Scientia Iranica D (2019) 26(6), 3485–3509

SCIENTIA IRANICA

Sharif University of Technology
Scientia Iranica
Transactions D: Computer Science & Engineering and Electrical Engineering
http://scientiairanica.sharif.edu

Twinner: A framework for automated software deobfuscation

B. Momeni and M. Kharrazi*

Department of Computer Engineering, Sharif University of Technology, Tehran, P.O. Box 11155/1639, Iran.

Received 29 July 2017; received in revised form 6 October 2018; accepted 12 October 2019

KEYWORDS
Virtualization obfuscation;
Malware analysis;
Automated deobfuscation;
Twincode generation.

Abstract. Malware analysis is essential to understanding the internal logic and intent of malware programs in order to mitigate their threats. As the analysis methods have evolved, malware authors have adopted more techniques such as the virtualization obfuscation to protect the malware inner workings. This manuscript presents a framework for deobfuscating software, which abstracts the input program as much as a mathematical model of its behavior through monitoring every single operation performed during the malware execution. Also, the program is guided to run through its different execution paths automatically in order to gather as much knowledge as possible in the shortest time span. This makes it possible to find hidden logics and deobfuscate different obfuscation techniques without being dependent on their specific details. The resulting model is recoded as a C program without the artificially added complexities. This code is called a twincode and behaves in the same manner as the obfuscated binary. As a proof of concept, the proposed framework is implemented and its effectiveness is evaluated on obfuscated binaries. Program control flow graphs are inspected as a measure of successful code recovery. The performance of the proposed framework is evaluated using the set of SPEC test programs.

1. Introduction

Malware programs have evolved rapidly over the past decade, initially developed for fun, now being tools for financial profit and espionage. The new generation of malware, as depicted by the EquationDrug and the GrayFish [1], are constructed from well-developed modules responsible for a variety of duties such as exploitation, C&C communication, rootkit functionality, and so on. For example, the main module of the Flame [2] occupies about 6 MB and integrates all of the noted components together.

*. *Corresponding author. Tel.: +98 21 6616 6627*
E-mail addresses: b_momeni@ce.sharif.edu (B. Momeni); kharrazi@sharif.edu (M. Kharrazi)

doi: 10.24200/sci.2019.21601

Understanding the internal logic of a malware is of great importance in order to defend against it and limit its effectiveness. For example, knowledge of the Domain Name Generation (DNG) algorithm of a botnet could be used to predict its following command and control domain name; the propagation IP address selection algorithm of a worm indicates its infection strategy and may provide insight into its target goals; and knowledge of vulnerabilities employed by a rootkit make it possible to immune systems to them. Gaining this knowledge is not easy, because malware authors use obfuscation techniques to protect the internal logic of their code. Although initial obfuscation techniques focused on changing the function names or replacing them with multiple functions to create a more complex looking code, as time has passed, much more advanced techniques have been proposed to obfuscate the code not only in its look but also in its logic.

One of the advanced obfuscation techniques currently being used is Virtualization Obfuscation (VO) [3]. VO can be seen as a virtual machine for a dynamically constructed computing architecture instance. In its simplest form, the obfuscator has a single virtual language consisting of a small set of assembly instructions. Given a program in source/binary form, it is (re)compiled to this virtual language. The obfuscator can translate each assembly instruction from the source language (e.g., x86 assembly) to one or more instructions in the target language. Finally, it assigns random opcodes to the target language instructions. This makes the final binary representation look like pseudorandom bytes. The resulting code is stored in the data section and the executable code is replaced with a dynamically generated interpreter.

Many deobfuscation methods [3-8] depend on specific aspects of the obfuscator. In the analysis of a VO protected malware, static solutions need to analyze the executable, which is entirely replaced with the VO interpreter code. Therefore, they need to go deep into the data section, distinguish those bytes as data and encoded virtual opcodes, and decipher them into a set of instructions for the used virtual machine. This makes static analysis impractical specially if the VO is combined with other obfuscation techniques such as anti-disassembly and data encryption tricks. Dynamic solutions are also challenging since many aspects of the software runtime behavior such as Control Flow Graph (CFG) and memory/registers access patterns have changed. For example, although a solution, which inspects executed branches to reconstruct the program CFG, obtains a view of the VO interpreter CFG, it misses the higher level CFG, which has been formed among the virtual instructions. This may require the deobfuscator to put more precise assumptions about the VO strategy to focus on the hidden virtual program and the deobfuscator may fail when those VO strategies or used template languages are changed.

There have been two general approaches to deobfuscating the code. Consider an image which has been broken into pieces like a jigsaw puzzle. One approach to solving the puzzle and restoring the image is to examine how each individual piece can fit into another piece. For example, the low-level obfuscation patterns [9] are listed in a database and whenever one of them is detected in the obfuscated binary, it is replaced by its corresponding original code. Nevertheless, growing number of possible obfuscation techniques/patterns and dependence on the proper knowledge about each pattern complicate such an approach. An alternative approach is to focus on the major image components (e.g., a cloud in the sky) and try to group puzzle pieces which hold a similar content together. These solutions [10] consider high level characteristics such as the syscalls existence and capture the related instructions (e.g., to prepare syscall arguments). Such methods can be used against new obfuscators to analyze their behavior (e.g., list called syscalls), but they are not useful for learning the internal logic, such as the DNG algorithm of a bot or hidden behavior, e.g., a backdoor which remains inactive until a secret message is received. In other words, like in our puzzle example, those solutions put a number of pieces in the center of the puzzle as they look related to the mountains and put other pieces at the top as they look like the clouds, but they cannot match pieces which are gathered at the top or in the center of the puzzle together. They cannot generate any source code, detect/remove any deadcode, or analyze hidden logics, which may be executed if the malware environment is different.

This manuscript proposes a middle approach to the deobfuscation problem. Like in the puzzle example, an alternative approach is to understand what the image is and to slice it to create a new puzzle, which shows the same image but with much simpler pieces. In the context of deobfuscation problem, we mathematically model the malware by capturing its behavior alongside all possible execution paths. Afterwards, the obtained model is used to generate a new code, namely *twincode*, which is a compilable C program without the initial artificial complexities of the obfuscated malware and behaves exactly as the original binary. Most importantly, twincode enables an analyst to inspect the functionality of different parts of the program by modifying, compiling, and reevaluating it as required.

In the proposed framework, dynamic analysis is employed to instrument the input binary using Pin [11], tracing all assembly instructions, and monitoring the binary (including changes in registers, memory, and invoked syscalls). Furthermore, to decrease the required resources for a fixed degree of deobfuscation, <u>conc</u>rete-symb<u>olic</u> (concolic [12,13]) execution is employed by which concrete inputs (and required environment configuration) drive the execution through a specified path while the executed instructions are also inspected symbolically to find the input/output relationship for *all* inputs which could drive the program through the same execution path. As the program becomes more complicated, an open opportunity for understanding the internal code logic is maintained. Understanding the internal logic of a malware such as DNG algorithm of a bot, generation of comprehensive behavioral signatures for categorizing malware in their related families, debugging benign software which are obfuscated, and analysis of close sourced and obfuscated benign software for the presence of concealed backdoor are a few possible use cases of this framework.

The resulting twincode has a CFG similar to that of the original code. It performs the same memory changes as the original code and invokes the same syscalls with the same parameters and memory/registers states. Behavior of complicated operations is captured by symbolic expressions in order to eliminate the effects of the junk codes. Since twincode is obtained by monitoring and simplifying the program behavior through different execution paths, it is effective against behavior-preserving obfuscations. For example, if an obfuscation transformation adds to program behavior by checking for existence of a file, its corresponding twincode will also check for existence of that file. Twinner framework implementation is open sourced and available to the research community.

The contributions of this manuscript can be summarized as:

1. Proposing an automated deobfuscation framework based on the deep abstraction and recoding approach, keeping deobfuscation independent of known obfuscators;
2. Providing a canonical representation of the deobfuscated code (i.e., twincode), which is a structured C code with the same runtime behavior as the input program, but simplified for easier analysis;
3. Evaluating the proposed framework by employing a proof of concept implementation to deobfuscate a series of VO protected binaries.

The rest of this manuscript is structured as follows. Section 2 enumerates and briefly explains the related deobfuscation techniques. Section 3 describes the proposed framework. It enumerates key framework elements and reduces the deobfuscation problem into a set of subproblems. Section 4 deals with the implementation details of the proof of concept code, which is used for the practical evaluations in Section 5. Afterwards, Section 6 discusses the remaining challenges and how they can be addressed, and compares the proposed framework and other related works objectively. The manuscript is concluded in Section 7.

2. Related work

Analysis and deobfuscation solutions can be put in the following general categories. Some methods try to detect obfuscation layers and reverse them one by one. Such solutions [3,5] lead to high quality results but are always a step behind the obfuscators. Alternatively, a number of methods focus on specific program features which are hard to hide, such as used syscalls. These methods [6,10] work in the presence of new obfuscation techniques but produce lower quality results. Some frameworks [14,15] use these approaches to present binary inspection services to higher level analysis algorithms. But, there is also a third middle approach, which is not investigated adequately. It is to try to remove obfuscation effects by abstracting program features and then, recoding the resulting abstract behavioral model. This approach minimizes the transformation errors since no program portion is ignored during the inspection and simplification process and works despite future/unknown obfuscation techniques, because the higher abstraction of the intermediary model captures the core logic of the protected program itself independent of the applied obfuscations. The rest of this section enumerates and briefly compares some of the most notable studies related to the proposed framework.

The Rolles [3] method from the first category employs symbolic execution to map different parts of a VO protected malware onto symbolic functions. It assumes that the VM interpreter is reverse engineered once and the only unknown thing about it is the randomization of its opcodes and/or internal obfuscations of the VM code. It also assumes that it is notified when the execution of the VM part starts. The VM parts are converted to symbolic functions and compared with the reverse engineered VM parts using theorem proving techniques. When it finds invoked parts, it replaces them with the reversed codes. This method fails if (a) the malware used VM is instantiated from a different family of VMs (with some different template language) or (b) the theorem prover cannot find the equivalent symbolic function of the executed part among reverse engineered parts.

Kinder [4] approaches this problem by static analysis. The main issue of the static analysis is domain-flattening, in which the analysis of the VO protected malware leads to analysis of the interpreter. Assuming that VM has a decode-dispatch style, there would be some Virtual Program Counter (VPC) indicating the virtual opcode, which should be executed. Assuming that the VPC can be found by some analysis, Kinder resolves the domain-flattening issue by creating a separate state per each value of (PC, VPC) pair. This method does not recover any code to allow further analysis of the resulting program, but allows analysis of the malware behavior statically without merging calculated states after each cycle of the decode-dispatch loop.

Yadegari et al. [5] used taint propagation to identify data flows during an execution and then, applied a series of semantics preserving transformations to simplify that logic. For this purpose, they employed Ether to obtain code traces and then, applied five simplification methods to the traces. Finally, they constructed a CFG from the simplified traces. This method can work for packers which do not test their environment settings (e.g., execution delay) to conceal their runtime behavior. Peng et al. [16] proposed an algorithm for exploring all possible execution paths of an input program. It forces branch instructions by changing them in memory to follow an intended side.

ROPMEMU [17] focuses on return oriented programming as an obfuscation technique. It uses dynamic analysis to discover ROP gadgets, chain them to obtain higher level traces by skipping over some of return-to-library functions, and then merges gadget contents. It reconstructs CFG diagrams from these gadgets as its deobfuscated output. It also applies standard compiler optimizations in order to remove some redundancies.

Methods from the second category resist more complicated obfuscations, but they need to sacrifice their output completeness. The Coogan et al.'s [10] method relaxes the problem from reversing the given code into eliminating as much VM code as possible. It achieves this by inspecting invoked syscalls, checking their arguments, and finding causal dependencies among instructions to find codes with some influence on those arguments. The Rotalumé method [6] focuses on reversing the code of the VM itself (instead of the protected code, which is encoded in the data section). For this purpose, it needs to know about the used opcode values (random numbers) and their meaning. It assumes that VM has a decode-dispatch structure in which a big loop over a switch-case statement fetches opcodes and then, jumps to the corresponding part of the VM to emulate that instruction. Accordingly, Rotalumé tries to figure out fetched numbers as opcodes and code parts, which are executed after each fetch as their corresponding functionalities.

Another kind of malware obfuscation, which is worth noting, is using weird machines. Weird machines [18] are used in some unexpected manner. For example, the ELF file metadata can be used as a Turing-complete language to execute the malware while the executable program is not yet loaded [19]. Since our method starts instrumenting the input program before the start of runtime loader, it can watch instructions of runtime loader itself and generate twincode of ELF-metadata weird machine. However, there are weird machines which employ kernel-level codes for their execution. As the kernel code is not instrumented by our method, such machines remain undetectable. For example, a sequence of page fault and double page fault handling procedures can be used as a weird machine [20]. The weird machine executes the malware completely while no single instruction is fetched to be executed yet.

3. Framework

Even though dynamic analysis is a powerful approach to binary analysis, it has a set of issues which need to be considered. Specifically, the executed code may be analyzed incompletely due to missing proper inputs to uncover important code portions. Even when the program is executed along an interesting path, it may look too complex since a simple logical operation could be replaced with an obfuscated version. The first issue leads to a trade-off between the dynamic code coverage and the used time and memory resources. The second issue can lead, in an extreme case, to analysis of the used packing and/or emulation code instead of the protected code, influencing the analysis effectiveness.

```
1  int getsecret (unsigned int key, int salt) {
2    if (key > 5)
3      while (key > 0)
4        key++;
5    else if (salt > 5)
6      if (key > 6)
7        return complicatedFunction (key);
8      else
9        return (key>>24)
10         | (key>>16) | (key>>8) | key;
11   return (key ^ salt) % 1024;
12 }
```

Figure 1. An example code with several obfuscations, e.g., a while loop, which does not stop before an overflow; an unreachable code that calls a complicated function; and some arithmetical obfuscations.

In order to better understand the two noted issues, consider the code presented in Figure 1. For example, all code lines except lines 9-10 are there to hide the main logic of the `getsecret` function. Each time the program is executed, information about one of its many execution paths is obtained, depending on the values passed to the function as input. Without prior knowledge about the internal logic of the code, it is challenging to distinguish between inputs, which activate important paths or alternatively trigger traps providing no useful analysis results.

Furthermore, after traversing different execution paths and identifying deadcodes, important parts of the code may contain further obfuscations. For example, the conditional branch of line 2 guarantees that the key fits in one byte when line 9 is being executed. Thus, lines 9-10 can be simplified to return the key itself. Hence, each analyzed execution path should be simplified based on the asserted constraints in the collected context in order to remove the path-dependent obfuscations.

In what follows, the twinner framework is presented, which eliminates both syntactical and logical obfuscations by observing the executed binary code during a gray-box instrumented run, modeling its behavior, simplifying the model by abstracting the behavior/logic, and then regenerating the code according to the simplified logic. It should be noted that it is difficult to conceal high level program behaviors. For example, a variable may be stored in multiple memory locations to hide existence of a common variable. However, still those addresses have to be used instead of that common variable. Thus, abstracting out the memory addresses used cancels out the effect of such an obfuscation. In fact, the higher the abstraction level, the easier the transformation to simplified logic in practice.

Figure 2 shows the proposed framework in which the components/products are shown as rectangles/ellipses. This framework inspects an input obfuscated program iteratively using dynamic analysis. In each iteration, an execution path of the program is analyzed to obtain the relevant information of that part. Meanwhile, the program interacts with its environment, manipulates its memory/registers, and invokes syscalls. A malware can hide its logic in each part. For example, it can move user input through many locations to evade its ultimate usage place or it may obfuscate the memory contents by non-interfering dummy operations such as calculating $(\alpha + 1) \times 2 - 2$ instead of 2α expression.

All such activities are monitored symbolically to aggregate the obtained information and form an abstract behavioral model. Behavior of the program is simplified in two phases. First, while capturing each execution trace, several simplification rules are applied to the program input-to-output symbolic relationships. Second, while updating the behavioral model based on the simplified traces, a theorem proving tool is used to further simplify expressions and constraints as well as to remove possible deadcode branches. The model evolves by each iteration and at the end, a *twincode* is generated, which has a simplified logic but behaves exactly as if it is the original code.

The Concolic Execution Engine (CEE) component, shown at the bottom of Figure 2, deals with the dynamic analysis challenges, runs the protected binary in an specific desired path, and reports its symbolic execution results. The Program Search Strategy (PSS) component, shown at the center of Figure 2, employs the CEE in each iteration to learn about an execution path, updates its behavioral model accordingly, decides about the next path, and places a new call to the CEE component to analyze the next selected path. The PSS component also consults with the Satisfiability Modulo Theories Solver (SMTS) component, at the top of Figure 2, in order to find the required concrete inputs to inspect the next program paths and to simplify the obtained mathematical model.

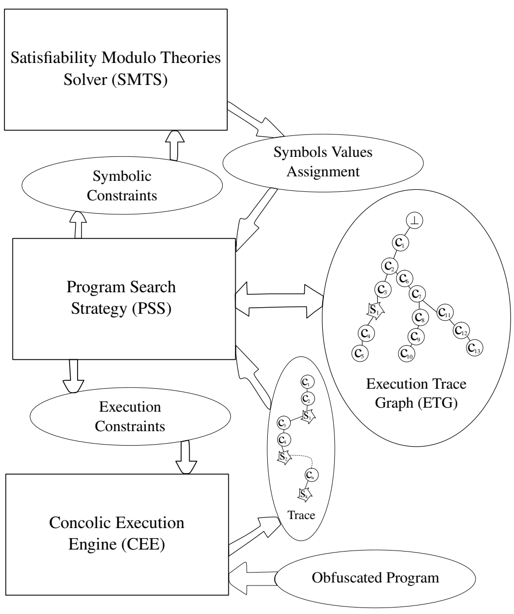


Figure 2. The deobfuscation framework in which components are shown as rectangles and products are drawn as ellipses.

The rest of this section discusses the proposed code behavior modeling approach in Section 3.1. Afterwards, Section 3.2 describes how the obtained model can be simplified by abstracting its behavior/logic and Section 3.3 discusses how it can be encoded as a twincode for further analysis.

3.1. Path exploration and modeling

An intermediate goal of the proposed deobfuscation framework is to obtain a behavioral model of the protected binary. This model is represented by the Execution Trace Graph (ETG) in the middle of Figure 2. Upon running a program, a series of assembly instructions are executed depending on the program inputs and its environment (e.g., network status). Each one of the followed execution paths covers parts of the program, forming a *trace*, which indicates a list of events including the satisfied conditions, the changes in the memory/registers contents, and invoked syscalls, all as symbolic expressions.

$$\text{statement } \mathfrak{s} ::= \text{confined } \mathfrak{c} \ \mid \text{illuminated } i. \tag{1}$$

Eq. (1) shows two types of statements which can be seen during the execution. Confined statements are those which are instrumented and their complete behavior can be tracked symbolically. Illuminated statements constitute the sources of uncontrolled randomness, such as invoked system calls. When a confined statement is executed, it updates the concrete memory state, the symbolic changes of the program, and satisfied constraints. When an illuminated statement is executed, it changes the concrete program state in an uncontrolled way and hence, it can lead to the creation of new symbols. The operational semantics of statements execution is shown in Figure 3. Each operation rule has the form of Eq. (2), in which E (bottom-left) shows the current state of the program and $\mathfrak{s}$ indicates the current statement. For execution of $\mathfrak{s}$, a series of computations are carried out (top) and consequently, the program state is updated to E' (bottom-right). The next statement is shown by $\mathfrak{s}'$:

$$\frac{\text{computations}}{E, \mathfrak{s} \rightsquigarrow E', \mathfrak{s}'}\texttt{Name-of-Operation}. \tag{2}$$

The first rule in Figure 3 models the execution of a confined statement $\mathfrak{c}$ while the concrete program state

$$\frac{f = Behavior(\mathfrak{c}) \quad c = Condition(\mathfrak{c}) \quad c(\mathcal{S}) = \top \quad c' = c \circ m\mathcal{C}_j \quad \mathfrak{s} = Next(f(\mathcal{S}), \mathfrak{c})}{\mathcal{S}, j, \langle \mathcal{TS}_k \rangle_{k=1}^{j-1}, \mathcal{C}_j, m\mathcal{C}_j, m\mathcal{I}, \mathfrak{c} \rightsquigarrow f(\mathcal{S}), j, \langle \mathcal{TS}_k \rangle_{k=1}^{j-1}, \mathcal{C}_j \wedge c', f \circ m\mathcal{C}_j, m\mathcal{I}, \mathfrak{s}} \texttt{CONFINED-OP}$$

$$\frac{\mathcal{S}' = \mathcal{S} \oplus m\mathcal{I}[j+1] \quad SC_j = Terminator(\mathfrak{i}) \quad \mathcal{TS}_j = (\mathcal{C}_j, m\mathcal{C}_j, SC_j) \quad \mathfrak{s} = Next(\mathcal{S}', \mathfrak{i})}{\mathcal{S}, j, \langle \mathcal{TS}_k \rangle_{k=1}^{j-1}, \mathcal{C}_j, m\mathcal{C}_j, m\mathcal{I}, \mathfrak{i} \rightsquigarrow \mathcal{S}', j+1, \langle \mathcal{TS}_k \rangle_{k=1}^{j}, \top, id_{m\mathcal{S}}, m\mathcal{I}, \mathfrak{s}} \texttt{ILLUMINATED-OP}$$

Figure 3. Operational semantics for execution of the program in CEE.

is denoted by $\mathcal{S}$. $Behavior(\mathfrak{c}) : m\mathcal{S} \rightarrow m\mathcal{S}$ is a function from the current memory state ($\mathcal{S}$) to the next state, encoding how the statement $\mathfrak{c}$ updates the program state. The memory state $m\mathcal{S} := \mathbb{N} \rightarrow \mathbb{N}$ is a map from memory addresses (and registers) to their encoded values. Finally, $Condition(\mathfrak{c}) : m\mathcal{S} \rightarrow \{\perp, \top\}$, is a function over the current program state, encoding the constraint which was checked by statement $\mathfrak{c}$. Updating the environment to invert the value of $Condition(\mathfrak{c})$ can change the subsequent instructions. Each execution trace, modeling a single execution path, can be divided into multiple trace segments ($\mathcal{TS}_k$) by illuminated statements. This division allows the symbolic changes of confined statements in each fully instrumented sequence of instructions to be kept separately. It also allows memory symbols, which are introduced by execution of illuminated statements, to be controlled and initialized at the beginning of each trace segment.

The `CONFINED-OP` rule updates the concrete state ($\mathcal{S}$) by passing it through the behavior function of $\mathfrak{c}$ statement ($f(\mathcal{S})$). As it does not finish the execution of a trace segment, the segment index (j) remains unchanged. Similarly, the sequence of trace segments ($\langle \mathcal{TS}_k \rangle_{k=1}^{j-1}$) is not affected. But, the symbolic encoding of memory changes and satisfied constraints in the current segment ($\mathcal{TS}_j$) should be updated based on the behavior (f) and condition (c) functions of statement $\mathfrak{c}$. $m\mathcal{C}_i : m\mathcal{S} \rightarrow m\mathcal{S}$, which maps the old program state (exactly before execution of $\mathcal{TS}_j$) onto its new state (after execution of $\mathcal{TS}_j$), can be updated by combining with the behavior function ($f \circ m\mathcal{C}_j$), because the program state should undergo changes of all executed instructions, sequentially ($f(m\mathcal{C}_j(.))$). $\mathcal{C}_i : m\mathcal{S} \rightarrow \{\perp, \top\}$ inspects the old program state (exactly before execution of $\mathcal{TS}_j$) and reports whether all assertions of $\mathcal{TS}_j$ are satisfied ($\mathcal{C}_i(.) = \top$) or some alternative sequence of statements and their corresponding segment should be executed ($\mathcal{C}_i(.) = \perp$). Because the condition function (c) of statement $\mathfrak{c}$ operates in the current program state ($\mathcal{S}$), we have to update it to take care of program state differences ($c' = c \circ m\mathcal{C}_j$) so that it can process the old program state and then, update the segment assertions using a boolean conjunction ($\mathcal{C}_j \wedge c'$). $m\mathcal{I} : \mathbb{N} \rightarrow \overline{m\mathcal{S}}$, which indicates how new symbols should be initialized at the beginning of each trace segment, is not affected too. Finally, the next statement, which is observed during the concolic execution, is denoted by $\mathfrak{s} = Next(f(\mathcal{S}), \mathfrak{c})$.

$$\mathcal{S}' = \mathcal{S} \oplus m\mathcal{I}[j+1](x) = \begin{cases} m\mathcal{I}[j+1](x), & m\mathcal{I}[j+1](x) \neq \perp \\ \mathcal{S}(x), & \text{otherwise} \end{cases} \quad (3)$$

The `ILLUMINATED-OP` rule is invoked after execution of an illuminated statement ($\mathfrak{i}$) and when the program state has been updated out of the instrumentation control. The program state ($\mathcal{S}$) needs to be updated as dictated in its corresponding memory initialization function ($m\mathcal{I}\ [j+1] : \overline{m\mathcal{S}}$ where $\overline{m\mathcal{S}} = \mathbb{N} \rightarrow \mathbb{N} \cup \{\perp\}$). As shown in Eq. (3), the new state is formed by overriding the memory cells which have been assigned to a new value by PSS decision. The trace segment index is increased to $(j+1)$ and the new trace segment is appended to the sequence of trace segments ($\mathcal{TS}_j = (\mathcal{C}_j, m\mathcal{C}_j, SC_j)$). $\mathcal{C}_j$ and $m\mathcal{C}_j$ are symbolic conditions and behavior encodings of the segment, which are accumulated during the execution of trace segment $\mathcal{TS}_j$, and $SC_j = Terminator(\mathfrak{i})$ denotes the terminating operation, which finishes execution of that segment (e.g., invoking the "open" system call). The condition and behavior tracking functions need to be updated for the new trace segment too. For this purpose, a tautology function ($\top$) is assigned as the new condition ($\mathcal{C}_{j+1}$) and an identity function ($id_{m\mathcal{S}}(x) = x$ for $x \in m\mathcal{S}$) is used for initializing the behavior function.

Definition 1 (Trace): *The trace $\mathcal{T}_n$ is a sequence of n trace segments $\langle \mathcal{TS}_k = (\mathcal{C}_k, m\mathcal{C}_k, SC_k) \rangle_{k=1}^{n}$ where the $\mathcal{TS}_k$ models a section terminated by $SC_k \in \mathbb{N}$. $\mathcal{C}_k : m\mathcal{S} \rightarrow \{\perp, \top\}$, and $m\mathcal{C}_k : m\mathcal{S} \rightarrow m\mathcal{S}$ are obtained as shown in the operational semantics in Figure 3 and they denote the conditions which must be satisfied in the program state before execution of $\mathcal{TS}_k$ in order to guide the execution through the statements of $\mathcal{TS}_k$ and observe changes of $m\mathcal{C}_k$ in the program state.*

Definition 2 (Guided execution): *When a program $\mathcal{P}$ is executed according to the operational semantics in Figure 3 and its memory state is overridden at the* *beginning of each trace segment $\mathcal{TS}_k$ according to $m\mathcal{I}$: $\mathbb{N} \to \overline{m\mathcal{S}}$, it is said that execution of $\mathcal{P}$ is guided by the function $m\mathcal{I}$. This guided execution is shown as $\mathcal{E}[\mathcal{P}, m\mathcal{I}]$, which is the set of all possible traces $\mathcal{T}$ that can be obtained through the execution.*

ETG is a graph consisting of all program traces. It is not necessarily a tree, because parts of the ETG can refer (via branches) to its other parts. However, it does not correlate with the obfuscated program CFG, because all branches that are resolved during a trace are simplified together. The ETG branches only when there is an input-dependent condition. For example, deobfuscating a VO protected program eliminates all checks on the values of the pseudo-opcodes, because inspecting those opcodes always leads to their corresponding implementation in the used virtual machine instance. Instead, the ETG corresponds to the original program CFG. The ETG is eventually encoded as the *twincode*, as will be discussed later in Section 3.3.

The PSS component is responsible for driving the deobfuscation process to extract information from the input binary iteratively. Pointing to the algorithm sketch in Figure 4, the PSS starts with an initial execution of the $\mathcal{P}$ obfuscated program. This initial trace establishes a single-path ETG to approximate the input program. In each iteration, PSS executes the binary through a new execution path trying to collect a set of traces covering all assembly instructions (static coverage) and all possible branching paths (dynamic coverage). As an example, given that a read register contains a symbol which was checked to be greater than α to follow an execution path, PSS can assign a number less than α to that symbol in order to guide the program along a new path in the following iteration. This issue is correlated with *deadcode* (i.e., the parts which cannot be executed with any input) detection challenge. To solve either of the above issues, the PSS needs to know whether there is a set of concrete values for the used symbols to drive the CEE along a specific path. This problem is equivalent to solving the constraints corresponding to the desired execution path, which is known as Satisfiability Modulo Theories (SMT) [21] solving. If an SMT query is proved to be unsatisfiable, the related code is a deadcode. If a concrete solution can be found, the CEE can be guided through that path.

The SMTS component, at the top of Figure 2, is designed to respond to SMT queries of PSS. Since SMT solvers are decision procedures, they can terminate eventually. However, deciding that a SMT query is satisfiable is an NP-complete problem and can be a performance bottleneck in the worst case. To ensure that SMTS terminates in a polynomial time similar to other components of the proposed framework, it is possible to configure a deadline for solving each SMT query and obtain a concrete solution. In the worst case, when the SMT solver cannot answer within a fixed-time deadline, the corresponding branch is left unexplored and marked as possible deadcode. An alternative in the worst-case scenarios is to negate the conditions of non-conforming branches in memory so that the solver can follow the desired side of each branch independent of the used concrete values. However, it can be used by a malware to slow down deobfuscation and delay analysis of other execution paths. The current PoC implementation does not use branch negation to avoid analysis of possible deadcode areas and save the analysis time for other parts.

The *search strategy* indicates which execution path should be queried thereafter. Putting collected traces together, it builds a conception of the complete ETG, searches within it, and decides where to explore more to construct a more comprehensive model. In other words, in each iteration, the constructed graph approximates the obfuscated binary in more details. An example of the strategy is to perform Depth-First Search (DFS) until an instruction is visited k times (hence, k rounds of an input-dependent loop unfolded). Search strategies and maximizing code coverage are research subjects [22,23] in the field of software automated testing.

function PSS($\mathcal{P}$) ▷ The $\mathcal{P}$ is an obfuscated binary program
    $\mathcal{T} \leftarrow$ CEE($\mathcal{P}$)
    ETG $\leftarrow \{\mathcal{T}\}$
    while ETG contains an uncovered path **do**
        $\zeta_i \leftarrow$ next uncovered execution path to analyze
        $\mathcal{C}_i \leftarrow$ SMTS(ζ_i)
        if $\mathcal{C}_i \neq$ Unsatisfiable **then**
            $\mathcal{T}_i \leftarrow$ CEE($\mathcal{P}, \mathcal{C}_i$)
            ETG $\leftarrow$ ETG $\cup \{\mathcal{T}_i\}$ ▷ Merges $\mathcal{T}_i$ trace with ETG graph
        end if
    end while
    return ETG encoded as twincode
end function

Figure 4. Sketch of the PSS algorithm using SMTS for solving symbolic constraints and CEE for executing through the intended paths.

3.2. Abstraction and simplification

As discussed earlier, dynamic analysis approaches need to address two main issues, namely effectiveness and dynamic coverage. In complex obfuscation scenarios, such as VO, the analysis may follow the logic of the used virtual machine instead of the protected code, making the deobfuscation ineffective (even though technically correct). To overcome the effectiveness challenge, instead of capturing the executed assembly instructions (like [8]), CEE captures the behavior of each executed instruction in terms of its mathematical formula. Although instructions can be obfuscated, their functionality and hence, the formula modeling how registers and memory addresses are changed by the program execution remain obtainable across all obfuscations. In addition, dynamic coverage makes a trade-off between completeness of analysis and the used time and memory resources. For this trade-off, concolic [12,13] execution is employed in which all instructions are executed normally (therefore, anti-disassembly techniques are mitigated automatically) and each instruction is inspected symbolically (therefore, the obtained input/output relationship is not restricted to the environment/concrete inputs).

Another challenge pertains to how to drive the program during the dynamic analysis (under the control of the CEE) along the execution path, which is determined by the PSS (in terms of a set of concrete inputs). For this purpose, the CEE traces all input values as symbols during the concolic execution. Therefore, the CEE has the chance of modifying values of all symbols in the memory and registers before they are read by the program.

Definition 3 (symbol): *A symbol $\mathcal{SYM} = (v, t)$ models a value which can be stored at some memory address or register. v is the concrete value represented by $\mathcal{SYM}$ in the current execution and t is the type of v (e.g., uint16_t) indicating the set of values which could be used by $\mathcal{SYM}$.*

Definition 4 (more constrained initialization relation): *A memory initialization function $m\mathcal{I}'$ is said to be "more constrained" than another memory initialization function $m\mathcal{I}$ and is denoted by $m\mathcal{I} \Subset m\mathcal{I}'$ if and only if $m\mathcal{I}'$ is exactly the same as $m\mathcal{I}$, but initializes more memory symbols. It is formally depicted in Eq. (4).*

$$\forall m\mathcal{I}, m\mathcal{I}' : \mathbb{N} \to \overline{m\mathcal{S}}. m\mathcal{I} \Subset m\mathcal{I}' \iff \Big(m\mathcal{I} \neq m\mathcal{I}' \wedge \forall j : \mathbb{N}. \forall x \in \mathbb{N}. \; m\mathcal{I}[j](x) \neq \perp \implies m\mathcal{I}[j](x) = m\mathcal{I}'[j](x) \Big). \quad (4)$$

An obfuscator can challenge dynamic analysis solutions in two main ways. On the one hand, it can add guard codes to hide the original program at runtime. A notable guard code is the latency-checker by which a time consuming action is performed and its measured time is compared with a threshold. As debugging slows down the execution, the malware can detect such extra latency. In defense, the notion of symbolic inputs can be extended to consider the intermediate memory states. For example, the `RDTSC` instruction loads the current time-stamp into registers. Although this value is not a user-input, it is not computed by the program either. It is a system-input. The CEE component can mark such inputs and report them back to the PSS. This resolves the guard code problem by asking the PSS component to find user and system inputs in order to maximize the analysis code coverage.

On the other hand, the computed function can be complicated (e.g., by additional neutral arithmetic); as a result, the observed functionality of the program remains incomprehensible. To address this issue, the calculated symbolic expression is simplified as it is gathered by the instruction analysis routines based on a set of expression simplification rules. Some of these rules are listed in Table 1. For the complete list, the *Operator* subclasses from the *exptoken* namespace in the Twinner git repository can be inspected. For example, the first row of Table 1 states that the bitwise AND of a symbol with $|S|$ bits and some bitmask, which is zero in its lowest $|S|$ bits, removes the symbol S.

Theorem 1 (repeatability): *Given a program $\mathcal{P}$ and one of its traces $\mathcal{T}_n = \langle \mathcal{TS}_k = (\mathcal{C}_k, m\mathcal{C}_k, SC_k) \rangle_{k=1}^{n}$, which has been acquired through an execution $\mathcal{E}[\mathcal{P}, m\mathcal{I}]$ for some memory initialization function $m\mathcal{I}$, either repeating the execution $\mathcal{E}[\mathcal{P}, m\mathcal{I}]$ always produces the same trace or there exists a more constrained memory initialization function $m\mathcal{I}^* : \mathbb{N} \to \overline{m\mathcal{S}}$ that always produces the same execution trace, as depicted in Eq. (5):*

$$\forall \mathcal{P}, m\mathcal{I}. \forall \mathcal{T}_n : \mathcal{E}[\mathcal{P}, m\mathcal{I}]. \exists m\mathcal{I}^* : \mathbb{N} \to \overline{m\mathcal{S}}. \Big((m\mathcal{I} = m\mathcal{I}^* \vee m\mathcal{I} \Subset m\mathcal{I}^*) \wedge |\mathcal{E}[\mathcal{P}, m\mathcal{I}^*]| = 1 \wedge \mathcal{T}_n \in \mathcal{E}[\mathcal{P}, m\mathcal{I}^*] \Big). \quad (5)$$

Proof. Repeating the execution $\mathcal{E}[\mathcal{P}, m\mathcal{I}]$ (using $m\mathcal{I} = m\mathcal{I}^*$), a trace such as $\mathcal{T}_m^* = \langle \mathcal{TS}_k^* = (\mathcal{C}_k^*, m\mathcal{C}_k^*, SC_k^*) \rangle_{k=1}^{m}$ is produced. If we can prove equality of this arbitrary trace with the target trace of the theorem ($\mathcal{T}_m^* = \mathcal{T}_n$), it is proved that no other trace will be produced during that guided execution

Table 1. List of the runtime symbolic expression simplification rules used within the CEE component. $X - Z$ show symbolic expressions; S shows a symbol; $a - e$ show concrete values; $|.|$ function shows the bit length; and $len(.)$ represents the number of used tokens.

Visited expression	Simplified formula	Required context
$S \wedge a$	0	$(2^{\lvert S \rvert} - 1) \wedge a = 0$
$S \wedge a$	S	$(2^{\lvert S \rvert} - 1) \wedge a = (2^{\lvert S \rvert} - 1)$
$Z \pm a + b$	$Z \pm c$	$c = a \pm b$
$Z \wedge 0$	0	
$Z \underset{\vee}{\overset{+}{-}} 0$	Z	
$(X \vee Y) \wedge a$	$(X' \vee Y') \wedge a$	$X' = X \wedge a,\ Y' = Y \wedge a$
		$len(X') \leq len(X), len(Y') \leq len(Y)$
$(X + Y) \wedge a$	$(X' + Y') \wedge a$	$a = 2^b - 1$
		$X' = X \wedge a,\ Y' = Y \wedge a$
		$len(X') \leq len(X), len(Y') \leq len(Y)$
$(X \overset{\wedge}{\vee} Y) \gg a$	$X' \overset{\wedge}{\vee} Y'$	$X' = X \gg a,\ Y' = Y \gg a$
		$len(X') \leq len(X), len(Y') \leq len(Y)$
$(Z \vee a) \wedge b$	b	$a \wedge b = b$
$(Z \underset{\oplus}{\overset{\wedge}{\vee}} a) \underset{\oplus}{\overset{\wedge}{\vee}} b$	$Z \underset{\oplus}{\overset{\wedge}{\vee}} c$	$c = a \underset{\oplus}{\overset{\wedge}{\vee}} b$
$(Z \overset{\ll}{\gg} a) \wedge b$	$(Z' \overset{\ll}{\gg} a) \wedge b$	$Z' = Z \wedge (b \overset{\gg}{\ll} a)$
		$len(Z') \leq (Z)$
$(Z \overset{\ll}{\gg} a) \ll b$	$Z \overset{\ll}{\gg} c$	$c = a \pm b$
$((Z \wedge a) \vee b) \wedge c$	$(Z \vee d) \wedge e$	$d = b \wedge c$
		$e = (a \vee d) \wedge c$
$((Z \wedge a) \underset{\times}{\overset{+}{-}} b) \wedge c$	$(Z \underset{\times}{\overset{+}{-}} b) \wedge c$	$a = 2^d - 1, c = a \wedge c$
$((Z \wedge a) \times b) \wedge c$	$(Z \wedge a) \times b$	$c = 2^d - 1, \lvert a \rvert + \lvert b \rvert \leq \lvert c \rvert$
$(Z \times a) \overset{\times}{/} b$	$Z \times c$	$c = a \overset{\times}{/} b$
$(Z \overset{\wedge}{\vee} a) \ll b$	$(Z \ll b) \overset{\wedge}{\vee} c$	$c = a \ll b$
$(Z \overset{\times}{/} a) \ll b$	$Z \ll d$	$a = 2^c, d = b \pm c, d > 0$

and hence, $|\mathcal{E}[\mathcal{P}, m\mathcal{I}^*]| = 1$ which completes the proof. For this purpose, it is enough to show that $n = |\mathcal{T}_n| = |\mathcal{T}_m^*| = m$ and $(\mathcal{C}_k, m\mathcal{C}_k, SC_k) = (\mathcal{C}_k^*, m\mathcal{C}_k^*, SC_k^*)$ for $1 \leq k \leq n$. Also, in scenarios that this equality cannot be proved, $\mathcal{E}[\mathcal{P}, m\mathcal{I}^*]$ can be changed by adding more constraints to $m\mathcal{I}^*$ in order to maintain the equality of traces while the equation $m\mathcal{I} \Subset m\mathcal{I}^*$ is not violated. In these scenarios, the second form of the target proposition (Eq. (6)) is proved.

$$\Big((m\mathcal{I} \Subset m\mathcal{I}^*) \wedge |\mathcal{E}[\mathcal{P}, m\mathcal{I}^*]| = 1 \wedge \mathcal{T}_n \in \mathcal{E}[\mathcal{P}, m\mathcal{I}^*]\Big). \tag{6}$$

In both traces, execution of $\mathcal{P}$ is initiated by the same statement; thus, the initial states of $\mathcal{TS}_1$ and $\mathcal{TS}_1^*$ trace segments are the same. If we can prove that this equality is maintained during the execution of the first trace segment and the same sequence of statements is executed in both runs, the final system call of both segments will be the same (since it is the last statement obtained during the execution of the first segment) and it observes the same concrete state of $\mathcal{P}$ (because the equality has been preserved during the execution of that trace segment). This ensures that either both traces will terminate or both will continue execution from the second trace segment. However, the possible differences in the kernel state can make the initial concrete states of the following two trace segments different.

By induction, we prove that if the first q trace segments are equal ($\langle \mathcal{TS}_k \rangle_{k=1}^q = \langle \mathcal{TS}_k^* \rangle_{k=1}^q$), the $q + 1$th trace segments ($\mathcal{TS}_{q+1}$ and $\mathcal{TS}_{q+1}^*$), which have to start with the same statements but different concrete states, can be made equal only by adding more constraints to $m\mathcal{I}^*$ while maintaining the equation $m\mathcal{I} \Subset m\mathcal{I}^*$. This completes the proof by showing that $n = m$, because no trace can be terminated earlier when the concrete states of both are the same at the time of invoking the same system calls. For this proof, we should note that the first statement of $\mathcal{TS}_{q+1}$ and

$$\frac{\mathcal{S}'_{q+1} = \mathcal{S}_{q+1} \oplus m\mathcal{I}[q+1] \; SC_q = Terminator(\mathfrak{i}) \; \mathcal{TS}_q = (\mathcal{C}_q, m\mathcal{C}_q, SC_q) \; \mathfrak{s} = Next(\mathcal{S}'_{q+1}, \mathfrak{i})}{\mathcal{S}_{q+1}, q, \langle \mathcal{TS}_k \rangle_{k=1}^{q-1}, \mathcal{C}_q, m\mathcal{C}_q, m\mathcal{I}, \mathfrak{i} \rightsquigarrow \mathcal{S}'_{q+1}, q+1, \langle \mathcal{TS}_k \rangle_{k=1}^{q}, \top, id_{m\mathcal{S}}, m\mathcal{I}, \mathfrak{s}} \texttt{ILLUMINATED-OP}. \tag{7}$$

Box I

$\mathcal{TS}^*_{q+1}$ is an illuminated statement $\mathfrak{i}$, which is seen after return of the $SC_q = SC^*_q$, while the concrete states of the two runs (see Figure 3) can differ ($\mathcal{S}_{q+1} \neq \mathcal{S}^*_{q+1}$) before executing $\mathfrak{i}$ itself. However, after execution of $\mathfrak{i}$, all following statements should be confined, because the first following illuminated statement will terminate the trace segment. Execution of the first illuminated statement uses the operation rule of Eq. (7) shown in Box I.

The set of possible difference points between two functions $u, v : \mathbb{N} \to \mathbb{N}$ can be obtained by Eq. (8):

$$u \ominus v = v \ominus u = \{x : \mathbb{N} | u(x) \neq v(x)\}. \tag{8}$$

The possible difference points of the initial program states ($\mathcal{S}_{q+1} \ominus \mathcal{S}^*_{q+1}$) can either be overridden by $m\mathcal{I}[q+1]$ during execution of $\mathfrak{i}$ (in which case it is not required to update $m\mathcal{I}^*$) or will remain different in some cases between two sets ($(\mathcal{S}_{q+1} \ominus \mathcal{S}^*_{q+1}) - \{x \in \mathbb{N} | m\mathcal{I}[q+1](x) \neq \bot\}$). In this case, the concrete states can be made equal by updating $m\mathcal{I}^*[q+1]$ according to Eq. (9) shown in Box II. This ensures that $\mathcal{S}'_{q+1} = \mathcal{S}'^*_{q+1}$.

Because this reasoning is made about an arbitrary member of the new guided execution set ($\mathcal{T}^*_m \in \mathcal{E}[\mathcal{P}, m\mathcal{I}^*]$), it shows that from any initial difference set such as ($\mathcal{S}_{q+1} \ominus \mathcal{S}^*_{q+1}$), an appropriate patch can be constructed for updating $m\mathcal{I}^*[q+1]$ and consequently equating trace segments $\mathcal{TS}_{q+1}$ and $\mathcal{TS}^*_{q+1}$. However, for constructing $m\mathcal{I}^*[q+1]$ in practice, there are two approaches. One approach is to override all memory cells which were not overridden by the first memory initialization vector, as shown in Eq. (10):

$$m\mathcal{I}^*[q+1] = \mathcal{S}_{q+1} \oplus m\mathcal{I}[q+1]. \tag{10}$$

Another approach is to repeat the execution iteratively, once running $\mathcal{E}[\mathcal{P}, m\mathcal{I}^{(1)}]$, where $m\mathcal{I}^{(1)} = m\mathcal{I}$, in order to obtain difference sets and produce $m\mathcal{I}^{(2)}$ for overriding those differences and then, running $\mathcal{E}[\mathcal{P}, m\mathcal{I}^{(2)}]$ to obtain $m\mathcal{I}^{(3)}$ and so on. Since more cells are initialized in each turn, the initialization vector becomes more constrained ($m\mathcal{I}^{(1)} \Subset m\mathcal{I}^{(2)} \Subset ... \Subset m\mathcal{I}^*$) and the guided execution set will converge to a single-member set.

Since concrete program states are made equal, the next executing statements in both trace segments will also be the same ($\mathfrak{s} = Next(\mathcal{S}'_{q+1}, \mathfrak{i}) = Next(\mathcal{S}'^*_{q+1}, \mathfrak{i})$). In addition, the initial constraints and behavior functions are initialized with the same value ($\top, id_{m\mathcal{S}}$). Execution of the next confined statement $\mathfrak{c}$ is performed by the operation rule of Eq. (11), shown in Box III.

Because both trace segments are running the same $\mathfrak{c}$ statement, their condition/behavior functions will also be the same ($f = f^*$ and $c = c^*$). Furthermore, since $\mathcal{S}_{q+1} = \mathcal{S}^*_{q+1}$ before execution of $\mathfrak{c}$, the following concrete states will also be the same ($f(\mathcal{S}_{q+1}) = f^*(\mathcal{S}^*_{q+1})$). Similarly, the updated condition (Eq. (12)) and behavior (Eq. (13)) functions will be equal. Finally, equality of the following concrete states ($f(\mathcal{S}_{q+1}) = f^*(\mathcal{S}^*_{q+1})$) leads to execution of the same following statements ($\mathfrak{s} = Next(f(\mathcal{S}_{q+1}), \mathfrak{c}) = Next(f^*(\mathcal{S}^*_{q+1}), \mathfrak{c})$).

$$\begin{aligned} \mathcal{C}_{q+1} \wedge c' &= \mathcal{C}_{q+1} \wedge (c \circ m\mathcal{C}_{q+1}) \\ &= \mathcal{C}^*_{q+1} \wedge (c^* \circ m\mathcal{C}^*_{q+1}) = \mathcal{C}^*_{q+1} \wedge c'^*, \end{aligned} \tag{12}$$

$$m\mathcal{I}^*[q+1](x) = \begin{cases} m\mathcal{I}[q+1](x), & m\mathcal{I}[q+1](x) \neq \bot \\ \mathcal{S}_{q+1}(x), & m\mathcal{I}[q+1](x) = \bot \wedge \mathcal{S}_{q+1}(x) \neq \mathcal{S}^*_{q+1}(x) \\ \bot, & \text{otherwise} \end{cases} \tag{9}$$

Box II

$$\frac{f = Behavior(\mathfrak{c}) \; c = Condition(\mathfrak{c}) \; c(\mathcal{S}_{q+1}) = \top \; c' = c \circ m\mathcal{C}_{q+1} \; \mathfrak{s} = Next(f(\mathcal{S}_{q+1}), \mathfrak{c})}{\mathcal{S}_{q+1}, q+1, \langle \mathcal{TS}_k \rangle_{k=1}^{q}, \mathcal{C}_{q+1}, m\mathcal{C}_{q+1}, m\mathcal{I}, \mathfrak{c} \rightsquigarrow f(\mathcal{S}_{q+1}), q+1, \langle \mathcal{TS}_k \rangle_{k=1}^{q}, \mathcal{C}_{q+1} \wedge c', f \circ m\mathcal{C}_{q+1}, m\mathcal{I}, \mathfrak{s}} \texttt{CONFINED-OP}. \tag{11}$$

Box III

$$f \circ m\mathcal{C}_{q+1} = f^* \circ m\mathcal{C}^*_{q+1}. \tag{13}$$

Repeating this reasoning for every following confined statement $\mathfrak{c}$, the concrete and symbolic program states will remain equal in both trace segments until execution of the next illuminated statement $\mathfrak{i}$, which terminates the trace segment.□

Theorem 1 states that all programs are deterministic. That is, without interacting with the outside world (e.g., an OS syscall), a program has no inherent source of randomness which is not marked as a symbolic input. In other words, since all instructions, which can provide some inputs to the program (e.g., syscalls or hardware timers), are instrumented, those inputs can be modified by CEE before being used for the first time. For example, opening a file may succeed/fail at different times, but even if a file is deleted, the malware can be made to believe that the file is present by feeding the file data stream to the malware through memory modifications. Ensuring that the environment is not changed at all (or the malware cannot sense those changes), re-executing it leads to the same results all the times.

3.3. Twincode generation

Completing the introduced chain of components of the proposed framework, this section explains how the twincode is generated as the ultimate result. Twincode is structured as the ETG, encoding the same conditions of the analyzed program, preparing memory/registers contents as a function of their previous states, and invoking the same syscalls. Consequently, it behaves like the obfuscated program with the difference that its functionality is simplified and visible in the C code. Thus, it can be used as input to other analysis techniques and/or manual inspection. Each trace represents the program behavior during an execution path. Therefore, if all execution paths of the input program are determined, their corresponding traces can be mined out and putting those traces together, the final twincode can be generated.

Definition 5 (twincode of a program): *The twincode of the program $\mathcal{P}$, shown as $\psi(\mathcal{P})$, is an encoding of its ETG as shown in Figure 5.*

The algorithm in Figure 5 starts with the ETG root node and visits its nodes as follows. Visiting a node such as u, which is connected to n other nodes, an `if-else` construct is outputted having an `if` part for each one of the n connected nodes. Upon visiting a segment terminator node, the symbolic changes of that part are simplified using SMTS and outputted; the syscall invocation code for its corresponding syscall is outputted afterwards; register/memory symbols of the next segment are instantiated and initialized with concrete register/memory values at that time instant; and the segment identifier is incremented. Upon visiting a previously encoded ETG node, its previous encoding is reused by moving its corresponding code to a separate function.

```
function EncodeAsTwincode(g : ETG)
    u ← Root(g)
    Output RegSymbolsDeclaration(u, 0)
    Output MemInitialization(u)
    Output MemSymbolsDeclaration(u, 0)
    EncodeChildren(u, 0)
end function
function EncodeChildren(u : Node, i : ℕ)
    SetOutputSeparator("else")
    for all c ∈ Children(u) do
        C ← Condition(c)
        Output "if (C)"
        if IsSegmentTerminator(c) then
            TS ← Segment(c)
            Output MemChanges(TS)
            Output RegChanges(TS)
            SC ← Syscall(TS)
            Output "InvokeSyscall(SC)"
            i ← i + 1
            Output RegSymbolsDeclaration(c, i)
            Output MemSymbolsDeclaration(c, i)
        end if
        EncodeChildren(c, i)
    end for
end function
```

Figure 5. Sketch of the twincode encoding algorithm.

As the program is analyzed more in each iteration, its new parts are discovered and its ETG is evolved. Thus, any realization of the PSS must be able to perform the search incrementally. Another feature of the twincode is that, starting from different obfuscated versions of a program, similar twincodes are produced. In other words, twincodes of all obfuscated versions will contain traces which can be obtained from the original program. This allows the twincode to be analyzed instead of a specific obfuscated version.

Definition 6 (equivalent programs): *Two programs $\mathcal{P}$ and $\mathcal{P}^*$ are equivalent, shown as $\mathcal{P} \equiv \mathcal{P}^*$, if and only if they behave equivalently for all possible memory initializations (inputs), as depicted in Eq. (14).*

$$\forall \mathcal{P}, \mathcal{P}^*. \mathcal{P} \equiv \mathcal{P}^* \iff \forall m\mathcal{I} : \mathbb{N} \rightarrow \overline{m\mathcal{S}}. \mathcal{E}[\mathcal{P}, m\mathcal{I}] = \mathcal{E}[\mathcal{P}^*, m\mathcal{I}]. \tag{14}$$

Theorem 2 (equivalence): *If $\mathcal{O}_\mathcal{P}$ and $\mathcal{O}^*_\mathcal{P}$ are two obfuscated instances of program $\mathcal{P}$, with the assumption that the used obfuscation transformation has neither eliminated nor added to the observable behavior of $\mathcal{P}$, both $\psi(\mathcal{O}_\mathcal{P})$ and $\psi(\mathcal{O}^*_\mathcal{P})$ are equivalent with $\mathcal{P}$, as depicted in Eq. (15).*

$$\forall \mathcal{P}, \mathcal{P}^*, \mathcal{O}_{(.)}. [(\mathcal{O}_\mathcal{P} \equiv \mathcal{P}) \land (\mathcal{O}^*_\mathcal{P} \equiv \mathcal{P})] \implies \psi(\mathcal{O}_\mathcal{P}) \equiv \psi(\mathcal{O}^*_\mathcal{P}) \equiv \mathcal{P}. \tag{15}$$

Proof. In order to prove the target proposition ($\psi(\mathcal{O}_\mathcal{P}) \equiv \psi(\mathcal{O}^*_\mathcal{P}) \equiv \mathcal{P}$) based on the given assumption ($\mathcal{O}_\mathcal{P} \equiv \mathcal{O}^*_\mathcal{P} \equiv \mathcal{P}$), we can use the proof by contradiction technique. In other words, we should start by assuming that Eq. (16) holds and show that it leads to a contradiction.

$$[\mathcal{O}_\mathcal{P} \equiv \mathcal{O}^*_\mathcal{P} \equiv \mathcal{P}] \wedge [(\psi(\mathcal{O}_\mathcal{P}) \not\equiv \mathcal{P}) \vee \psi(\mathcal{O}^*_\mathcal{P}) \not\equiv \mathcal{P}] . \quad (16)$$

Without loss of generality, assume that $\psi(\mathcal{O}_\mathcal{P}) \not\equiv \mathcal{P}$ holds (the same reasoning can be used about the case of $\psi(\mathcal{O}^*_\mathcal{P}) \not\equiv \mathcal{P}$). Note Eqs. (17) to (22) shown in Box IV.

Alternatively, in Eq. (19), it is possible to say:

$$\exists \mathcal{T}_m^{\mathcal{P}[x]} \in \mathcal{E}[\mathcal{P}, m\mathcal{I}].\forall \mathcal{T}_n^{\psi(\mathcal{O}_\mathcal{P})[x]} \in \mathcal{E}[\psi(\mathcal{O}_\mathcal{P}), m\mathcal{I}].$$
$$\mathcal{T}_n^{\psi(\mathcal{O}_\mathcal{P})[x]} \neq \mathcal{T}_m^{\mathcal{P}[x]}.$$

However, due to their symmetry, we use the first case without loss of generality. Thus, the opposite assumption of Eq. (17) leads to $\mathcal{O}_\mathcal{P} \not\equiv \mathcal{P}$ in Eq. (22), which is in contradiction with the assumption of $\mathcal{O}_\mathcal{P} \equiv \mathcal{P}$.□

Providing better, faster, and more optimized search strategies and/or SMT solvers, as presented in [24], automatically provides a better twincode generator. This is a notable contribution, which allows the research line of software deobfuscation to directly benefit from progress in the software automated testing and SMT solving fields. Any SMT solver can be fitted in this framework affecting the analysis performance, but not its correctness or completeness. The following section explains the details of the implementation and the related practical concerns.

4. Implementation

This section discusses how the implementation has been carried out and enumerates some of the major practical challenges which should be resolved in the process. It is noteworthy that the proposed framework has been implemented in C++ and consists of more than 35 thousand lines of code. Furthermore, the code is released under GPLv3 through the project page in [25].

Figure 6 depicts the component diagram of this reference implementation. The PSS is realized by the *twinner* component. It provides a command line interface to configure the deobfuscation parameters and uses DFS as its search strategy. In each analysis round, it forks a new process to execute the CEE; extracts a trace information including a sequence of satisfied constraints; and updates the ETG accordingly. Although the DFS works over an evolving graph, it does not miss any part since the graph can be ordered unambiguously. ETG nodes contain constraints and are ordered in such a way that all constraints observed in the former traces are placed on the left of those which are seen afterwards.

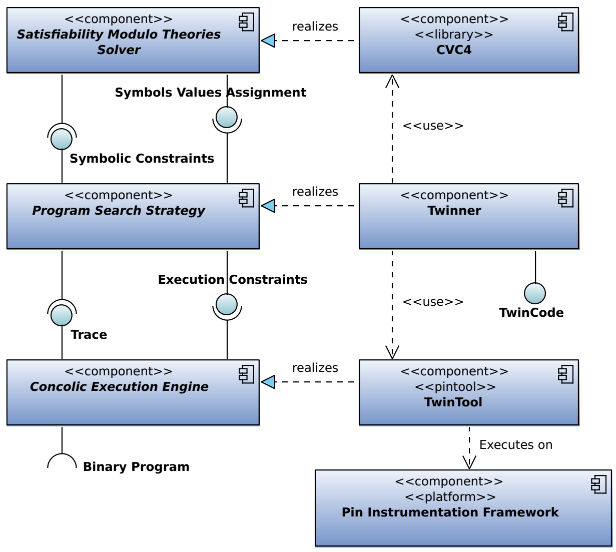


Figure 6. Component diagram of the proposed framework reference implementation.

$$\psi(\mathcal{O}_\mathcal{P}) \not\equiv \mathcal{P}, \quad (17)$$

$$\Longrightarrow \exists m\mathcal{I} : \mathbb{N} \to \overline{m\mathcal{S}}.\mathcal{E}[\psi(\mathcal{O}_\mathcal{P}), m\mathcal{I}] \neq \mathcal{E}[\mathcal{P}, m\mathcal{I}] \qquad \triangleright \text{ by Definition 6,} \quad (18)$$

$$\Longrightarrow \exists \mathcal{T}_n^{\psi(\mathcal{O}_\mathcal{P})[x]} \in \mathcal{E}[\psi(\mathcal{O}_\mathcal{P}), m\mathcal{I}].\forall \mathcal{T}_m^{\mathcal{P}[x]} \in \mathcal{E}[\mathcal{P}, m\mathcal{I}].\mathcal{T}_n^{\psi(\mathcal{O}_\mathcal{P})[x]} \neq \mathcal{T}_m^{\mathcal{P}[x]} \qquad \triangleright \text{ by Definition 2,} \quad (19)$$

$$\Longrightarrow \exists \mathcal{T}_n^{\mathcal{O}_\mathcal{P}[x]} \in \mathcal{E}[\mathcal{O}_\mathcal{P}, m\mathcal{I}].\mathcal{T}_n^{\mathcal{O}_\mathcal{P}[x]} = \mathcal{T}_n^{\psi(\mathcal{O}_\mathcal{P})[x]} \wedge \forall \mathcal{T}_m^{\mathcal{P}[x]} \in \mathcal{E}[\mathcal{P}, m\mathcal{I}].\mathcal{T}_n^{\mathcal{O}_\mathcal{P}[x]} \neq \mathcal{T}_m^{\mathcal{P}[x]} \qquad \triangleright \text{ by Definition 5,} \quad (20)$$

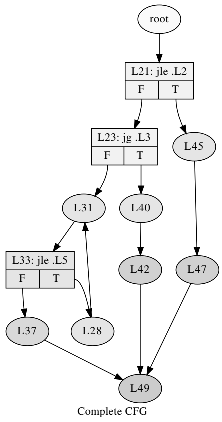
$$\Longrightarrow \exists m\mathcal{I} : \mathbb{N} \to \overline{m\mathcal{S}}.\mathcal{E}[\mathcal{O}_\mathcal{P}, m\mathcal{I}] \neq \mathcal{E}[\mathcal{P}, m\mathcal{I}] \qquad \triangleright \text{ by Definition 2,} \quad (21)$$

$$\Longrightarrow \mathcal{O}_\mathcal{P} \not\equiv \mathcal{P} \qquad \triangleright \text{ by Definition 6.} \quad (22)$$

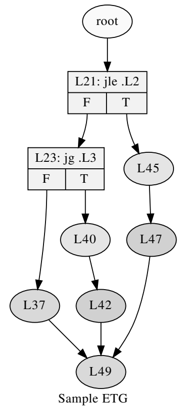
Box IV

```
1  .section .rodata
2 .LC0:
3  .string "overflow\n"
4 .LC1:
5  .string "underflow\n"
6 .LC2:
7  .string "%d\n"
8  .text
9 main:
10  pushq %rbp
11  movq %rsp, %rbp
12  subq $32, %rsp
13  movl %edi, -20(%rbp)
14  movq %rsi, -32(%rbp)
15  movq -32(%rbp), %rax
16  addq $8, %rax
17  movq (%rax), %rax
18  movzbl (%rax), %eax
19  movb %al, -9(%rbp)
20  cmpb $47, -9(%rbp)
21  jle .L2
22  cmpb $57, -9(%rbp)
23  jg .L3
24  movsbl -9(%rbp), %eax
25  subl $48, %eax
26  movl $0, -8(%rbp)
27  jmp .L4
28 .L5:
29  addl %eax, %eax
30  addl $1, -8(%rbp)
31 .L4:
32  cmpl $9, -8(%rbp)
33  jle .L5
34  movl %eax, %esi
35  movl $.LC2, %edi
36  movl $0, %eax
37  call printf
38  movl $0, %eax
39  jmp .L7
40 .L3:
41  movl $.LC0, %edi
42  call puts
43  movl $-2, %eax
44  jmp .L7
45 .L2:
46  movl $.LC1, %edi
47  call puts
48  movl $-1, %eax
49 .L7:
50  leave
51  ret
```

(a) X86_64 code (AT&T syntax)



(b) Non-pruned CFG



(c) The ETG

```
#include "twincode.h"

int main (int argc, char *argv[]) {
 struct RegistersSet regs;
 SAVE_REGISTERS (regs);
 const UINT64 rax_0 = regs.rax, rbx_0 = regs.rbx, rsp_0 = regs.rsp, ......;
 const UINT8 m7fffffffe5eb_0_8 = *((UINT8 *) 0x7fffffffe5eb);
 ......
 if ((((UINT64 (rsp_0) + UINT128 (0xffffffff, 0xffffffff, 0xffff8000, 0x1d28)) & 0xffffffffffffffff) == 0)) {
  if ((((UINT64 (argv[1]) + UINT128 (0xffffffff, 0xffffffff, 0xffff8000, 0x1a15)) & 0xffffffffffffffff) == 0)) {
   if ((UINT128 (0x0, 0x0, 0x0, 0x2f) < signExtend_0x80_0x8 (UINT64 (m7fffffffe5eb_0_8)))) {
    if ((UINT128 (0x0, 0x0, 0x0, 0x39) >= signExtend_0x80_0x8 (UINT64 (m7fffffffe5eb_0_8)))) {
     ......
     regs.rdi = UINT64 (0x4006a7);
     regs.rsi = UINT64 ((((signExtend_0x20_0x8 (m7fffffffe5eb_0_8) - 0x30) * 0x400) & 0xffffffff));
     printf (/*"%d\n"*/ (const char *) 0x4006a7,
       (UINT64) (((signExtend_0x20_0x8 (m7fffffffe5eb_0_8) - 0x30) * 0x400) & 0xffffffff));
     regs.rax = UINT64 (0x0);
     return int (regs.rax);
    } else {
     regs.rdi = UINT64 (0x400694);
     puts (/*"overflow"*/ (const char *) 0x400694);
     regs.rax = UINT64 (0xfffffffe); // -2
     return int (regs.rax);
    }
   } else {
    regs.rdi = UINT64 (0x40069d);
    puts (/*"underflow"*/ (const char *) 0x40069d);
    regs.rax = UINT64 (0xffffffff); // -1
    return int (regs.rax);
   }
  }
 }
}
```

(d) Twincode of the above assembly code

Figure 7. Example assembly code, corresponding CFG, ETG, and twincode depicting some of trace extraction runtime challenges such as requirements for the management of the memory consumption, constraint simplification, and other abstractions.

For example, consider the program shown in Figure 7(a). It reads an input byte in line 18; compares it in lines 21 and 23 to filter out small and large values; computes a function over the input in lines 27-34; and prints the result in lines 35-37. Assume that the first execution goes through the left path shown in Figure 7(c) (false branches of lines 21 and 23 conditions). When the DFS finishes traversing the subgraph starting at $L37$, i.e., left child $\mathbb{L}$ of constraint $\mathbb{C}$ of node $L23$, it continues by asserting all constraints from the root node of ETG up to $\mathbb{C}$ and negation of constraint $\mathbb{L}$. The next execution of the program will produce a list of constraints starting with $L21$ and $L23$. Therefore, the new nodes which are added to the ETG are placed in the subgraph starting at right child of $\mathbb{C}$ and the DFS does not miss the ETG parts which are added during the search.

The CEE is realized by *twintool*, which is implemented within the *ldmbl* [26] architecture. The *ldmbl* architecture abstracts low-level OS and hardware-dependent implementation details based on the Pin [11] Dynamic Binary Instrumentation (DBI) framework in order to facilitate heavyweight instrumentation use cases such as concolic execution. Twintool is responsible for extracting a single trace from the given binary. It runs the input program concolically, instrumenting all assembly instructions, tracking memory, and register changes symbolically. This gray-box approach has the advantage that the intermediate conditions which have been inspected by the binary will become visible to twintool and can be recorded in the trace without any dependence on extra information, such as a specific compiler or the source code, which is normally unavailable during the malware analysis. The intel64 assembly instructions are represented by 1148 different mnemonics, each one having several modes for different operands. For example, the `SUB` mnemonic encodes 22 distinguished forms [27, p. 1459] such as `SUB AX, imm16` and `SUB r/m8, r8`. Deobfuscating a program needs all of the used forms of its assembly instructions to be supported in the twintool. To make it scalable, twintool implements a Generic Instrumentation Layer (GIL) as designed in the *ldmbl* [26] architecture. It automatically detects operand types (e.g., memory address) and their read/write sizes; organizes them within 40 generic categories (e.g., DstRegSrcImplicit); and instruments them using a set of Generic Analysis Routines (GARs). Thereafter, every invoked GAR finds the appropriate instruction analysis callback; wraps operands with proxy objects hiding different operand types/sizes; and invokes the callback to operate on them.

There are three proxy classes for memory, register, and constant values. Each one allows reading and writing (on mutable expressions), taking care of separation of pintool and input program memory, operation sizes, and overlapping locations. For example, the *RegisterResidentExpressionValueProxy* class updates EAX, AX, AH, and AL registers when an instruction updates the RAX register. This allows each instruction analysis routine to focus on its own specific logic while the GIL applies it to a bunch of lower-level assembly instruction models.

Figure 8 shows implementation of the `ADD` instruction analysis routine. It reads immutable/mutable source/destination expressions from proxy objects at lines 4 and 6, clones the destination expression at line 7, performs symbolic addition at line 9, and updates the destination (memory or register) using the proxy object at line 10. The abstraction provided by the twintool library makes it more extensible and easier to support new assembly instructions. For example, whenever an address is accessed for the first time, a new symbol is allocated for it. If an `ADD` instruction uses a new user input as its source operand, line 4 of Figure 8 allocates the new symbol for it. Afterwards, symbols may be copied to other addresses and/or undergo different operations (e.g., addition and multiplication). When symbols are created, some concrete values exist in them. Those initial values form the concrete state of the program and dictate which execution path will be followed within its running instance. Consequently, the program can be forced to follow another path by changing initial values of the symbols. Finally, line 11 updates the $EFLAGS$ register with an *AdditionOperationGroup* object, which can be queried in the following conditional instructions, e.g., conditional branches.

The last component of the framework (i.e., the SMTS) is realized using the CVC4 [28] library for solving symbolic constraints. It accepts a set of symbolic constraints and produces one of the three possible answers. It analyzes constraints to find a concrete solution satisfying all of them, simultaneously. If constraints are unsatisfiable, the CVC4 library either proves this fact or fails after a maximum analysis time. Several challenges arise here. First, the constraints which twintool has extracted from the binary may depend on the real inputs to the program and/or artificially added parts such as the VM interpreter code in a VO scenario or the decryptor/decompresser code for a packed malware. Next, these symbolic constraints may become more and more complicated; as a result, keeping them in the memory becomes a challenge. This makes solving of those constraints time consuming or infeasible within the given time limits.

Addressing these challenges requires more knowledge about the program inputs and how they affect the execution path. The first input category is user inputs, which are given as initial values to the program, and the second category is system inputs (including syscalls), which may depend on states which are hidden from the malware. User inputs (i.e., command line arguments of the program and environment variables) are placed in the stack and can be read by the program. Consequently, they change the execution path either (a) explicitly, via conditional branches (e.g., `je .L2`); or (b) implicitly, via calculated values (e.g., `movq -16(%rbp), %rax; jmp %rax`). The next influencing factor (i.e., syscalls) depends on the OS. This includes reading from a file, network communication, or even reading from the standard input. A typical syscall receives some arguments from the caller program (e.g., a buffer given to "read(fd, buf, len)") and completes its logic according to the given arguments and other parts of its memory (state of the caller process, in general, such as the open "fd" file descriptor) as well as the internal state of the kernel itself (e.g., the mounted file systems). Finally, syscall may change any parts of the caller's memory and/or registers.

On the one hand, the user and system inputs are captured as symbols. On the other hand, obfuscator related data items such as artificial arithmetic operations are treated as constants. For example, the virtual opcodes which encode the text

```
1  void InstructionSymbolicExecuter::addAnalysisRoutine (const MutableExpressionValueProxy &dst, const ExpressionValueProxy &src) {
2   edu::sharif::twinner::trace::Trace *trace = getTrace ();
3   edu::sharif::twinner::util::Logger::loquacious () << "addAnalysisRoutine(...)\n\tgetting src exp...";
4   const edu::sharif::twinner::trace::Expression *srcexp = getExpression (src, trace);
5   edu::sharif::twinner::util::Logger::loquacious () << "\tgetting dst exp...";
6   const edu::sharif::twinner::trace::Expression *dstexpOrig = getExpression (dst, trace);
7   edu::sharif::twinner::trace::Expression *dstexp = dstexpOrig->clone ();
8   edu::sharif::twinner::util::Logger::loquacious () << "\tbinary operation...";
9   dstexp->add (srcexp);
10  setExpression (dst, trace, dstexp);
11  eflags.setFlags (new edu::sharif::twinner::twintool::operationgroup::AdditionOperationGroup (dstexpOrig, srcexp));
12  edu::sharif::twinner::util::Logger::loquacious () << "\tdone\n";
13 }
```

Figure 8. Implementation of the ADD assembly instruction analysis routine.

section of the malware are the same in all executions. Although the execution path depends on both the virtual opcodes and the program inputs, the opcodes may not be changed based on the user interactions or the network and file system states. Consequently, simplifying constraints and transformation expressions based on the symbols is enough to automatically remove any dependence on those obfuscator related details.

Another example is shown in Figure 7(b). The loop created by node $L33$ is totally removed in Figure 7(c) because it depends on `-8(%rbp)`, which is initialized in line 26 of Figure 7(a) as a constant. This will not help the obfuscator to initialize it as a function of user inputs and then, cancel their effects in the comparison of line 32, because simplification of the constraints can simplify and remove the artificial dependency. A more concerning challenge is how these obfuscations affect the memory used for keeping transformations and constraint symbolic expressions. As an example, consider lines 28-33 in Figure 7(a), in which the value of `%eax` register, e.g., symbolic input s, is added to itself repeatedly to produce $s+s$, $s+s+s+s$, etc. growing the required memory exponentially. To overcome this challenge, twintool simplifies symbolic expressions as they are captured, instead of keeping them up to the end of execution to be processed in SMTS. Thus, the expression is kept as $2s$, $4s$, etc. with constant memory footprint.

The twincode generated by the described assembly code is shown in Figure 7(d). Lines 9-10 check the stack location and the program arguments on the stack; lines 11-12 check for the input interval leading to the execution of part $L37$ of the ETG, as shown in Figure 7(c); line 16 prints the input multiplied by 1024 (i.e., 0x400); and lines 22 and 28 print messages about overflow/underflow scenarios of the original program. Some of the initialization parts of Figure 7(d) are replaced with dots for sake of clarity. The complete compilable version can be retrieved from the test folder within the twinner repository.

5. Evaluation

This section evaluates Twinner from the practical point of view. For this purpose, Twinner is tested against the VO technique to observe the practical quality of the deobfuscation results. The VO replaces the entire text section of the program with an interpreter and it contains the classic obfuscations. That is, evaluation of the VO-protection scenario subsumes the deobfuscation of classical methods such as packing and encryption. Additionally, the set of SPEC [29] test programs is instrumented to examine the performance of a trace extraction run for large and complex real-world programs.

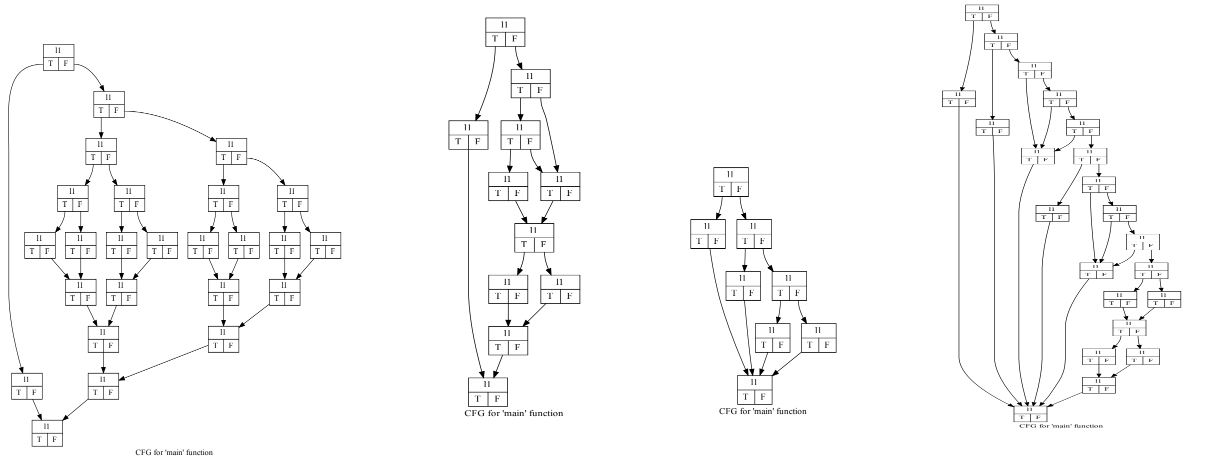
5.1. Effectiveness

To measure the similarity of the original program and the deobfuscated twincode, we can compare their ETGs. The structure of ETG is preserved while being encoded as a twincode. ETG is an appropriate metric, because it encodes the behavioral model which has been learned by analysis of the given binary. It also corresponds with the CFG of its twincode counterpart while CFG of the original program can be altered and replaced completely by obfuscation transformations. VO is one of the most complicated methods for obfuscating an arbitrary program. In this technique, a random instance of a template language is selected; the program is compiled to it and placed in the data section; and the entire text section is replaced by a virtual machine interpretor generated to be able to parse that random template language instance. Thus, CFG of the obfuscated program is completely independent of the original program and it can be examined as a difficult test scenario. If the CFG of the original program can be recovered in the structure of the resulting twincode, which can be seen as the resulting ETG, it shows that obfuscation effect has been cancelled.

In this section, we will VO-protect 4 programs with a variety of CFGs as the first step to obtain similarly incomprehensible and indistinguishable binaries, which only differ in the pseudorandom data sections. Afterwards, VO-protected programs are reversed and deobfuscated to obtain their execution trace graphs. Finally, similarity of ETGs of the deobfuscated programs and the corresponding original programs is measured according to Definition 7 in order to quantify effectiveness of the Twinner deobfuscation process.

Definition 7 (ETG similarity): *Similarity of two ETG instances, shown as $g_1 = (V_1, E_1)$ and $g_2 = (V_2, E_2)$ graphs, where $|V_1| \leq |V_2|$, is given by δ_{g_1,g_2} and defined as follows:*

- $\mathbb{S}_G = \{H = (V_H, E_H) | \exists V_h \subseteq V_H . H[V_h] \cong G\}$ *is the set of H graphs which have an induced subgraph $H[V_h]$ being the isomorph of the given G graph;*
- $\mathbb{S}_{g_1,g_2} = \mathbb{S}_{g_1} \cap \mathbb{S}_{g_2}$ *is the set of all graphs which have some induced subgraphs being the isomorph of the given g_1 and g_2;*
- $\hat{\mathbb{S}}_{g_1,g_2} = (\hat{V}, \hat{E}) \in \mathbb{S}_{g_1,g_2} \wedge \forall G = (V_G, E_G) \in \mathbb{S}_{g_1,g_2} . |V_G| \geq |\hat{V}|$ *indicates the supremum graph of the g_1 and g_2 graphs, which is a member of $\mathbb{S}_{g_1,g_2}$ set and has the minimum number of vertices among all members of the set;*
- *The similarity of the g_1 and g_2 graphs is defined as $\delta_{g_1,g_2} = 1 - \frac{|\hat{V}| - |V_1|}{|\hat{V}|}$ while $\delta_{g_1,g_2} = 1$ shows identical graphs and $\delta_{g_1,g_2} = 0$ shows the minimum normalized similarity among them.*



(a) P0 with nine main execution paths (b) P1 with two consecutive conditions (c) P2 with three exceptional branches (d) P3 with a mix of exceptional and merging paths

Figure 9. Control flow graphs of test input programs set.

Table 2. Virtual opcodes of the VO-protection interpreter.

Opcode	Description
0x00	Reads an argument and calls exit syscall on it.
0x01	Reads two args and a comparison code, applies one of six comparison operators on them to select between then/else parts, reads two jump offsets and use one of them to execute the then/else part of the conditional block.
0x02	Reads a format string and calls printf on it.
0x03	Reads an offset and jumps to it unconditionally.
0x04	Reads two pointer args, compares their corresponding C strings by calling strcmp, and stores the result in the aux variable.

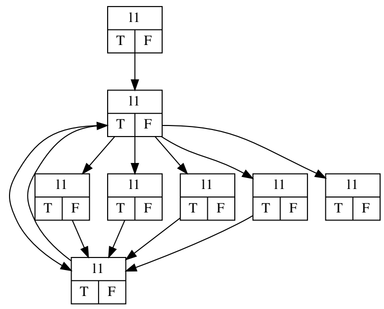
The test input programs have no specific importance. It is only required to select programs with different behaviors to clearly show how much the VO-protection eliminates their differences and how much the ETG restores those eliminated features. Figure 9 shows the CFGs of the selected four input programs. The first program, depicted in Figure 9(a), has 9 parallel execution paths. The first path is run when the input arguments are not well-formed. The other 8 paths are selected based on the three code characters in the program argument and an appropriate message is printed in each path. The next program, depicted in Figure 9(b), has two main execution paths one of which consists of 2 consecutive conditional blocks. This test can be used to examine how the repeated code sequences of the second conditional block are reused during the analysis of different paths of the first conditional block. The P2 program, shown in Figure 9(c), has a main execution path with three exceptional branches. Finally, Figure 9(d) depicts the P3 program with a mix of exceptional branches and consecutive merging paths.

The next step of evaluation is to obtain two artifacts from each given input test program. One of them is a VO-protected binary, which can be used as the input to the deobfuscation process. The other artifact is the ETG of the given test program without any obfuscation. This initial ETG can be compared with the result of the deobfuscation process to determine how much the deobfuscation effect has been removed successfully. To VO-protect each one of the test programs, they are compiled for a virtual language with 5 primitive opcodes. The interpreter of this language is shown in Figure 10(a) and its virtual opcodes are described in Table 2. The program itself is encoded in the *program_text*, followed by the *ptr* variable at runtime. In an infinite loop containing a *switch-case*, *ptr* is inspected to select one of the virtual opcodes and perform the corresponding conditional/unconditional jumps, printing operation, etc.

The CFG of the resulting VO-protected code is shown in Figure 10(b), which is clearly independent of the initial programs. The top node in Figure 10(b) corresponds to the beginning of interpreter, which calls the switch (line 19 in Figure 10(a)). The second node of CFG can jump to five destinations, which correspond

```c
int aux; // for intermediate values such as ret value

int end_of_execution(const char *&ptr);
void printf_command(const char *&ptr);
void strcmp_command (const char *&ptr);
void jump_command(const char *&ptr);
void if_then_else_command(const char *&ptr);
bool do_comparison(const char *&ptr, int arg0, int arg1);
int get_argument(const char *&ptr);
const char *init_program(int *argcptr, char *argv[]);

...

#define program_text "\x01\x01\xff\xff...\x00\x00\x02\xff↩
      \xff\xff\xff\xff\xff\xff\xff\x00\x02\x00\x00\x00\x00"

int main (int argc, char *argv[]) {
  const char *ptr = init_program (&argc, argv);
  for (;;) {
    switch (*ptr) {
      case 0x00:
        return end_of_execution (ptr);
      case 0x01:
        if_then_else_command (ptr);
        break;
      case 0x02:
        printf_command (ptr);
        break;
      case 0x03:
        jump_command (ptr);
        break;
      case 0x04:
        strcmp_command (ptr);
        break;
    }
  }
}
```

(a) Parts of the interpretor code



(b) Corresponding control flow graph

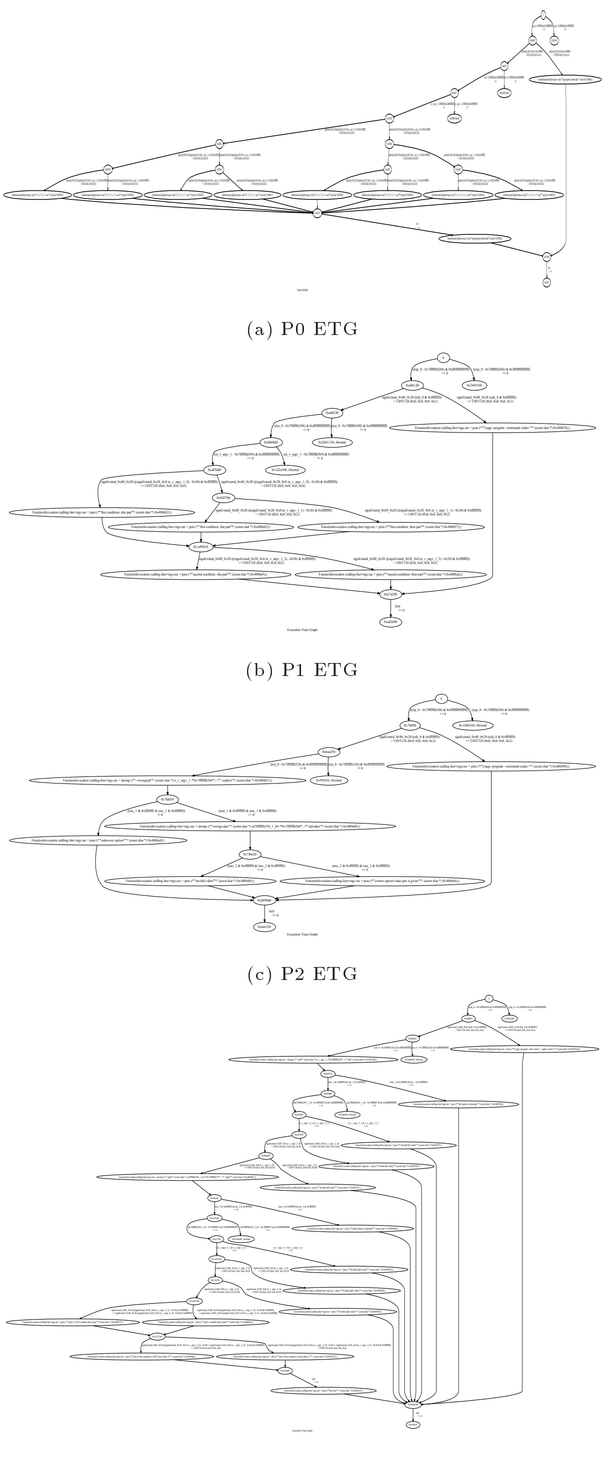
Figure 10. The interpreter used for virtualization obfuscation protection of test programs.

to five virtual opcodes. The first destination on the right belongs to *end_of_execution* opcode and hence, halts the interpreter. The second destination on the right in Figure 10(b) corresponds to *if_then_else* virtual opcode, which reads five arguments (two operands to be compared, a comparison code to determine the comparison operator, and two offsets for then/else parts of the conditional jump). The following two nodes correspond to the *printf* and *unconditional jump*, respectively, which call printf high-level function and change the *ptr* variable according to the jump offset. The last node corresponds to the *strcmp* function, which reads two pointer arguments, compares the two strings that are found at the given addresses, and stores the result of the comparison in the *aux* variable for the next checks. This *aux* variable can be accessed as an encoded argument by the following virtual operations. The four nodes corresponding to the last four cases of the *switch* statement merge into the bottom-most node in Figure 10(b) and jump back to the loop header node. Afterwards, they continue with interpretation of the next virtual operation.

All programs are mapped onto exactly the same interpreter with the same CFG and the difference between four given programs is limited to the contents of the *program_text* string. This makes all obfuscated binaries syntactically the same, while their different runtime behaviors are preserved. Now, there are two binaries for each test program; one without any protection and one with VO-protection. Analyzing them by Twinner to obtain their corresponding twincode and ETG leads to a pair of graphs for each program. Figure 11 shows ETG of the given input binaries and their corresponding deobfuscated versions are depicted in Figure 12. Comparing Figure 11(a)-(d) with their counterparts in Figure 12(a)-(d) side by side, it is clear that each ETG is fully recovered.

For example, consider the ETG of the obfuscated version of P1 program, which is depicted in Figure 12(b), and its corresponding pre-obfuscation ETG, which is shown in Figure 11(b). Both graphs consist of three main parts. The first part, which branches from the main execution path on the right side of figure, checks for the correct number of arguments. This part corresponds to the left-most path in Figure 9(b). The second part branches into three scenarios and prints the "*first-else-part* " message in two scenarios and the "*first-then-part* " in the last scenario. All the three paths are correctly merged before reaching the last part of the program. These three branching scenarios correspond to the two expressions of which the logical conjunction has been checked as shown in the top-right side of Figure 9(b). The third part prints two *then/else-part* messages, merges similar to the bottom-right side of Figure 9(b), and then, joins the first part (for checking arguments). Comparing the output ETG in Figure 12(b) with the input ETG in Figure 11(b) shows the complete recovery of all scenarios from the VO-protected binary.

Also, twincodes of these four programs are pro-



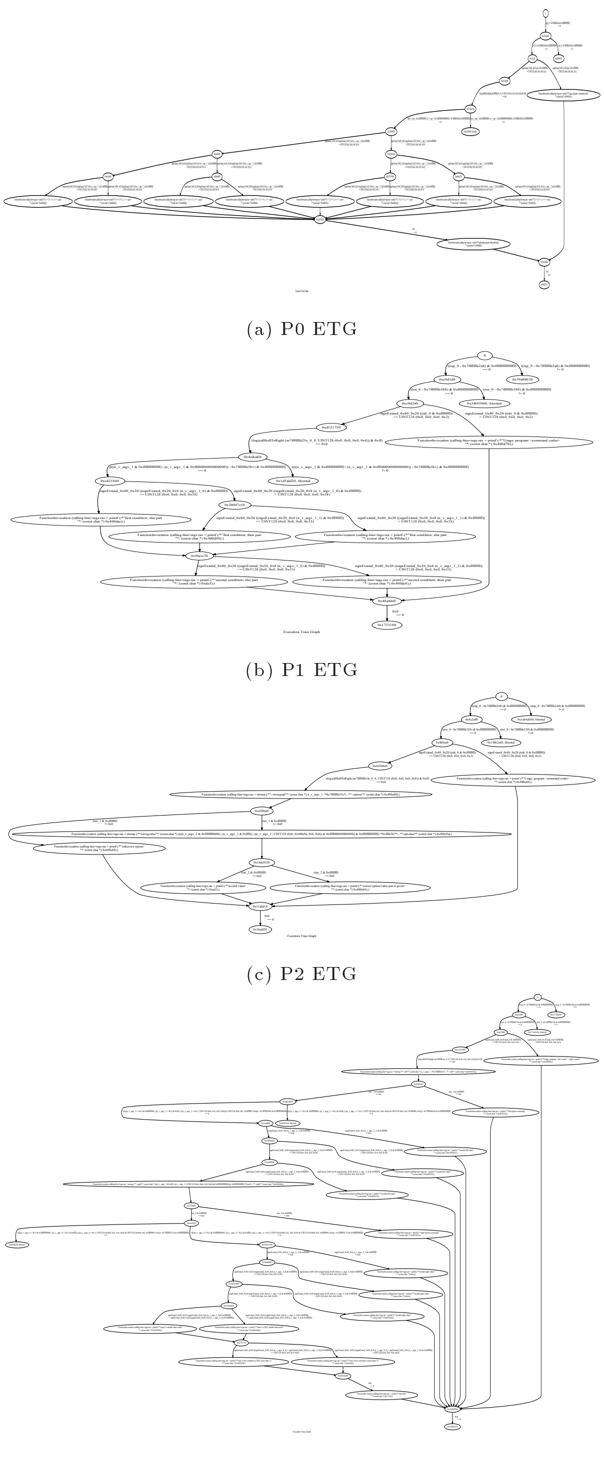
(a) P0 ETG

(b) P1 ETG

(c) P2 ETG

(d) P3 ETG

Figure 11. Execution trace graphs of input binaries before obfuscating them.



(a) P0 ETG

(b) P1 ETG

(c) P2 ETG

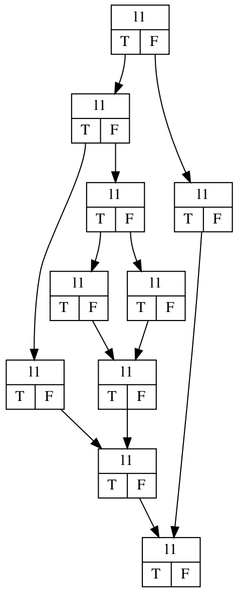
(d) P3 ETG

Figure 12. Execution trace graphs of output binaries which were VO-protected.

duced, which can be used for further analysis. Figure 13(a) shows the twincode obtained by analysis of the VO-protected version of P2. The first condition (line 7) has renamed the `argc` to `rdi_0` and checks for the correct number of arguments. Line 15 checks for the presence of a valid option and line 23 checks for the valid option value. The CFG of this code is depicted in Figure 13(b), which corresponds to its ETG shown in Figure 12(c). Looking at the CFG in Figure 13(b) from top to bottom, the first node corresponds to the condition of line 7, comparing `rdi_0` with number three. If there are less than

```
1  int main (int argc, char *argv[]) {
2   struct RegistersSet regs;
3   SAVE_REGISTERS (regs);
4   const UINT64 rax_0 = regs.rax, rbx_0 = regs.rbx, ...;
5   const UINT128 xmm0_0 = UINT128 (regs.xmm0), ...;
6   const UINT64 m7fffffffe1a0_0_64 = *((UINT64 *) 0x7fffffffe1a0);
7   if ((signExtend_0x80_0x20 ((UINT64 (rdi_0) & 0xffffffff)) >= UINT128 (0x0, 0x0, 0x0, 0x3))) {
8    /*Memory Changes*/
9    *((UINT64 *) 0x400ee8) = UINT64 (0x10000000302ffff);
10   *((UINT64 *) 0x602180) = UINT64 (0xffffff020000000f);
11   *((UINT64 *) 0x7fffffffe210) = UINT64 (((rsp_0 - 0x8) & 0xffffffffffffffff) /*0x7fffffffe240*/);
12   *((UINT64 *) 0x7fffffffe228) = UINT64 ((((rdi_0 & 0xffffffff) << 0x20) | m7fffffffe228_0_32));
13   /*Registers Changes*/
14   regs.rcx = UINT64 (0x14);
15   regs.rax = strcmp (/*"--wrongopt"*/ (const char *) argv[1], /*"--option"*/ (const char *) 0x400e80);
16   const UINT64 rax_1 = regs.rax, rbx_1 = regs.rbx, ...;
17   const UINT128 xmm0_1 = UINT128 (regs.xmm0), ...;
18   const UINT64 m7fffffffe1a0_1_64 = *((UINT64 *) 0x7fffffffe1a0);
19   const UINT32 m7fffffffe1b0_1_32 = *((UINT32 *) 0x7fffffffe1b0);
20   if (((UINT64 (rax_1) & 0xffffffff) /*0x8*/ != 0)) {
21    regs.rax = printf (/*"unknown option!\n"*/ (const char *) 0x400e89);
22   } else {
23    regs.rax = strcmp (/*"wrongvalue"*/ (const char *) argv[2], /*"optvalue"*/ (const char *) 0x400e9a);
24    const UINT64 rax_2 = regs.rax, rbx_2 = regs.rbx, ...;
25    const UINT128 xmm0_2 = UINT128 (regs.xmm0), ...;
26    if (((UINT64 (rax_2) & 0xffffffff) /*0x8*/ != 0)) {
27     *((UINT64 *) 0x7fffffffe240) = UINT64 (rbp_0 /*0x0*/);
28     regs.rax = printf (/*"invalid value!\n"*/ (const char *) 0xea3);
29    } else {
30     regs.rax = printf (/*"correct option/value pair is given!\n"*/ (const char *) 0x400eb8);
31     const UINT64 rax_3 = regs.rax, ...;
32     const UINT128 xmm0_3 = UINT128 (regs.xmm0), ...;
33    }
34   }
35  } else {
36   regs.rax = printf (/*"Usage: program <command codes>\n"*/ (const char *) 0x400e60);
37  }
38 }
```

(a) Main part of twincode generated by analyzing the VO protected version of P2



(b) Corresponding CFG

Figure 13. Sample twincode and its corresponding CFG.

three arguments, the right-most path of CFG will be followed, which leads to the execution of line 36 and printing the program usage message. Otherwise, the second node, which corresponds to the operations of lines 15-20, will be invoked. This portion compares `argv[1]` with the hard-coded string of "--option" (the concrete value of `argv[1]` is also given in the comment as a hint). If the name of a wrong option has been used, the left-most path of CFG will be followed and hence, line 21 prints the corresponding error message. The last condition, which is located at the center of Figure 13(b), corresponds to the check in lines 23-26, comparing `argv[2]` with the "optvalue" string and selecting one between line 28 (error case, right branch in CFG) and line 30 (target case, left branch in CFG) to execute. Except for the arrangement of nodes, the CFG in Figure 13(b) has a main execution path (in which all conditions are evaluated to be true) from which three exceptional paths deviate, similar to the original CFG of P2 depicted in Figure 9(c).

The input/output ETGs can be compared pairwise according to Definition 7 in order to assign a similarity measure to each pair. Table 3 aggregates these calculated similarities. The diagonal entries in Table 3 are all relatively higher than non-diagonal entries. The diagonal entries with values close to one indicate high restoration of the initial ETG figures. Also, the similarity of unmatched graphs is reduced quickly as each graph becomes more complicated. For small programs, pre/post-obfuscation graphs are separated by the similarity value of 0.7. For larger programs, the similarity drops to below 0.5 while all diagonal entries stay higher than the similarity value of 0.88.

As this proof of concept implementation can be extended to analyze any other application only by supporting its possibly different assembly instructions for recognizing and formulating the calculated symbolic expressions, it can be deduced that VO can be automatically reversed on other protected programs similar to the mentioned example. It is worthy to note that this framework does not make any assumptions about the structure of the used virtual machine to produce the twincode. For example, the VM can eliminate the **ptr** (i.e., VPC) and connect different pieces of the virtualized program directly together

Table 3. Similarity degree of pre/post-obfuscation ETG figures.

VO-protected / Pre-obfuscation	P0	P1	P2	P3
P0′	**0.96**	0.56	0.47	0.5
P1′	0.61	**0.90**	0.63	0.44
P2′	0.5	0.70	**0.88**	0.39
P3′	0.52	0.41	0.37	**0.95**

Table 4. Execution times and relative overheads for SPEC programs analysis.

Program	Native[a] (ms)	Count[b] (s)	IAO[c] (x)	Twintool[d] (s)	TAO[e] (x)
astar	3.602	2.496	691.8	5.999	1.4
bzip2	2.961	2.514	848.2	3.768	0.5
gcc	8.328	25.312	3038.4	89.03	2.5
gnugo	48.816	23.863	487.8	41.64	0.7
grover	2.39	2.222	928.7	10.968	3.9
hmmer	5.135	3.828	744.5	11.015	1.9
jm-h264ref	84.939	8.977	104.7	136.287	14.2
omnetpp	36.468	45.752	1253.6	1543.674	32.7
perl	5.221	28.234	5406.8	28.793	0.02
sjeng	85.513	3.814	43.6	50.094	12.1
xalan	6.058	15.2	2508.1	70.982	3.7

[a]Native: The execution times in milliseconds without any instrumentation.

[b]Count: The execution times in seconds when an instruction counting pintool is used.

[c]IAO: The Instruction-counting Added Overhead as a factor of native execution times.

[d]Twintool: The measured times when the Twintool pintool is used to instrument each SPEC test program.

[e]TAO: The Twintool-added overhead as a factor of instruction counting pintool execution times.

(e.g., similar to what is done in the jump oriented programming [30]) without causing any change in its corresponding twincode.

The generated twincode can be used in static analysis instead of the obfuscated code to directly obtain results about the original program. For example, the CFG in Figure 13(b) is drawn by static analysis of the corresponding code in Figure 13(a) using the *LLVM* [31] `-dot-cfg` pass.

5.2. Performance

In order to measure the execution time overhead of the twintool analysis runs, a set of complicated programs is selected based on the SPEC cpu test to obtain a real-world estimate of the average implied overhead. Latest versions of these programs, which are released for the Ubuntu server 14.04, are used for performance tests. The used evaluation scripts and program inputs are available on the *evaluation* branch of the Twinner git repository. Moreover, all experiments have been executed in different scenarios including a non-instrumented run for native programs, an instrumented run with instruction counting analysis routines (to find out about the minimum possible instrumentation overhead), and a run with twintool instrumentation to observe the relative overheads.

Experiments were performed on a single-core QEMU/KVM virtual machine with 8GB RAM running Ubuntu server 14.04 with kernel 3.19.0-25-generic x86 64 hosted on a quad-core Intel i7-6700HQ machine. Each experiment scenario has been repeated as many times as required according to the central limit theorem [32] in order to limit the maximum error of the reported mean execution time to at most 0.5% with confidence level of 95%.

The obtained results are aggregated in Table 4 of which the second column reports the native execution time, the third and fourth columns indicate the baseline overhead caused by using the Pin dynamic binary instrumentation framework, and the last two columns indicate the overhead caused by the twintool itself. As indicated in the fourth and sixth columns of Table 4, the Pin framework slows down the overall execution time by an order of thousand times, but the twintool-added overhead (relative to the instruction counting instrumentation) stays relatively small even for very complicated programs such as the gcc.

6. Discussion

Given the evaluation results in the previous section, in the following, the possible deobfuscation challenges are discussed; how they are mitigated in the Twinner framework is expressed; and an objective comparison with previous works is presented.

One of the hardest scenarios for SMTS is an opaque predicate (i.e., an always true/false condition), which is so complicated that SMTS cannot reason about its negated constraint within a short time span. For always-true constraints, the logic of the code is captured in the first trace containing it. Also, its negated constraint is assumed to be unsatisfiable, which is the case. For always-false constraints, the code is never executed and the SMTS cannot find any concrete input to drive the program through it. Thus, this path is assumed to be deadcode, which is the case as well. However, when SMTS cannot find an answer for the queried constraints (such as a one-way hashing function) within the provided deadline, the corresponding path is marked as deadcode and remains unexplored. Although it is possible to enforce the execution through that possible deadcode by temporarily modifying the program assembly instructions in the memory, this makes the concrete state of the program invalid and increases the number of paths to be analyzed. One strategy, which can be inspected in the future research, is to prioritize the execution paths, giving lower priority to the suspected paths and allocating available resources for their analysis according to their priorities. This may lead to a more balanced answer for the trade-off between full analysis of all possible branches and minimizing the analysis time for reachable code paths.

Nevertheless, the current approach to focusing on the constraints, which are solvable within the given time limits as described in the following example, is applicable to most practical scenarios. Consider a bot which tries to retrieve a command from its C&C server and perform the corresponding order (e.g., a denial of service attack). The bot may download new code modules and execute them. In that case, the executed code is not available at the analysis time, i.e., before contacting a real C&C server. But, for command codes which lead to execution of existing malware modules, it is possible to extract, analyze, and generate the corresponding twincode only by communicating with an arbitrary network server, not necessarily the real C&C server. This is in contrast to methods such as [33] that analyze a real traffic trace for modeling and simulating the C&C server for the bot.

The bot program needs cooperation of the OS for receiving the Protocol Data Unit (PDU) from the corresponding socket (e.g., using "recv" function). When control returns to the user-space, the bot can be notified about syscall operations by reading its own memory (e.g., the buffer passed to "recv") and/or looking at registers (e.g., the return value depicting the number of read bytes). By instrumenting all assembly instructions (including those which are generated dynamically by the malware), it is possible to preempt when an address is read for the first time. Instantiated symbols can be written at other addresses (copied) or undergo arithmetical (e.g., `addq $4, -16(%rbp)`) and logical operations (e.g., `cmpl $0, -4(%rbp)`) while their expected concrete values are being inspected by every operation. At each memory address or register, a symbolic formula is being kept in addition to its normal concrete value. When all executed instructions are instrumented, the acquired symbolic expressions for all addresses have to match with their concrete values. However, syscall, executed in the kernel-space, is out of the instrumentation scope. Therefore, concrete/symbolic values can mismatch after a syscall. When this occurs (e.g., contents of a file are read in a buffer), new symbols are required to capture the changed concrete values. Thus, all bytes of the bot-C&C communication PDU are controlled by symbols and can be manipulated by twintool at symbol instantiation time.

For example, an arbitrary network server sends command code 0xB to the bot. The bot checks the code and terminates, because it expects a 4 bytes long PDU. This constraint is captured by twintool symbolically and solved to obtain a four bytes long input for use in the next analysis round. Then, the network server sends the 0xB code again, but the buffer is modified by twintool on the fly to be seen as the calculated four bytes value. The bot continues execution and tries to invoke a function based on the read code. Although the network server program does not know about the format of the message, which is expected by the bot, twintool can automatically deduce it from the program itself iteratively. Possible calling targets can be determined similarly.

Another notable challenge is the high number of assembly instructions and how they can be instrumented correctly and efficiently. For this purpose, the *ldmbl* [26] architecture is used, which abstracts the APIs provided by Pin [11] in two layers. In one layer (GIL), it minimizes the instrumentation calls and in another layer (GARL), it minimizes the number of required analysis routines through a series of proxy classes. These abstractions help twintool to be implemented in fewer lines of code while maintaining its efficiency as evaluated in Section 5.2. Pin Dynamic Binary Instrumentation (DBI) framework uses a Just-In-Time (JIT) compilation mechanism internally for transferring visited instructions to a code cache region before execution, Therefore, the instructions cannot affect the instrumentation without being monitored by some prior instrumentation. Consequently, different obfuscations of a program cannot affect the instrumentation except by exploiting some vulnerability in the instrumentation framework. Although it is impossible to guarantee the absence of vulnerabilities in Pin DBI framework, like other software/hardware components, such a vulnerability (if any) should be addressed in the underlying DBI framework and it is out of the adversarial model of twintool.

Table 5 summarizes a comparison of the mentioned related studies and the Twinner framework. Each cell is marked with $\surd$ if the solution mentioned in the respective row is able to outperform the obfuscation technique noted in its column. If it cannot reverse the transformation, the cell is marked with $\times$. On the other hand, if it cannot be reversed completely, but it is partially considered in the solution, the mark is used. The $\sim$ mark indicates that the mentioned feature is inapplicable to the solution.

Table 5. Comparison of analysis and deobfuscation solutions. $\surd$ shows that the mentioned solution can overcome the obfuscation technique, $\times$ represents an unsupported feature, - indicates inapplicable features, and $\sim$ is for obfuscation features that cannot be reversed completely, but are partially considered.

Feature/solution	**X-Force [16]**	**Yadegari et al. [5]**	**ROPMEMU [17]**	**Rolles [3]**	**Rotalume [6]**	**Coogan [10]**	**Kinder [4]**	**Twinner**
Code packing	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Constant obfuscation	$\times$	$\surd$	$\times$	$\surd$	$\times$	$\sim$	$\times$	$\surd$
Logical obfuscation	$\times$	$\sim$	$\times$	$\surd$	$\times$	$\sim$	$\times$	$\surd$
Branch obfuscation	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\times$	$\surd$
Analysis type: Static (S), Dynamic (D), or Both (B)	D	D	D	D	D	D	S	B
Source code presentation	$\times$	$\times$	$\times$	$\sim$	$\times$	$\times$	$\times$	$\surd$
Deadcode elimination	$\times$	$\sim$	$\times$	$\sim$	$\times$	$\times$	$\times$	$\surd$
VO-protection	$\times$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
VO's VM language independence	-	$\surd$	-	$\times$	$\surd$	$\surd$	$\surd$	$\surd$
VO's VM metamodel independence	-	$\surd$	-	$\times$	$\times$	$\surd$	$\times$	$\surd$
Coverage/completeness	$\surd$	$\sim$	$\times$	$\times$	$\times$	$\times$	$\sim$	$\surd$
Loader analysis	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\surd$	$\times$	$\surd$
Multi-thread programs	$\sim$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Anti-disassembly	$\surd$	$\surd$	$\surd$	$\surd$	$\sim$	$\surd$	$\times$	$\surd$
Anti-debugging	$\surd$	$\times$	$\times$	$\times$	$\times$	$\times$	$\surd$	$\surd$

All solutions support packing, except ROPMEMU [17], which starts its work with a memory dump assuming that ROP chains exist in that dump. Three solutions [5,3,10] consider arithmetical and logical obfuscations by means of compiler transformations and/or symbolic expression equivalence tests. The next technique is branch obfuscation, which uses emulated jumps and function calls or complicates the CFG with opaque backward branches. Most dynamic solutions can bypass this technique by sacrificing the analysis completeness. A counter-example is X-Force [16], which exhibits exponential execution time by forcing analysis of deadcode portions.

Except Twinner and the Rolles [3] method, which find a correspondence between the used binary portions and the previously reverse-engineered VM parts, all the solutions consent to learn some model, such as CFG, and do not provide a source code representation that can be used as input to other analysis tools. Among the solutions that support the VO-protection scheme, Rolles [3] requires priori knowledge of the used VM. This knowledge needs to be updated per VM language instance. Sharif et al. [6] and Kinder [4] need a specific VM metamodel to detect as a binary pattern, and extract VM and the protected codes based on it. Yadegari et al. [5] and Coogan et al. [10] focused on some given execution traces and did not have a systematic approach to maximizing the code analysis coverage.

The other feature is related to the analysis of the program loader. Malware can hide with techniques such as ELF weird machines [19] out of the common executable area and run before beginning of the actual program. X-Force [16] and Twinner support program analysis from the entry point and hence, can detect such tricks. Also, Yadegari et al. [5] and Coogan et al. [10] employed Ether [34] low-level instruction traces and hence, could observe loader-encoded behavior. Other solutions require a previously recognized code region (e.g., ROP chains or VM interpretor routines) to start their analysis. A common shortcoming in all solutions is related to the analysis of multi-thread programs. Except X-Force [16], which serializes all threads by replacing thread creation API with a direct call to the thread entry function, all solutions follow the OS scheduler decisions about the order of threads.

Finally, the last two columns of Table 5 indicate resistance of the solutions against anti-assembly and anti- debugging techniques. Kinder [4], as a static analysis method, suffers more from the anti-disassembly, which stops it from beginning the analysis in the first step, while anti-debugging techniques can guide the analysis efforts of all solutions, except

Twinner, X-Force [16], and Kinder [4], to the benign code portions due to the lack of a code coverage maximization strategy.

7. Conclusion

This manuscript presented a framework for software deobfuscation, namely Twinner, which could dynamically analyze an arbitrary Windows/Linux executable program. The presented framework mapped the deobfuscation problem onto three components. The first component was a Concolic Execution Engine (CEE), which instrumented the given binary and captured its runtime behavior in a series of symbolic expressions and constraints. The second one was a Path Search Strategy (PSS), which learnt a behavioral model of the program by running it through different execution paths iteratively. In each run, an independent instance of CEE was employed to run the binary along a specific path and produce its corresponding trace object. The third component was a library for solving symbolic constraints (SMTS) used by PSS to find out candidate concrete values for symbols in order to guide CEE runs along the following unexplored paths.

Also, a proof of concept implementation of the proposed framework was presented and evaluated to measure the deobfuscation effectiveness and performance using different real-world programs. The proposed method was not dependent on any obfuscation process or structure of the obfuscated program. CEE, which was realized as twintool library, was implemented using the ldmbl architecture based on the Intel Pin. By instrumenting assembly instructions, it found out about memory symbols before their first use and hence, it could guide the program along a specific execution path by modifying their concrete values. PSS, which was realized as twinner, provided a command line interface for configuring the deobfuscation parameters and combined the traces received from twintool to update an Execution Trace Graph (ETG). In each round, the twinner selected the next execution path by DFS searching the ETG and obtained a list of constraints for satisfying all branches along the selected path. The CVC4 SMT solver was used to find a concrete solution to those constraints and feed the twintool.

As the code was analyzed concolically along all paths, anti-debugging techniques were not an obstacle and as the behavior was tracked symbolically, the complete functionality of the program was captured in the generated twincode. The concepts used, e.g., trace and guided executions, were formally defined and used to prove properties of the deobfuscation process. To evaluate their effectiveness, several programs were protected with VO and then, used as test inputs. ETG graphs of the analyzed programs were drawn and compared with each other to see how much details of the original programs were restored after deobfuscation. A graph similarity measure was defined for this purpose, which showed that the obtained ETGs after deobfuscation were considerably similar to the original CFGs and could distinguish test inputs with at least 18% margin. ETGs matched CFGs of the original programs, except for rearrangement of graph nodes, and were encoded as twincode in C language, which could be used for further analysis. To evaluate the performance of the deobfuscation process, the SPEC test programs were used. Analysis of complex programs such as gcc demonstrated that the additional overhead of twintool was in the order of 10 times, which was considerably lower than the additional overhead of Pin being in the order of 1000 times.

The presented deobfuscation framework can be used for different use cases, such as understanding the internal logic of a malware, e.g., DNG algorithm of a bot, which is impossible without deobfuscation; generation of comprehensive behavioral signatures for categorizing malware in related families; debugging the obfuscated benign software to examine correctness of the used obfuscation transformation; and analysis of close sourced and obfuscated software for the presence of possible backdoors.

References

1. Global Research & Analysis Team (GReAT), "Equation group: Questions and answers", Kaspersky Labs, https://securelist.com/files/2015/02/Equation_group_questions_and_answers.pdf Online. Retrieved on 25th Feb 2015.
2. sKyWIper Analysis Team "skywiper: A complex malware for targeted attacks", Tech. Rep., Laboratory of Cryptography and System Security (CrySyS Lab), Budapest University of Technology and Economics (2012).
3. Rolles, R. "Unpacking virtualization obfuscators", *3rd USENIX Conference on Offensive Technologies*, USENIX Association, pp. 1-1 (2009).
4. Kinder, J. "Towards static analysis of virtualization-obfuscated binaries", *19th Working Conference on Reverse Engineering (WCRE), IEEE*, pp. 61-70 (2012).
5. Yadegari, B., Johannesmeyer, B., Whitely, B., et al. "A generic approach to automatic deobfuscation of executable code", *2015 IEEE Symposium on Security and Privacy, IEEE*, pp. 674-691 (2015).
6. Sharif, M., Lanzi, A., Giffin, J., et al. "Automatic reverse engineering of malware emulators", *30th IEEE Symposium on Security and Privacy, IEEE*, pp. 94-109 (2009).
7. Newsome, J., Karp, B., and Song, D. "Polygraph: Automatically generating signatures for polymorphic worms", *IEEE Symposium on Security and Privacy, IEEE*, pp. 226-241 (2005).

8. Kang, M.G., Poosankam, P., and Yin, H. "Renovo: A hidden code extractor for packed executables", *2007 ACM Workshop on Recurring Malcode, ACM*, pp. 46-53 (2007).

9. Raber, J. and Laspe, E. "Deobfuscator: An automated approach to the identification and removal of code obfuscation", *14th Working Conference on Reverse Engineering, WCRE'07, IEEE Computer Society*, pp. 275-276 (2007).

10. Coogan, K., Lu, G., and Debray, S. "Deobfuscation of virtualization-obfuscated software: a semantics-based approach", *18th ACM Conference on Computer and Communications Security, ACM*, pp. 275-284 (2011).

11. Luk, C.K., Cohn, R., Muth, R., et al. "Pin: Building customized program analysis tools with dynamic instrumentation", *ACM SIGPLAN Notices*, **40**(6), pp. 190-200 (2005).

12. Sen, K. "Concolic testing", *22nd IEEE/ACM International Conference on Automated Software Engineering, ACM*, pp. 571-572 (2007).

13. Sen, K. "Concolic testing: a decade later (keynote)", *13th International Workshop on Dynamic Analysis, ACM*, pp. 1-1 (2015).

14. Brumley, D., Jager, I., Avgerinos, T., et al. "BAP: A binary analysis platform", *Computer Aided Verification*, Springer, pp. 463-469 (2011).

15. Shoshitaishvili, Y., Wang, R., Salls, C., et al. "Sok: (state of) the art of war: Offensive techniques in binary analysis", *IEEE Symposium on Security and Privacy (SP), IEEE*, pp. 138-157 (2016).

16. Peng, F., Deng, Z., Zhang, X., et al. "X-force: Force-executing binary programs for security applications", *2014 USENIX Security Symposium*, San Diego, CA, August (2014).

17. Graziano, M., Balzarotti, D., and Zidouemba, A. "ROPMEMU: A framework for the analysis of complex code-reuse attacks", *11th ACM on Asia Conference on Computer and Communications Security, ACM*, pp. 47-58 (2016).

18. Vanegue, J. "The weird machines in proof-carrying code", *Security and Privacy Workshops (SPW), 2014 IEEE*, IEEE, pp. 209-213 (2014).

19. Shapiro, R., Bratus, S., and Smith, S.W. ""Weird machines", in ELF: A Spotlight on the Underappreciated Metadata", *WOOT'13: Presented as part of the 7th USENIX Workshop on Offensive Technologies*, USENIX (2013).

20. Bangert, J., Bratus, S., Shapiro, R., et al. "The page-fault weird machine: lessons in instruction-less computation", *7th USENIX Workshop on Offensive Technologies, WOOT'13*, USENIX (2013).

21. Barrett, C. "SMT: Where do we go from here?", *12th International Workshop on Satisfiability Modulo Theories*, Available at: http://smt2014.it.uu.se/ (2014).

22. Fraser, G. and Arcuri, A. "Evosuite: automatic test suite generation for object-oriented software", *19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM*, pp. 416-419 (2011).

23. Shahamiri, S.R., Kadir, W.M.N.W., Ibrahim, S., et al. "An automated framework for software test oracle", *Information and Software Technology*, **53**(7), pp. 774-788 (2011).

24. Vanegue, J., Heelan, S., and Rolles, R. "SMT solvers in software security", *WOOT*, pp. 85-96 (2012).

25. http://ce.sharif.edu/~b_momeni/projects/twinner

26. Momeni, B. and Kharrazi, M. "LDMBL: An architecture for reducing code duplication in heavyweight binary instrumentations", *Software: Practice and Experience*, **48**(9), pp. 1642-1659 (2018).

27. Intel Corporation "Intel®64 and IA-32 architectures software developer's manual, combined volumes 1, 2ABC, 3ABC", Intel Corporation (2013).

28. Barrett, C., Conway, C.L., Deters, M., et al. "CVC4", *Computer Aided Verification*, Springer, pp. 171-177 (2011).

29. SPEC "SPEC CINT2006 benchmarks", Standard Performance Evaluation Corporation, https://spec.org/cpu2006/CINT2006/ (2006).

30. Bletsch, T., Jiang, X., Freeh, V.W., et al. "Jump-oriented programming: a new class of code-reuse attack", *6th ACM Symposium on Information, Computer and Communications Security, ACM*, pp. 30-40 (2011).

31. Lattner, C. and Adve, V. "LLVM: A compilation framework for lifelong program analysis & transformation", *International Symposium on Code Generation and Optimization, CGO, IEEE*, pp. 75-86 (2004).

32. Le Cam, L. "The central limit theorem around 1935", *Statistical Science*, **1**(1), pp. 78-91, Institute of Mathematical Statistics (Feb. 1986).

33. Graziano, M., Leita, C., and Balzarotti, D. "Towards network containment in malware analysis systems", *28th Annual Computer Security Applications Conference, ACM*, pp. 339-348 (2012).

34. Dinaburg, A., Royal, P., Sharif, M., et al. "Ether: malware analysis via hardware virtualization extensions", *15th ACM Conference on Computer and Communications Security, ACM*, pp. 51-62 (2008).

Biographies

Behnam Momeni received BSc and MSc degrees in Computer Engineering and Information Technology (with honors) from Sharif University of Technology, Tehran, Iran, in 2010 and 2012, respectively. He then joined the PhD program at Sharif University of Technology. He is currently a PhD candidate in Safety and Security in Software and Systems Laboratory (S4Lab), Department of Computer Engineering, Sharif University of Technology. His current research interests include information, operating system, and software security and techniques of software obfuscation and deobfuscation.

Mehdi Kharrazi received the BSc degree in Electrical Engineering from the City College of New York, New York, in 1999, and MSc and PhD degrees in Electrical Engineering from Polytechnic University, Brooklyn, NY, in 2002 and 2006, respectively. He is currently an Assistant Professor with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His current research interests include software, network, and multimedia security.