2: Types of nodes.

Type	Information
Access	address, execution, mode, size, value
Integer	size, value
Operation	size, value
Symbol	name, size, value
Variable	address, size, value

with operands. The operands of an operation are independently represented by separate nodes and connected by edges to the Operation node. A Symbol node represents a value that is determined by the environment. For example, the start address of a section in memory, which is determined by the loader when the image of the executable is loaded, will be represented by a specific Symbol node. Each Symbol node is given a name, which is stored in the node. A Variable node represents a value given from outside. For example, a command line argument will be represented by a specific Variable node. The value of a Variable node is read from an address in the memory or a register that is stored in the Variable node.

Access type has two special subtypes: Input and Output. Input subtype is used for values from system-dependent operations. For example, an output value of RDTSC instruction, which reads the time-stamp counter of the processor, is represented by an Input node. Output subtype is used to mark output values. For example, a target address of a jump is represented with an Output node in our experiments.

Operation type has various subtypes that are categorized according to their semantics as in Table 3. Operation node subtypes are based on the machine instruction types but are modified to facilitate analysis. Nodes for associative operations can have arbitrary numbers of operands, although actual machine instructions have limited numbers of operands. If an instruction has multiple output values, each is separately represented. Furthermore, the computation of an output value is represented separately from the computation of the status register. A node for a conditional jump has three operands. The first is a value from the status register. If the condition is satisfied for the first operand, the target address is set to the value from the second operand; otherwise, the target address is set to the value from the third operand, which contains the address of the next instruction of the jump instruction. Note that machine instructions for conditional jumps have only one target; if

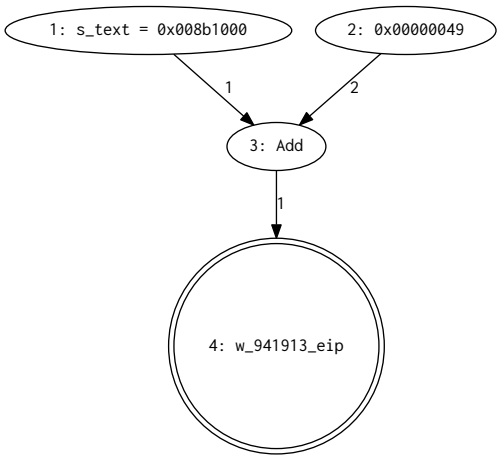

Figure 7: Dynamic data flow graph for `jmp 0x8b1049`.

the condition is not satisfied, the next instruction is executed. Indirect memory access is expressed by IndirectRead and IndrectWrite nodes, each of which will have two operands—one for the memory address and the other for the value at the address. Merge and Split nodes represent combinations of operations of different sizes. For example, if a program first writes to the EAX register and then to the AL register, which is the lower half of the EAX register, the final value of the EAX register will be a combination of the results of the two operations. The value of a Merge node is the concatenation of the values of its operands, while that of a Split node is computed by selecting bits from the value of its operand. The positions at which a selection is started and stopped are stored in a Split node.

The proposed method uses an address scheme that gives an address to each register, with a 64-bit flat memory used for both registers and memory. Register addresses are taken from a set of addresses not used during the execution. Currently, the non-canonical addresses of x64 [1], which are not to be used by any 32-bit or 64-bit application, are used for this purpose. Registers of different threads use different addresses. If possible, a memory address is expressed as the sum of a base address and an offset to allow for consistent expressions over multiple executions regardless of the memory allocation including relocation.

Generated graphs can be visualized or expressed in prefix notation. Figure 7 shows an example of visualization of the dynamic data flow graph for `jmp 0x8b1049` in Figure 4. The target address is converted to a sum of a section address and an offset. The numbers before the colons in the nodes are the node identification numbers. Node 1 is a Symbol node that contains the address of the `.text` section, which is `0x008b1000`. Node 2 is an Integer node whose value is `0x00000049`. Node 3 is an Operation node for addition whose operands are Nodes 2 and 3. Node 4 is an Access node for the output, which is a write access at execution instance 941913 to the EIP register. In prefix notation, this is the output `w_941913_eip` set to `(Add s_text 0x00000049)`.

4 TRACE AND GRAPH GENERATION

The trace generation tool is written in C++ using Pin [13], a dynamic binary instrumentation framework from Intel. Traces are written in extensible markup language (XML). Each trace has a `trace` entity containing a sequence of `img`, `ins`, `execution`, and `access` entities. Each `img` entity has information on the image of an executable and contains a `raw` entity and several `rgn` entities. Executable images in the memory can differ from executable images on the disk. Some addresses within the images are changed during relocation. In Windows, the security cookies in the images are set to new values. Images are recorded both prior to and following loading for completeness. The main executable prior to loading is stored in a `raw` entity. A region of an executable is a collection of sections that occupies a contiguous area within the memory. There can be multiple regions for an executable. Each region of the post-loading main executable is stored in a `rgn` entity. An `ins` entity has information about an instruction, which can be executed multiple times. An `execution` entity has information about an execution instance of an instruction. Each execution instance is accompanied by access to registers and memory locations. An `access` entity has information about such access.

It is worth noting that research has been conducted on anti-instrumentation techniques and their countermeasures, e.g., Polino et al. [17]. However, because it is outside the scope of this work, we will not discuss this issue further.

A generated trace can be stored in a database. A table is created for each entity in the XML. Columns of each table are taken from the attributes of the corresponding entity.

The proposed method uses a tool that constructs and manipulates dynamic data flow graphs in Python. Each graph is implemented as a dictionary that maps a node identification number to a tuple of the node and a tuple of identification numbers of the nodes of its direct predecessors. For faster access, a hash value is assigned to each graph node and each value in the dictionary for the graph. The graphs can be stored in a JavaScript object notation (JSON) file.

A dynamic data flow graph is generated backwards from each write access to the program counter. The program counter is written by call and return instructions and unconditional and conditional jumps. For each conditional jump, the target address and value of the status register are both investigated; otherwise, only the target address is investigated.

Graph generation begins with a graph with Access nodes as output. Each jump is represented by a node for write access to the program counter. The graph grows by adding predecessors to the Access nodes. If an Access node corresponds to write access, nodes for read access that affect the computation based on the type of operation are added. In this case, write and read access are performed by the same execution instance. If an Access node corresponds to read access, nodes for the latest write access are added to the target addresses from the previous execution instances in the obfuscated part of the execution. If there are multiple such nodes, they are merged using an Operation node for merge operation. If there are no such nodes, a Variable node is added and it is concluded that the value of the read access is from the outside of the obfuscated part of the execution. Nodes are added until there is no Access node without a predecessor. The process terminates because the length

Table 3: Subtypes of Operation type.

Category	Subtypes
Arithmetic and Logic	Add, And, Bswap, …, Cmpxchg, …, Mul, Neg, Not, Or, …, Rol, Ror, …, Shl, Shr, Sub, Xor
Flags	ADD_FLAGS, …, CMP_FLAGS, CMPXCHG_FLAGS, …, TEST_FLAGS, …
Conditional Jumps	JB, JBE, JL, JLE, JNB, JNBE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JS, JZ
Indirection	IndirectRead, IndirectWrite
Merge and Split	Merge, Split

of the trace is finite, with a time complexity roughly linear to the length of the trace.

5 GRAPH SIMPLIFICATION AND INPUT-DEPENDENT JUMP IDENTIFICATION

To remove the effects of obfuscation, the proposed method simplifies graphs by applying simplification rules until no change can be made. Each node is tested with its predecessors to check whether it can satisfy the conditions of any simplification rules; if a rule can be applied, the corresponding action is performed. The simplification rules need to be adjusted depending on the situation and the desired granularity. It must be ensured that the simplification process eventually terminates.

Our simplification process begins with identification of non-input values. We use two conditions based on general properties of well-behaved programs. First, if a value of a Variable node is taken from a value embedded in the code area, it is considered as constant. Then the node is replaced by an Integer node. If the address of a Variable node belongs to the range of the main executable, all access to that address is investigated. Secondly, the value of the trap flag of the status register is simplified to a constant because the trap flag is not one of the seven flags used by applications.

Computations using constants are simplified. If all operands of an Operation node are constants, the result of the operation is considered to be constant and the node is replaced by an Integer node.

Movement of data is simplified. If a value is moved multiple times, the movements are simplified to a single movement. Indirect memory access is simplified to a direct memory access if its target address is fixed. If a Split node has a Merge node as an operand and only one operand of the Merge node corresponds to the position and size of the Split node, the Split node is replaced by the node of that operand.

Operations are simplified depending on their semantics. Figure 8 shows the rules for simplification using prefix notation. If an operation is commutative and associative, its operands are collected together. If such an operation has multiple constant operands, they are simplified to a single operand. Like terms in the operands of an addition are combined. For example, (Add x x x) becomes (Mul x 3); because addition is easier to simplify than subtraction, subtracting x from y is represented as adding the negation of x to y.

Replacing nodes with simpler nodes makes some nodes unusable; nodes that do not reach an output node are removed.

A jump is identified as input-dependent if its simplified graph has an Input node that represents a result of a system-dependent

∗ Rules using associativity:
- (Add x … (Add y …)) → (Add x … y …)
- (And x … (And y …)) → (And x … y …)
- (Mul x … (Mul y …)) → (Mul x … y …)
- (Or x … (Or y …)) → (Or x … y …)
- (Xor x … (Xor y …)) → (Xor x … y …)

∗ Rules using identity:
- (Add x (Neg x)) → 0
- (Add x 0) → x
- (And x (Not x)) → 0
- (And x -1) → x
- (And x 0) → 0
- (And x x) → x
- (Mul x 0) → 0
- (Mul x 1) → x
- (Neg (Neg x)) → x
- (Not (Not x)) → x
- (Or x (Not x)) → -1
- (Or x -1) → -1
- (Or x 0) → x
- (Or x x) → x
- (Xor x (Not x)) → -1
- (Xor x -1) → (Not x)
- (Xor x 0) → x
- (Xor x x) → 0

Figure 8: Sample rules for simplification.

operation or a Variable node that represents a value from outside of the obfuscated area. If an input-dependent jump is found, all access to flag operation results in the computation of the jump is considered as used. Values of used access are considered to be constants in the analysis of later jumps. Jumps computed by applying non-input-dependent computation to a previous input-dependent jump are not identified as input-dependent.

6 EXPERIMENTAL RESULTS

We generated and simplified dynamic data flow graphs to identify input-dependent jumps for factorial and bubble sort programs treated with the commercial obfuscators and Tigress challenges. Traces of new samples and Tigress challenges are generated, respectively, on x86 Windows and x64 Linux machines. The traces were analyzed on a Linux system with dual Intel Xeon E5-2640 v3 processors and 128GB of memory.

Tables 4–6 show the results. In the tables, the first column shows the names of the obfuscators or samples used. The second column shows the input values given to the program through the command line. The third columns show the number of instances of executed instructions in the obfuscated part of the execution, while the fourth columns display the number of total jumps, i.e., the number of write accesses to the program counter. The fifth and sixth columns show, respectively, the number of input-dependent jumps before and after simplification. The seventh columns show the time spent to

generate the dynamic data flow graph for all target addresses from the trace stored in the database in seconds and, finally, the eighth columns show the time spent to simplify the dynamic data flow graph and identify the input-dependent jumps in seconds.

For the factorial programs, the number of input-dependent jumps exceeds the input `n` by one because the counter `i` is compared to `n` before each iteration of the loop. For the bubble sort programs with a single input value, only one jump is input-dependent because only one comparison is performed to check the counter `i`. For the bubble sort programs with three input values, there are six input-dependent jumps because three comparisons each are carried out to check the counter and compare the values of the array, respectively. There is no input-dependent jump for the inner loop `for (j = i; j > 0; j--)` because both `i` and `j` are initialized and computed in the obfuscated part of the execution.

It is seen that the number of input-dependent jumps taken by the obfuscated programs are significantly decreased after simplification. For three of the obfuscators, all input-dependent jumps were identified with precise information about the input; however, the simplification did not work as well for the remaining obfuscators, but it is expected that the use of better simplification rules would improve these results. The programs obfuscated by Themida 2 performed initialization before executing code in the obfuscated area; some values from this initialization were incorrectly identified as input values, and excluding them from the input would also improve the results.

Although the process is not straightforward, simplified dynamic data flow graphs of input-dependent jumps can be used to identify jump conditions with reasonable effort. Input values for new paths can be generated using the identified jump conditions. Without simplification, identification of jump conditions is nearly impossible.

Figure 9 shows a simplified dynamic data flow graph for `jnle 0x8b105d` in Figure 4. The jump corresponds to `i <= n` in Figure 2, where the variable `i` is set to one and the variable `n` has the input value. The jump is performed by the jump if not less or equal (JNLE) instruction following the compare (CMP) instruction. The branch is taken if the input is less than one; because in this case the value of the input is zero, the branch is taken and the loop body is skipped. A new path can be discovered by using an input value greater than or equal to one.

Figure 10 shows a dynamic data flow graph of the input-dependent jump, corresponding to the jump of the original program in Figure 9, from the execution of the factorial program obfuscated by Code Virtualizer 1 with input zero. In the original program, the JNLE instruction performs a jump to the target if zero flag (ZF, `0x40`) is 0 and sign flag (SF, `0x80`) and overflow flag (OF, `0x800`) are the same. In the obfuscated program, the target address is still computed using the flags from the CMP instruction but the semantics of the JNLE instruction are emulated using other operations. Node 4 contains the flags computed by comparing the input variable `n` with the constant one. Flags of interest are extracted by the AND operations. SF in Node 6 and OF in Node 10 are compared by the XOR operation at Node 13. The least significant bits of Nodes 14, 15, and 24 will be 1 if SF and OF are the same. ZF in Node 17 is manipulated so that the second least significant bit of Node 24 will be 1 if ZF is 0. The jump to the target address is taken if the value of Node 24 is 3, which occurs if ZF = 0 and SF = OF. Although following this graph is by no means trivial, the simplification process makes it significantly easier to understand the execution. The simplified graph has 34 nodes and 40 edges, as compared to 15,863 nodes and 19,717 prior to simplification.

Figure 11 shows the corresponding graph from VMProtect 3. The graph has 59 nodes and 70 edges, as compared to the 15,064 nodes and 18,223 edges on the pre-simplified graph. In this case, the computation of the flags by the CMP instruction is split into two parts. SF and ZF are computed using the ADD operation at Node 3, while OF is computed using the ADD_FLAGS operation at Node 10; the three flags are then merged by the ADD operation at Node 12 and evaluated to see whether the condition of the JNLE instruction is met. The value of Node 34 is `0x40` if ZF = 0 and SF = OF, and the target address is computed using the value of Node 34 and other non-input-dependent values.

7 RELATED WORK

Collberg et al. [6] is a classic survey of obfuscation and deobfuscation techniques in which obfuscation is defined as transformation of a source program to a target program provided that the source and target programs have the same observable behavior. Virtualization-obfuscation is mentioned as one of the most effective obfuscation transformations, while identification and evaluation of opaque variables and predicates is identified as the most difficult part of deobfuscation.

Schrittwieser et al. [20] is a relatively recent survey that provides a classification of analysis scenarios based on analysis methods and goals. In their work, analysis methods are classified into four categories: pattern matching, static analysis, dynamic analysis, and human analysis. Analysis goals are similarly classified as: locating data, locating code, extracting code, and understanding code. According to their categorization, our work can be classified as locating code through dynamic analysis.

Obfuscators are used to make programs difficult to understand. It is worth noting that currently used obfuscators generally do not have a theoretical proof of effectiveness. In Barak et al. [3], obfuscation is required to have the black box property: any analysis result from an obfuscated program can be obtained from oracle access to its original program. Although obfuscators with the black box property have interesting cryptographic applications, it is shown that they do not exist. Although there has been research on weaker notions of obfuscation with promising results, e.g., Garg et al. [9], they have yet to be used in practice. Accordingly, to date the effectiveness of obfuscators has been evaluated through empirical means.

In Coogan et al. [8], the analysis of virtualization-obfuscation is classified into two categories: outside-in approaches and inside-out approaches. Under an outside-in approach, the structure of the virtual machine is first analyzed and the result is used to understand an obfuscated program. Under an inside-out approach, an obfuscated program is directly analyzed without regard to the structure of the virtual machine. The method proposed in this paper takes an inside-out approach, which, because of its generality, makes it effective against various obfuscation techniques.

Table 4: Input-dependent jumps from factorial program.

Obfuscator	Input	Length	Total Jumps	Without Simplification	With Simplification	Generation Time (s)	Simplification Time (s)
Original	0	6	2	1	1	0.03	0.01
	1	16	4	2	2	0.07	0.01
	2	26	6	3	3	0.10	0.02
	10	106	22	11	11	0.31	0.05
Code Virtualizer 1	0	8178	1762	165	1	28	7
	1	18674	4061	366	2	73	27
	2	29170	6360	567	3	110	57
	10	113138	24752	2175	11	437	891
Code Virtualizer 2	0	82909	1647	1425	1	114	32
	1	127961	2531	2272	2	164	59
	2	173029	3416	3120	3	214	91
	10	533581	10492	9904	11	640	642
Themida 2	0	65311	1276	1265	117	71	40
	1	108583	2137	2122	194	135	87
	2	151874	2999	2980	271	183	157
	10	498202	9895	9844	887	610	1471
VMProtect 2	0	37773	5548	590	1	67	510
	1	72090	10613	1123	2	133	1149
	2	106345	15678	1656	3	206	1792
	10	380721	56198	5920	11	971	8723
VMProtect 3	0	13311	1275	369	1	19	4
	1	28443	2826	789	2	43	11
	2	43575	4377	1209	3	69	17
	10	164631	16785	4569	11	262	121

Table 5: Input-dependent jumps from bubble sort program.

Obfuscator	Input	Length	Total Jumps	Without Simplification	With Simplification	Generation Time (s)	Simplification Time (s)
Original	1 2 3	68	19	11	6	0.26	0.06
	2	5	2	1	1	0.02	0.01
	3 2 1	110	19	11	6	0.37	0.07
Code Virtualizer 1	1 2 3	102598	19255	1291	6	219	276
	2	13233	2464	166	1	27	8
	3 2 1	177538	33502	2200	6	397	747
Code Virtualizer 2	1 2 3	324025	6896	6453	6	605	540
	2	75656	1552	1343	1	163	72
	3 2 1	571369	12062	11355	6	1060	1400
Themida 2	1 2 3	310336	6370	6197	533	358	353
	2	60729	1194	1161	110	73	27
	3 2 1	557585	11350	11051	968	658	1072
VMProtect 2	1 2 3	263724	30344	10953	22	585	463
	2	34505	3713	1330	3	52	14
	3 2 1	326556	35213	12726	40	766	761
VMProtect 3	1 2 3	115449	14334	3456	6	194	79
	2	11649	1486	333	1	18	4
	3 2 1	132483	16635	3987	6	226	88

Table 6: Input-dependent jumps from Tigress challenges (Linux-x86_64).

Sample	Input	Length	Total Jumps	Without Simplification	With Simplification	Generation Time (s)	Simplification Time (s)
0000/challenge-0	0	39742	2872	1999	0	1575	28
0000/challenge-1	0	86212	11426	8142	1	10505	94
0000/challenge-2	0	30801	10409	9816	3	1766	24
0000/challenge-3	0	43534	3421	2488	0	1908	32
0000/challenge-4	0	17665	2725	1366	1	322	9
0001/challenge-0	0	12253	411	228	188	51	8
0001/challenge-1	0	13975	455	247	208	72	13
0001/challenge-2	0	31431	1119	610	609	248	71
0001/challenge-3	0	24415	804	435	338	146	22
0001/challenge-4	0	17080	567	304	303	81	17
0003/challenge-0	0	437698	24623	13331	2	11783	809
0003/challenge-1	0	113812	6843	3721	2730	1204	247
0003/challenge-2	0	89682	5181	2921	991	672	82
0003/challenge-3	0	50412	3579	1839	1	391	20
0003/challenge-4	0	216907	17952	9486	7891	4025	1452

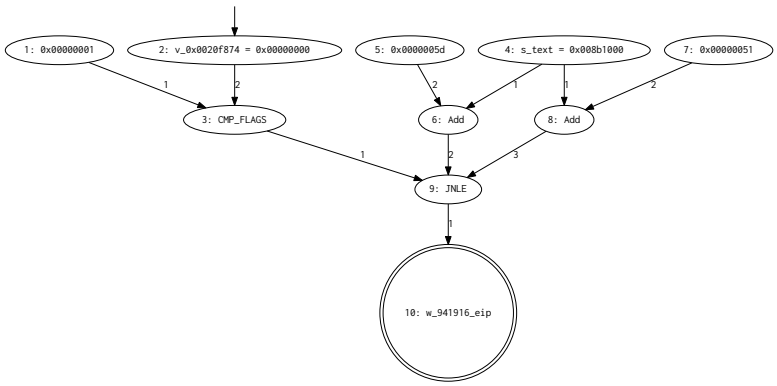

Figure 9: Simplified dynamic data flow graph for `i <= n` (original).

Sharif et al. [21] takes an outside-in approach to dynamic analysis on traces in which variables are identified using forward and backward dynamic data flow analysis. After identifying the virtual program counter from variables, each part of the bytecode interpreter is identified using dynamic taint analysis. After extracting the syntax and semantics of the bytecode instructions, a control flow graph is generated for the bytecode.

Rolles [18] takes another outside-in approach using human analysis in which the structure of the virtual machine is manually reverse-engineered and the bytecode is converted into intermediate representation. After applying optimization to the intermediate representation, x86 code is generated.

Kinder [11] takes yet another outside-in approach using static analysis. Precise static analysis of a virtualization-obfuscated program is difficult because unrelated locations in the original program can be mapped to the same location in the obfuscated program. This problem, called domain flattening, is solved by using the virtual program counter.

Coogan et al. [8] takes an inside-out approach using dynamic analysis in which a relevant subtrace is extracted from a trace of an obfuscated program as an approximation of the trace of the original program. Here, a relevant subtrace is defined as one comprising relevant instructions, i.e., instructions that affect the values of the arguments of system calls of interest or the conditional control flow or instructions related to function calls and returns. Conditional control flow is analyzed using the equational reasoning system [7], and an expression for the target address is generated and simplified for each jump. Conditional control flow is considered relevant if the target address depends on the value of a flag operation. One

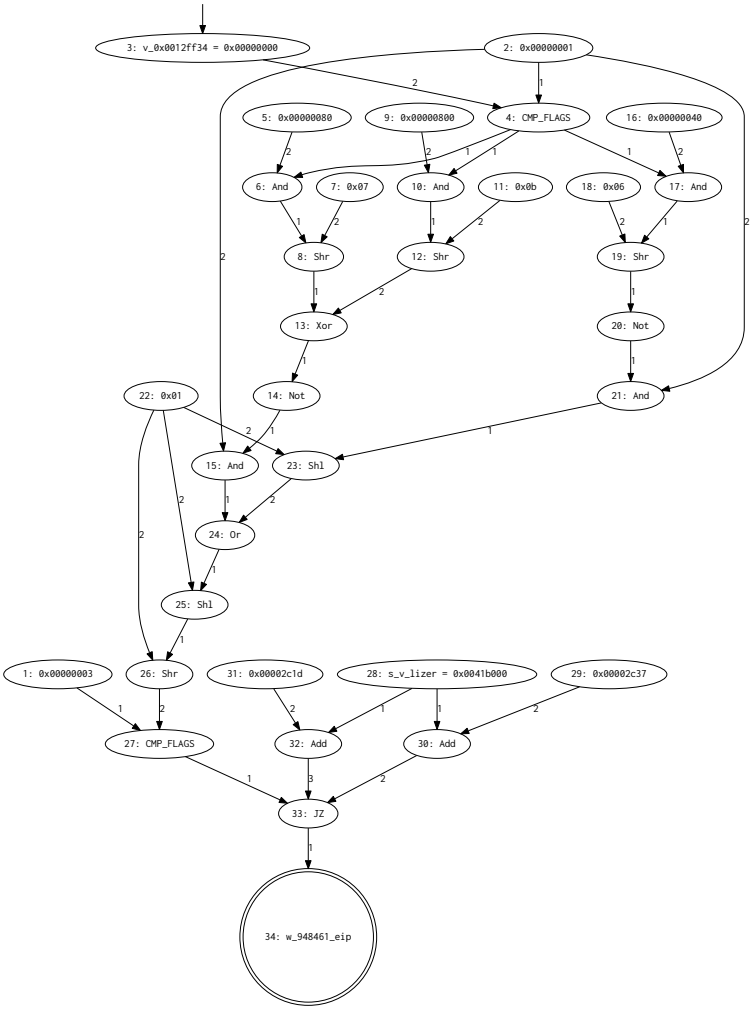

Figure 10: Simplified dynamic data flow graph for `i <= n` (Code Virtualizer 1).

limitation of this work is that, unlike our work, the use of constant values is not specially treated.

Yadegari et al. [24] develop enhanced bit-level taint analysis to analyze obfuscated machine code. They improve the precision of taint analysis through the use taint source information; for example, the result of the exclusive disjunction of two values with the same taint information is not considered tainted. An application of such taint analysis is its use in control flow graph construction [26], in which a control flow graph of an original program is constructed from traces of an obfuscated program using input-tainted conditional control transfers. Another application is its use in symbolic execution [25], in which a predicate is computed for each jump as a logical conjunction of conditions that are obtained from the taint information of the jump. The path constraint is updated by performing logical conjunction with these jump predicates and the input for a new path is generated by solving the path constraint.

One of the advantages of using dynamic data flow graphs instead of enhanced bit-level taint for analysis is that the former allows for more simplification. For example, the code in Figure 12 sets the value of the EAX register to zero. Assuming that `input_1` and `input_2` have values from different input sources, then dynamic data flow graphs can be used to simplify the final value of the EAX register to zero, which is not input-dependent. If enhanced bit-level taint analysis is used instead, different taint markings are given to the `input_1` and `input_2` operands, which causes another taint marking to be given to the result of the first XOR operation. Similarly, new taint markings are given to the results of the second and third XOR operations. As a result, the final value of the EAX register is considered as a tainted value. Improved precision therefore requires additional simplification.

Symbolic execution [2] has been actively applied to the analysis of opaque predicates. In Ming et al. [14], forward dynamic symbolic

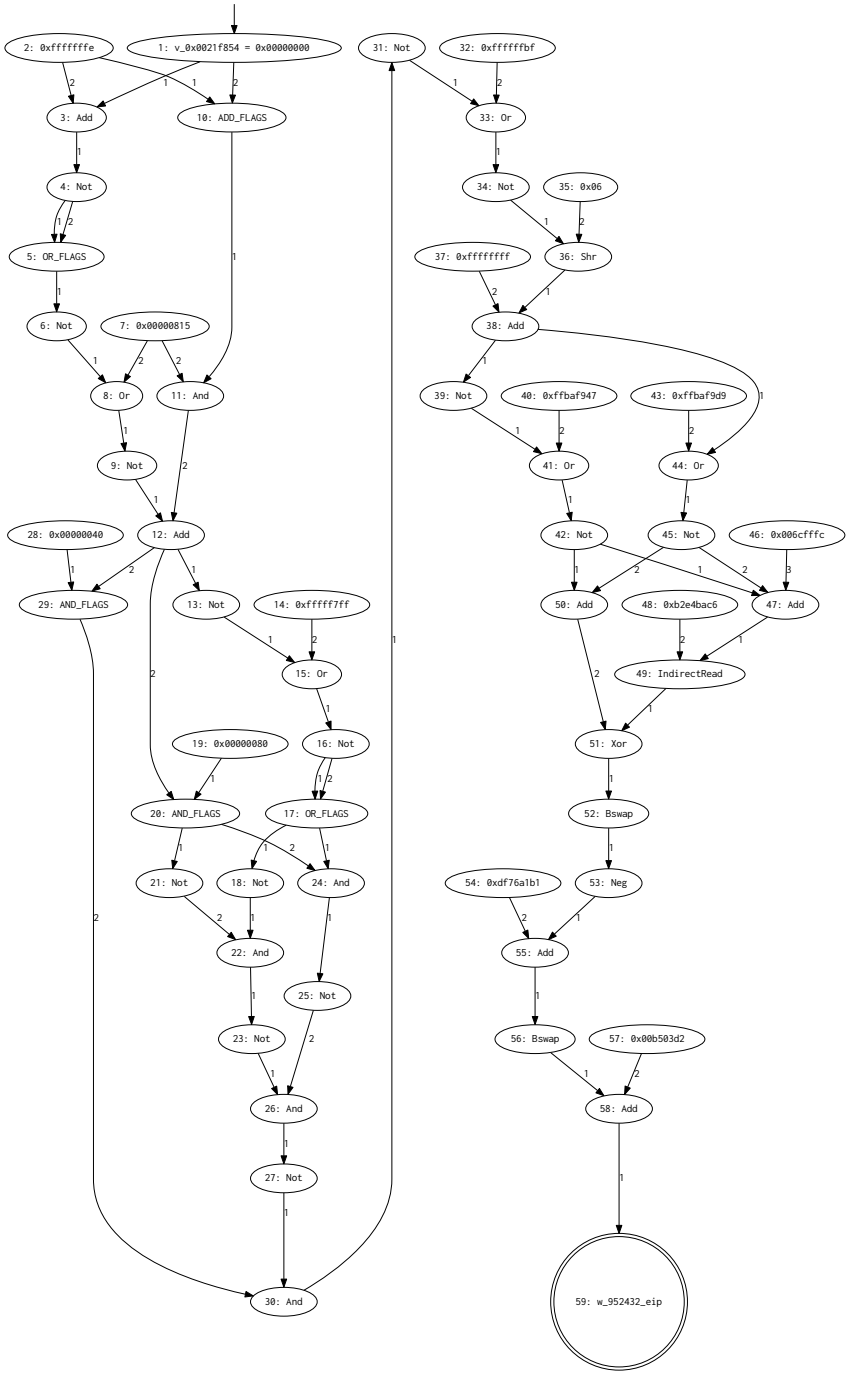

Figure 11: Simplified dynamic data flow graph for `i <= n` (VMProtect 3).

```
mov eax, input_1
xor eax, input_2
xor eax, input_1
xor eax, input_2
```

Figure 12: Setting the EAX register to zero.

execution is used to detect opaque predicates, while Bardin et al. [4] applies backward-bounded dynamic symbolic execution. Whereas the focus of such work is on opaque predicates using algebraic equalities and inequalities, our work is focused on how jumps are affected by the input. In Xu et al. [23], multiple granularity symbolic execution is used to simplify virtualized snippets.

In Salwan et al. [19], dynamic taint analysis and dynamic symbolic execution are used to generate intermediate representation for LLVM [12] from virtualization-obfuscated machine code. In this approach binary code is generated from the intermediate representation using LLVM. However, programs with user-dependent loop or memory access are not considered.

8 CONCLUSION

We developed a method for identifying input-dependent jumps from obfuscated execution using dynamic data flow graphs. Jumps from the original programs can be distinguished from the jumps from the obfuscation. Although generation and simplification of the dynamic flow graphs require considerable computational effort, this is justified by the reduction in human effort needed for manual analysis. The performance of the proposed method can be further improved by using better algorithms with parallel execution. Our method can be applied to improve other techniques such as symbolic execution. In future work, we intend to perform further control flow analysis using dynamic data flow graphs generated using the proposed method.

ACKNOWLEDGMENTS

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2015R1D1A1A01057195). The authors would like to thank the anonymous reviewers for their instructive comments.

REFERENCES

[1] Advanced Micro Devices, Inc. 2017. AMD64 Architecture Programmer's Manual, Volume 1: Application Programming (Revision 3.22).

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[3] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. 2001. On the (Im)possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference (CRYPTO '01)*. 1–18. https://doi.org/10.1007/3-540-44647-8_1

[4] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*. 633–651. https://doi.org/10.1109/SP.2017.36

[5] Christian Collberg. 2018. Reverse Engineering Challenges! Retrieved November 12, 2018 from http://tigress.cs.arizona.edu/challenges.html

[6] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. Department of Computer Science, The University of Auckland, New Zealand.

[7] Kevin Coogan and Saumya Debray. 2011. Equational Reasoning on x86 Assembly Code. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '11)*. 75–84. https://doi.org/10.1109/SCAM.2011.15

[8] Kevin Coogan, Gen Lu, and Saumya Debray. 2011. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*. 275–284. https://doi.org/10.1145/2046707.2046739

[9] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. 2013. Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits. In *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS '13)*. 40–49. https://doi.org/10.1109/FOCS.2013.13

[10] Matthew S. Hecht and Jeffrey D. Ullman. 1972. Flow Graph Reducibility. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing (STOC '72)*. 238–250. https://doi.org/10.1145/800152.804919

[11] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE '12)*. 61–70. https://doi.org/10.1109/WCRE.2012.16

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[13] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 190–200. https://doi.org/10.1145/1065010.1065034

[14] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15)*. 757–768. https://doi.org/10.1145/2810103.2813617

[15] Oreans Technologies. 2018. Code Virtualizer Overview. Retrieved November 12, 2018 from https://www.oreans.com/codevirtualizer.php

[16] Oreans Technologies. 2018. Themida Overview. Retrieved November 12, 2018 from https://www.oreans.com/themida.php

[17] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*. 73–96. https://doi.org/10.1007/978-3-319-60876-1_4

[18] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT '09)*.

[19] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Proceedings of the 15th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '18)*. 372–392. https://doi.org/10.1007/978-3-319-93411-2_17

[20] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (April 2016), 37 pages. https://doi.org/10.1145/2886012

[21] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. 94–109. https://doi.org/10.1109/SP.2009.27

[22] VMProtect Software. 2018. VMProtect Software Protection. Retrieved November 12, 2018 from https://vmpsoft.com/

[23] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS '18)*. 442–458. https://doi.org/10.1145/3243734.3243827

[24] Babak Yadegari and Saumya Debray. 2014. Bit-Level Taint Analysis. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '14)*. 255–264. https://doi.org/10.1109/SCAM.2014.43

[25] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15)*. 732–744. https://doi.org/10.1145/2810103.2813663

[26] Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*. 674–691. https://doi.org/10.1109/SP.2015.47