Twinner: A Framework for Automated Software Deobfuscation

Behnam Momeni, Mehdi Kharrazi*

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

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 then 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.

Keywords: Virtualization Obfuscation, Malware Analysis, Automated Deobfuscation, Twincode Generation

1. Introduction

Malware programs have evolved rapidly over the past decade, initially being developed for fun, now they are 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.

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 the 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 VO short for Virtualization Obfuscation [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

*Corresponding author (Office Phone: +98 21 6616 6627)

Email addresses: b_momeni@ce.sharif.edu (Behnam Momeni), kharrazi@sharif.edu (Mehdi Kharrazi)
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 to 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 depend on specific aspects of the obfuscator. Analyzing a VO protected malware, static solutions need to analyze the executable which is entirely replaced with the VO interpreter code. So 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 been changed. For example a solution which inspects executed branches to reconstruct the program CFG, although obtains a view of the VO interpreter CFG but misses the higher level CFG which has been formed among the virtual instructions. This may require the deobfuscator to put preciser assumptions about the VO strategy to focus on the hidden virtual program and so fail when those VO strategies or used template languages are changed.

There have been two general approaches to deobfuscate the code. Consider an image which has been broken into pieces like a jigsaw puzzle. One approach to solve the puzzle and restore the image is to examine how each individual piece can fit into another piece. For example the low-level obfuscation patterns 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 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 are not useful for learning their internal logic such as the DNG algorithm of a bot or hidden behavior such as a backdoor which remains inactive until a secret message is received. In other words and as 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 top or center of the puzzle together. They cannot generate any source code, detect/remove any deadcode, nor analyze hidden logics which may be executed if the malware environment was different.

This manuscript proposes a middle approach to the deobfuscation problem. As in the puzzle example, an alternative approach is to understand what the image was and 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, tracing all assembly instructions, and monitoring the binary (including changes in registers, memory, and invoked syscalls). And to decrease the required resources for a fixed degree of deobfuscation, concrete-symbolic (concolic) execution is employed where 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, this keeps an open opportunity for understanding the internal code logic. 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 presence of concealed backdoor are a few possible use cases of this framework.
The resulting twincode has a CFG similar to 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 is 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, 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. For example solutions such as [3, 5] lead to high quality results but are always an 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 such as [6, 10] work in presence of new obfuscation techniques but produce lower quality results. Some frameworks such as [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 also 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 works 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 into symbolic functions. It assumes that the VM interpreter is reverse engineered once and so 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 are 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), (b) or the theorem prover cannot find the equivalent symbolic function of the executed part among reverse engineered parts.

The 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 on 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] use taint propagation to identify data flows during an execution and then apply a series of semantics preserving transformations to simplify that logic. For this purpose they employ the Ether to obtain code traces and then apply five simplification methods on traces. Finally, they construct 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 to remove some of redundancies.

Methods from the second category resist upon more complicated obfuscations but need to sacrifice their output completeness. The Coogan [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 Rotalume [6] method 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 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. Based on it, Rotalume 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 worth noting is using weird machines. Weird machines [18] are machines which 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 as shown in [19]. Since our method starts instrumenting the input program before 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 as described in [20]. That weird machine executes the malware completely while no single instruction is fetched to be executed yet.

3. Framework

Dynamic analysis, even though a powerful approach for analyzing binaries 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. And 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, affecting the analysis effectiveness.

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. So 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. But still those addresses have to be used instead of that common variable. So 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. By 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 a lot of 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 \(2\alpha\) expression.

All such activities are monitored symbolically to aggregate obtained information and form an abstract behavioral model. The program behavior is simplified in two phases. First, while capturing each execution trace, several simplification rules are applied on 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 and also 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 was 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 top of the Figure 2 in order to find the required concrete inputs to inspect the next program paths and also to simplify the obtained mathematical model.

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 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.

\[
\text{statement} \ s ::= \text{confined}\ c \mid \text{illuminated}\ i
\]  

(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 make 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 so can lead to 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 the $E$ (bottom-left) shows the current state of the program and $s$ indicates the current statement. For execution of $s$, a series of computations are carried (top) and consequently the program state is updated to $E'$ (bottom-right). Next statement is shown by $s'$.

$$\text{computations} \quad \frac{E, s \rightarrow E', s'}{\text{Name-of-Operation}} \quad (2)$$

First rule in Figure 3 models the execution of a confined statement $c$ while the concrete program state is denoted by $S$. The $\text{Behavior}(c) : mS \rightarrow mS$ is a function from the current memory state ($S$) to the next state, encoding how the statement $c$ updates the program state. The memory state $mS := \mathbb{N} \rightarrow \mathbb{N}$ is a map from memory addresses (and registers) to their encoded values. And the $\text{Condition}(c) : mS \rightarrow \{\perp, \top\}$ is a function over the current program state, encoding the constraint which was checked by statement $c$. Updating the environment to invert the value of $\text{Condition}(c)$ can change the subsequent instructions. Each execution trace, modeling a single execution path, can be divided into multiple trace segments ($TS_k$) 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 $\text{CONFINED-OP}$ rule updates the concrete state ($S$) by passing it through the behavior function of $c$ statement ($f(S)$). As it does not finish the execution of a trace segment, the segment index ($j$) remains unchanged. Similarly, the sequence of trace segments ($\langle TS_k \rangle_{k=1}^m$) is not affected. But the symbolic encoding of memory changes and satisfied constraints in the current segment ($TS_j$) should be updated based on the behavior ($f$) and condition ($c$) functions of $c$ statement. The $mC_j : mS \rightarrow mS$ which maps the old program state (right before execution of $TS_j$) to its new state (after execution of $TS_j$), can be updated by combining it with the behavior function ($f \circ mC_j$) because the program state should undergo changes of all executed instructions sequentially ($f(mC_j(.)$)). The $C_i : mS \rightarrow \{\perp, \top\}$ function inspects the old program state (right before execution of $TS_j$) and reports whether all assertions of $TS_j$ are satisfied ($C_i(.) = \top$) or some alternative sequence of statements and their corresponding segment should be executed ($C_i(.) = \perp$). Because the condition function ($c$) of $c$ statement operates on the current program state ($S$), we have to update it to take care of program state differences ($c' = c \circ mC_j$), so it can process the old program state and then update the segment assertions using a boolean conjunction ($C_j \land c'$). The $mI : \mathbb{N} \rightarrow \overline{mS}$ which indicates how new symbols should be initialized at the beginning of each trace segment is not affected too. Finally, the next state which is observed during the concolic execution is denoted by $s = \text{Next}(f(S), c)$.

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

The $\text{ILLUMINATED-OP}$ rule is invoked after execution of an illuminated statement ($i$) and when the program state has been updated out of the instrumentation control. This program state ($S$) needs to be updated as dictated in its corresponding memory initialization function ($mI[j + 1] : \overline{mS}$ where $\overline{mS} := \mathbb{N} \rightarrow \mathbb{N} \cup \{\perp\}$). As shown in Eq. (3), the new state is formed by overriding those memory cells which have been assigned to a new value by PSS decision. The trace segment index is incremented ($j + 1$) and the new trace segment is appended to the sequence of trace segments ($TS_i = (C_j, mC_j, SC_j)$). The $C_j$ and $mC_j$ are the segment’s symbolic conditions and behavior encodings which were accumulated during the execution of $TS_j$ trace segment and $SC_j = \text{Terminator}(i)$ denotes the terminating operation which finished execution of that segment (e.g. invoking the “open” system call). The condition and behavior tracking functions needs to be updated for the new trace segment too. For this purpose, a tautology function ($\top$) is assigned as the new condition ($C_{j+1}$) and an identity function ($id_{mS}(x) = x$ for $x \in mS$) is used for initializing the behavior function.
ETG is a graph consisted of all program traces. It is not necessarily a tree, because parts of the ETG can reference (via branches) to its other parts. However, it is not correlated with the obfuscated program CFG, because all branches which are resolved during a trace are simplified together. The ETG just branches when there is an input-dependent condition. For example, deobfuscating a VO protected program eliminates all checks on values of the pseudo-opcodes, because inspecting those opcodes always lead to the execution of 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 discussed later in Section 3.3.

The PSS component is responsible for driving the deobfuscation process to extract information out of the input binary iteratively. Pointing to the algorithm sketch in Figure 4, the PSS starts with an initial execution of the `P` obfuscated program. This initial trace, establishes a single-path ETG to approximate the input program. By 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, knowing that a read register contains a symbol which was checked to be greater than \( \alpha \) to follow an execution path, PSS can assign a number less than \( \alpha \) to that symbol in order to guide the execution of the program along a new path in the following iteration. This issue is correlated with the deadcode detection (i.e. parts which cannot be executed with any input) 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) solving. If an SMT query is proved to be unsatisfiable, the related code is deadcode. If a concrete solution can be found, the CEE can be guided through that path.

The SMTS component, at top of the Figure 2, is designed to answer 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 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 those worst case scenarios, is to negate the conditions of the non-conforming branches in memory, so it can follow the desired side of each branch independent of used concrete values. However, this 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, by each iteration the constructed graph approximates the obfuscated binary in more details. An example strategy is to depth-first search (DFS) until an instruction is visited \( k \) times (so \( k \) rounds of an input-dependent loop are unfolded). Search strategies and maximizing code coverage are research subjects in the field of software automated testing.

### 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 virtualization obfuscation, 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 themselves (like [8]), CEE captures the behavior of each executed instruction in terms of its mathematical formula. Although instructions can be obfuscated, their functionality and so the formula modeling how registers and memory addresses are changed by the program execution, remain obtainable across all obfuscations. Second, dynamic coverage puts a trade-off between completeness of analysis and the used time and memory resources. To resolve this trade-off, concolic [12] [13] execution is employed in which all instructions are executed normally (so anti-disassembly techniques are mitigated automatically) and also each instruction is inspected symbolically (so the obtained input/output relationship is not restricted to the environment/concrete inputs).

Another challenge is how to drive the program, during the dynamic analysis (under the control of the CEE), along the execution path which is selected 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 being read by the program.

Definition 3 (Symbol). A symbol \( \text{SYM} = (v, t) \) models a value which can be stored at some memory address or register. The \( v \) is the concrete value represented by \( \text{SYM} \) in the current execution. The \( t \) is the type of the \( v \) (e.g. \text{uint16}, t) indicating the set of values which could be used by \( \text{SYM} \).

Definition 4 (More Constrained Initialization Relation). A memory initialization function \( mI' \) is said to be “more constrained” than another memory initialization function \( mI \) and is denoted as \( mI \in mI' \) if and only if \( mI' \) is just like \( mI \) but initializes more memory symbols. Formally, depicted in Eq. (4).

\[
\forall mI, mI' : N \rightarrow mS \cdot mI \in mI' \iff (mI \neq mI' \land \forall j : N \cdot \forall x \in N \cdot mI[j](x) \neq \bot \Rightarrow mI'[j](x) = mI'[j](x))
\]

(4)

An obfuscator can challenge dynamic analysis solutions in two main ways. On one hand, it can add guard codes to hide the original program at runtime. A notable guard code is the latency-checker where 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 RDTS instruction loads the current time-stamp into registers. Although this value is not a user-input, it is not computed by the program too. 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 to maximize the analysis code coverage.

On the other hand, the computed function itself can be complicated (e.g. by additional neutral arithmetic) so 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 bitwise-and of a symbol with \(|S| \) bits and some bitmask which is zero in its lowest \(|S| \) bits removes the \( S \) symbol.

Theorem 1 (Repeatability). Given a program \( \mathcal{P} \) and one of its traces \( T_n = \langle TS_k = (C_k, mC_k, SC_k) \rangle_{k=1}^{n} \) which had been acquired through an execution \( E[\mathcal{P}, mI] \) for some memory initialization function \( mI \), either repeating the execution \( E[\mathcal{P}, mI] \) always produces the same trace or there exists a more constrained memory initialization function \( mI^* : N \rightarrow mS \) which always produces the same execution trace. Formally, depicted in Eq. (5).

\[
\forall \mathcal{P}, mI \cdot \forall T_n : E[\mathcal{P}, mI] \cdot \exists mI^* : N \rightarrow mS \cdot (mI = mI^* \lor mI \in mI^*) \land |E[\mathcal{P}, mI^*]| = 1 \land T_n \in E[\mathcal{P}, mI^*])
\]

(5)
Proof. Repeating the execution $E[P, mI]$ (using $mI = mI^*$), a trace such as $T_m^n = \langle TS_k^* = (C_k, mC_k^*, SC_k^*) \rangle_{k=1}^m$ is produced. If we can prove equality of this arbitrary trace with the target trace of the theorem ($T_m^n = T_n$), it shows that no other trace could be produced during that guided execution and so $|E[P, mI^*]| = 1$ which completes the proof. For this purpose, it is enough to show that $n = |T_n| = |T_m^n| = m$ and $(C_k, mC_k, SC_k) = (C_k^*, mC_k^*, SC_k^*)$ for $1 \leq k \leq n$. Also in scenarios that this equality cannot be proved, the $E[P, mI^*]$ can be changed by adding more constraints to $mI^*$ in order to maintain the equality of traces as while as the $mI \in mI^*$ equation is not violated. In these scenarios, the second form of the target proposition (Eq. (6)) is proved.

\[
\left( (mI \in mI^*) \land |E[P, mI^*]| = 1 \land T_n \in E[P, mI^*] \right)
\]

(6)

In both traces, execution of $P$ is initiated by the same statement and so the initial states of $TS_1$ and $TS_1^*$ trace segments are the same. If we can prove that this equality is maintained during the execution of first trace segment and the same sequence of statements are executed in both runs, the final system call of both segments will be the same (since it is the last statement which is obtained during the execution of first segment) and it observes the same concrete state of $P$ (because the equality had been preserved during the execution of that trace segment). This ensures that either both traces will terminate or both of them will continue execution from the second trace segment. However, the possible differences in the kernel state can differentiate the initial concrete states of two following trace segments. 

By induction, we prove that if the first $q$ trace segments are equal ($(TS_k^*)_{k=1}^q = (TS_k^*)_{k=1}^q$), the $q+1$-th trace segments ($TS_{q+1}^*$ and $TS_{q+1}$) which have to start with the same statements but different concrete states can be made equal only by adding more constraints to $mI^*$ while maintaining the $mI \in mI^*$ equation. 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 an observation that the first statement of $TS_{q+1}^*$ and $TS_{q+1}$ is an illuminated statement $i$ which is seen after return of the $SC_q = SC_q^*$ while the concrete state of two runs (see Figure 3) can differ ($S_{q+1}^* \neq S_{q+1}$) before executing the $i$ itself. But after execution of $i$, all following statements should be confined states because the first following illuminated statement will terminate the trace segment. Execution of the first illuminated statement uses the operation rule of Eq. (7).

\[
S_{q+1}^* = S_{q+1} \oplus mI[q + 1], \quad SC_q = \text{Terminator}(i), \quad TS_q = (C_q, mC_q, SC_q), \quad s = \text{Next}(S_{q+1}^*), \quad T_{q+1} = \langle TS_{k=1}^* \rangle, \quad \text{Illuminated-OP (7)}
\]

The set of possible difference points between two functions $u, v : \mathbb{N} \rightarrow \mathbb{N}$ can be obtained according to Eq. (8). The possible difference points of initial program states ($S_{q+1}^* \oplus S_{q+1}^*$) can either be overridden by $mI[q + 1]$ during execution of $i$ (in which case it is not required to update $mI^*$) or some points will remain different among two sets ($S_{q+1}^* \oplus S_{q+1}^*$) $\{x \in \mathbb{N} | mI[q + 1](x) = \perp \}$. In this case, the concrete states can be made equal by updating $mI^*[q + 1]$ according to Eq. (9). This ensures that $S_{q+1}^* = S_{q+1}^*$. 

\[
\begin{align*}
u \in v \in u \in &= \{x : \mathbb{N} | u(x) \neq v(x)\} \quad \text{(8)} \\
mI^*[q + 1](x) &= \begin{cases} mI[q + 1](x), & mI[q + 1](x) \neq \perp \\ S_{q+1}(x), & mI[q + 1](x) = \perp \land S_{q+1}(x) \neq S_{q+1}^* \\ \perp, & \text{otherwise} \end{cases} \quad \text{(9)} \\
mI^*[q + 1] &= S_{q+1} \oplus mI[q + 1] \quad \text{(10)}
\end{align*}
\]

Because this reasoning is made about an arbitrary member of the new guided execution set ($T_m^n \in E[P, mI^*]$), it shows that from any initial difference set such as ($S_{q+1} \oplus S_{q+1}^*$) an appropriate patch can be constructed for updating $mI^*[q + 1]$ and equating $TS_{q+1}$ and $TS_{q+1}^*$ trace segments. However, for constructing $mI^*[q + 1]$ 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). Another approach is to repeat the execution iteratively. Running once using $E[P, mI^{(1)}]$ where $mI^{(1)} = mI$ in order to obtain
difference sets and produce \( m \mathcal{L}^{(2)} \) to override those differences, then running \( \mathcal{E} [\mathcal{P}, m \mathcal{L}^{(2)}] \) to obtain \( m \mathcal{L}^{(3)} \) and so on. Since more cells are initialized in each turn, the initialization vector becomes more constrained \((m \mathcal{L}^{(1)} \subseteq m \mathcal{L}^{(2)} \subseteq \cdots \subseteq m \mathcal{L}^{*})\) and the guided execution set will converge to a single-member set.

Since concrete program states are made equal, the next executing statement in both trace segments will be the same too (\( s = \text{Next}(S_{q+1}^\prime, i) = \text{Next}(S_{q+1}^*, i) \)). Also initial constraints and behavior functions are initialized to the same values (\( \top, id_{mS} \)). Execution of the next confined statement \( c \) uses the operation rule of Eq. (11).

\[
\begin{align*}
  f = \text{Behavior}(c) & \quad c = \text{Condition}(c) & c(S_{q+1}) = \top & \quad c' = c \circ mC_{q+1} & \quad s = \text{Next}(f(S_{q+1}), c) \\
  S_{q+1}, q + 1, (TS_k)^T_{k=1}, C_{q+1}, mC_{q+1}, mL, c \rightarrow f(S_{q+1}), q + 1, (TS_k)^T_{k=1}, C_{q+1} + c', f \circ mC_{q+1}, mL, s & \quad \text{CONFINED-OP} \quad (11)
\end{align*}
\]

Because both trace segments are running the same \( c \) statement, its condition/behavior functions will be the same too \((f = f^* \text{ and } c = c^*)\). And since \( S_{q+1} = S_{q+1}^* \) before execution of \( c \), the following concrete states will be the same too \((f(S_{q+1}) = f^*(S_{q+1}^*))\). Similarly, the updated condition (Eq. (12)) and behavior (Eq. (13)) functions will be equal. Finally, equality of following concrete states \((f(S_{q+1}) = f^*(S_{q+1}^*))\) leads to execution of the same following statement \((s = \text{Next}(f(S_{q+1})), c) = \text{Next}(f^*(S_{q+1}^*)), c)\).

\[
\begin{align*}
  C_{q+1} \land c' = C_{q+1} \land (c \circ mC_{q+1}) = C_{q+1}^* \land (c^* \circ mC_{q+1}^*) = C_{q+1}^* \land c^* & \quad (12) \\
  f \circ mC_{q+1} = f^* \circ mC_{q+1}^* & \quad (13)
\end{align*}
\]

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

The theorem (1) states that all programs are deterministic by themselves. 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 program (e.g. syscalls or hardware timers) are instrumented, those inputs can be modified by CEE before their first use. For example opening a file may succeed/fail at different times, but even if a file is deleted the program (e.g. syscalls or hardware timers) are instrumented, those inputs can be modified by CEE before their first use. For example opening a file may succeed/fail at different times, but even if a file is deleted the program (e.g. syscalls or hardware timers) are instrumented, those inputs can be modified by CEE before their first use.

### 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 is 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. So 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 Figure 5 algorithm starts with the ETG root node and visits its nodes as follows. Visiting a node such as \( n \) which is connected to \( n \) other nodes, an if-else construct is outputted having an if part for each one of \( n \) 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.
As the program is analyzed more in each iteration, new parts of it are discovered and its ETG is evolved too. So 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 $P$ and $P^*$ are equivalent, shown as $P \equiv P^*$, if and only if they behave equivalently for all possible memory initializations (inputs). Formally, depicted in Eq. (14).

$$\forall P, P^* . P \equiv P^* \iff \forall mI : N \to m\mathcal{S} . \mathcal{E}[P,mI] = \mathcal{E}[P^*,mI]$$

**Theorem 2 (Equivalence).** If $O_P$ and $O_{P^*}$ are two obfuscated instances of program $P$, with this assumption that used obfuscation transformation has neither eliminated nor added to the observable behavior of $P$, then both of $\psi(O_P)$ and $\psi(O_{P^*})$ are equivalent with the $P$. Formally, depicted in Eq. (15).

$$\forall P, P^* , O(\). \left[(O_P \equiv P) \land (O_{P^*} \equiv P)\right] \implies \psi(O_P) \equiv \psi(O_{P^*}) \equiv P$$

**Proof.** In order to prove the target proposition ($\psi(O_P) \equiv \psi(O_{P^*}) \equiv P$) based on the given assumption ($O_P \equiv O_{P^*} \equiv P$), 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.

$$\left[O_P \equiv O_{P^*} \equiv P\right] \land \left[\left(\psi(O_P) \neq P\right) \lor \left(\psi(O_{P^*}) \neq P\right)\right]$$

Without loss of generality, assume that $\psi(O_P) \neq P$ holds (the same reasoning can be used about the $\psi(O_{P^*}) \neq P$ case).

$$\psi(O_P) \neq P$$

$$\implies \exists mI : N \to m\mathcal{S} . \mathcal{E}[\psi(O_P),mI] \neq \mathcal{E}[P,mI] \quad \triangleright \text{by Def. 6}$$

$$\implies \exists \mathcal{T}_n^{\psi(O_P)[x]} \in \mathcal{E}[\psi(O_P),mI] . \forall \mathcal{T}_m^{P[x]} \in \mathcal{E}[P,mI] . \mathcal{T}_n^{\psi(O_P)[x]} \neq \mathcal{T}_m^{P[x]} \quad \triangleright \text{by Def. 2}$$

$$\implies \exists \mathcal{T}_n^{O_{P^*}[x]} \in \mathcal{E}[O_{P^*},mI] . \forall \mathcal{T}_m^{P[x]} \in \mathcal{E}[P,mI] . \mathcal{T}_n^{O_{P^*}[x]} \neq \mathcal{T}_m^{P[x]} \quad \triangleright \text{by Def. 2}$$

$$\implies \exists mI : N \to m\mathcal{S} . \mathcal{E}[O_P,mI] \neq \mathcal{E}[P,mI] \quad \triangleright \text{by Def. 2}$$

$$\implies O_P \neq P \quad \triangleright \text{by Def. 6}$$

Alternatively, in Eq. (15), it is possible to say $\exists \mathcal{T}_n^{P[x]} \in \mathcal{E}[P,mI] . \forall \mathcal{T}_m^{P[x]} \in \mathcal{E}[P,mI] . \mathcal{T}_n^{P[x]} \neq \mathcal{T}_m^{P[x]}$ but due to their symmetry, we used the first case without loss of generality. Thus, the opposite assumption of Eq. (17) leads to $O_P \neq P$ in Eq. (22) which is in contradiction with the assumption of $O_P \equiv P$.

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 nor completeness. In what follows, Section 4 explains the implementation details and related practical concerns.

**4. Implementation**

This section discusses how the implementation was carried out and enumerates some of the major practical challenges which had to be resolved in the process. It should be noted 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 [http://ce.sharif.edu/~b_momeni/projects/twinner](http://ce.sharif.edu/~b_momeni/projects/twinner) project page.
The 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 is working 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 which were seen in former traces are placed on left of those which are seen afterwards.

For example, consider the program shown in Figure 7a. 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 7c (false branches of lines 21 and 23 conditions). When the DFS finishes traversing the subgraph starting at L37, left child L of the constraint C of L23 node, it continues by asserting all constraints from the root node of ETG till C and negation of constraint L. Next execution of the program will produce a list of constraints starting with L21 and L23 constraints. Therefore, new nodes which are added to the ETG are placed in the subgraph starting at right child of 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 ldmlb [25] architecture. The ldmlb 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 to extract a single trace from the given binary. It runs the input program concolically; instrumenting all assembly instructions and tracking memory and register changes symbolically. This gray-box approach has the advantage that intermediate conditions which were inspected by the binary will become visible to twintool and can be recorded in the trace, without any dependence on extra information such as the usage of a specific compiler or access to the source code which is normally unavailable during the malware analysis. The int64 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 [26, p. 1459] such as SUB AX, imm16 and SUB r/m8, r8. Deobfuscating a program needs all of its assembly instructions used forms to be supported in the twintool. To make it scalable, twintool implements a generic instrumentation layer (GIL) as designed in the ldmlb [25] architecture. It automatically detects operand types (e.g. memory address), their read/write sizes, organizes them within 40 generic categories (e.g. DstRegSrcImplicit), and instruments them using a set of generic analysis routines (GAR). 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 on 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 the 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 program’s concrete state and dictate which execution path will be followed within that running instance of the program. Consequently, program can be forced to follow another path by changing symbols’ initial values. Finally, line 11 updates the EFLAGS register with an AdditionOperationGroup object which can be queried in the following conditional instructions such as conditional branches.
The last component of the framework (i.e. the SMTS) is realized using the CVC4 \cite{27} library for solving symbolic constraints. It accepts a set of symbolic constraints and produces one of 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, constraints which twintool has extracted out of the binary may depend on the program real inputs and/or artificially added parts such as the VM interpreter code in a virtualization obfuscation scenario or the decryptor/decompresser code for a packed malware. Next, these symbolic constraints may become more and more complicated so keeping them in the memory becomes a challenge by itself. 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. First input category is user inputs which are given as initial values to the program and 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, can be read by the program, and consequently change the execution path either (a) explicitly; via conditional branches (e.g. \texttt{je .L2}) or (b) implicitly; via calculated values (e.g. \texttt{movq -16(%rbp), %rax; jmp %rax}). 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 “\texttt{read(fd, buf, len)}”) and completes its logic according to the given arguments and other parts of its state (the caller process in general such as the open “\texttt{fd}” file descriptor) and also 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 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 section of the malware are the same in all executions. Although the execution path depends on both of the virtual opcodes and 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 \ref{fig:7b}. The loop created by \texttt{L33} node is totally removed in Figure \ref{fig:7c} because it depends on \texttt{-8(%rbp)} which is initialized in line 26 of Figure \ref{fig:7a} as a constant. It wouldn’t help the obfuscator to initialize it as a function of user inputs and then cancel their effects in the comparison of line 32 because the constraints simplification could simplify and remove that 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 of Figure \ref{fig:7a} in which the value of \%eax register, e.g. \texttt{s} symbolic input, is added to itself repeatedly to produce \texttt{s+s}, \texttt{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 till the end of execution to be processed in SMTS. So the expression is kept as \texttt{2s}, \texttt{4s}, etc. with constant memory footprint.

The twincode generated from the described assembly code is shown in Figure \ref{fig:7d}. 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 \texttt{L37} part of the ETG as shown in Figure \ref{fig:7c} line 16 prints the input multiplied by 1024 (i.e. \texttt{0x400}), while lines 22 and 28 print messages about overflow/underflow scenarios of the original program. Some of initialization parts of the Figure \ref{fig:7d} 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 virtualization obfuscation technique to observe the practical quality of the deobfuscation results. The VO replaces the entire text section of the program with an interpreter and also 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 [28] test programs are 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 execution trace graphs (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 was learned by analysis of the given binary and also corresponds with the CFG of its twincode counterpart while CFG of the original program can be altered and replaced completely by obfuscation transformations. Virtualization obfuscation is one of the most complicated methods for obfuscating an arbitrary program. In this technique a random instance of a template language is selected, program is compiled to it and placed in the data section, and the entire text section is replaced by a virtual machine interpreter 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 four programs with a variety of control flow graphs in the first step to obtain similarly incomprehensible and indistinguishable binaries which just differ in their pseudorandom data sections. Afterwards, VO-protected programs are reversed and deobfuscated to obtain their execution trace graphs. Finally, similarity of ETG of deobfuscated programs and ETG of their corresponding original programs are measured according to Def. 7 in order to quantify the effectiveness of 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 shown as \( \delta_{g_1, g_2} \) and is defined as follows:

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

The test input programs themselves have no specific importance. It is just 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 four selected input programs. First program depicted in Figure 9a has nine parallel execution paths. The first path runs when the input arguments are not well-formed. The eight other paths are selected based on three code characters in the program argument and print an appropriate message in each path. Next program depicted in Figure 9b has two main execution paths which one of them consists of two 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 9c has a main execution path with three exceptional branches. Finally, Figure 9d 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 of the deobfuscation process. And another 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 of the deobfuscation effect has been removed successfully. To VO-protect each one of test programs, they are compiled for a virtual language with five primitive opcodes. The interpreter of this language is shown in Figure 10a and its virtual opcodes are described in Table 2. Program itself is encoded in the program text and is 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 corresponding conditional/unconditional jumps, printing operation, and so on.

The CFG of the resulting VO-protected code is shown in Figure 10b which is clearly independent of the initial programs. The top node in Figure 10b corresponds to the beginning of interpreter which calls the switch (line 19 of Figure 10a). The second node of CFG can jump to five destinations which correspond to five virtual opcodes. The first destination from right belongs to end_of_execution opcode and so halts the interpreter. Second destination from right of Figure 10b corresponds to if_then_else virtual opcode which reads five arguments (two operands to be compared and a comparison code to determine the comparison operator in addition to two offsets for then/else parts of the conditional jump). Two following 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 which are found at the given addresses, and stores the comparison result in the aux variable for next checks. This aux variable can be accessed as an encoded argument by the following virtual operations. These four nodes (corresponding to the last four cases of the switch statement) merge in the bottom most node of Figure 10b and jump back to the loop header node afterwards to continue with interpretation of the next virtual operation.

All programs are mapped to 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 the same syntactically 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 given input binaries while their corresponding deobfuscated versions are depicted in Figure 12. Comparing these four figures 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 12b and its corresponding pre-obfuscation ETG which is shown in Figure 11b. Both graphs consist of three main parts. First part which branches from the main execution path at the right side of figure, checks for the correct number of arguments. This part correspond to the left most path in Figure 9b. Second part branches into three scenarios, prints the “first-else-part” message in two scenarios and the “first-then-part” in the last scenario. All three paths are correctly merged before reaching to the last part of the program. These three branching scenarios correspond to the two expressions which their logical conjunction had been checked as shown in the top-right side of Figure 9b. Third part prints two then/else-part messages, merges similar to the bottom-right side of Figure 9b, and then joins with the first part (for checking arguments). Comparing the Figure 12b output ETG with the Figure 11b input ETG shows the complete recovery of all scenarios from the VO-protected binary.

Also twincodes of these four programs are produced which can be used for further analysis. Figure 13a shows the twincode which is obtained by analysis of the VO protected version of P2. First condition (line 7) has renamed the argc to rdi_0 and checks for the correct number of arguments. Line 15 checks for presence of valid option and line 23 checks for valid option value. The CFG of this code is depicted in Figure 13b which corresponds to its ETG which was shown in Figure 12a. Looking at CFG of Figure 13b from top to bottom, first node corresponds to the condition of line 7 which compares rdi_0 with number three. If there are less than three arguments, the right most path of CFG will be followed which leads to execution of line 36 and printing the program usage message. Otherwise, the second node which corresponds to lines 15-20 operations 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 comment as a hint). If the wrong option name was used, the left most path of CFG will be followed and so line 21 prints the corresponding error message. The last condition which is located at the center of Figure 13b corresponds to lines 23-26 check which compares argv[2] with the “optvalue” string and selects one of line 28 (error case; right branch in CFG) or line 30 (target case; left branch in CFG) to be executed. Except for arrangement of nodes, the CFG of Figure 13b has a main execution path (in which all conditions are evaluated to true) from which three exceptional paths have deviated similar to the original CFG of P2 which was depicted in Figure 9b.

These input/output ETGs can be compared pairwise according to Def. 7 in order to assign a similarity measure to each pair. Table 3 aggregates these calculated similarities. The diagonal entries of Table 3 are all relatively higher than non-diagonal entries. The close to one value for diagonal entries indicates 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 0.7 similarity value. For larger programs, the similarity drops to below 0.5 while all diagonal entries stay higher than 0.88 similarity value.

As this proof of concept implementation can be extended to analyze any other application just by supporting their possibly different assembly instructions to recognize and formulate their calculated symbolic expressions, it can be deduced that virtualization obfuscation can be automatically reversed on other protected programs similar to the mentioned example. It is worthy to note that this framework did not assume anything about the structure of the used virtual machine to produce the twincode. For example, the VM can eliminate the ptr (i.e. virtual program counter) and connect different pieces of the virtualized program directly together (e.g. similar to what is done in the jump oriented programming [29]) without causing any change in its corresponding twincode.

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

5.2. Performance

In order to measure the execution time overhead of the twintool analysis runs, a set of complicated programs are 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 is repeated as many times as required according to the central limit theorem in order to limit the maximum error of the reported mean execution time to at most 0.5% with confidence level of 95%.

Obtained results are aggregated in Table 4 where second column reports the native execution time, third and fourth columns indicate the baseline overhead which is caused by using the Pin dynamic binary instrumentation framework, and two last 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, what follows discusses the possible deobfuscation challenges, how they are mitigated in the Twinner framework, and presents an objective comparison with previous works.

One of the hardest scenarios for SMTS is an opaque predicate (i.e. an always true/false condition) so complicated that SMTS cannot reason about its negated constraint within a short time span. For always true constraints, the code’s logic is captured in the first trace containing it and 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 too. Thus, this path is assumed to be deadcode (which is the case as well). But when SMTS cannot find an answer for 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 also increases the number of paths to be analyzed. One strategy which can be inspected in future works is to prioritize the execution paths, giving lower priority to those suspected paths, and allocating available resources for their analysis according to their priorities. This may lead to a more balanced answer to the trade-off between full analysis of all possible branches and minimizing the analysis time for reachable code paths.

Nevertheless, the current approach of focusing on the constraints which are solvable within the given time limits which is described in the following example is enough for 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 with 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 just by communicating with an arbitrary network server and not necessarily the real C&C server. This is in contrast to methods such as who analyze a real traffic trace in order to model it and simulate the C&C server for the bot.

The bot program needs cooperation from the OS for receiving its 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’s 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. As while all executed instructions are instrumented, acquired symbolic expressions for all addresses have to match with their concrete values. But syscall, executed in the kernel-space, is out of the instrumentation scope. So concrete/symbolic values can mismatch after a syscall. When this happens (e.g. contents of a file is 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. Bot checks the code and terminates because it expects a 4 bytes long PDU. This constraint is captured by twintool symbolically and is solved to obtain a four bytes long input to be used in the next analysis round. Then, the network server sends 0xB code again, but the buffer is modified by twintool on the fly to be seen as the calculated four bytes value. 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 about the high number of assembly instructions and how they can be instrumented correctly and efficiently. For this purpose, the ldmbl architecture has been used which abstracts provided APIs of Pin 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 it was 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, so they 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, but such a vulnerability (if exists) must be addressed in the underlying DBI framework and is out of the adversarial model of twintool.

Table 5 summarizes a comparison of mentioned related works and the Twinner framework. Each cell is marked with ✓ if the solution which is mentioned in that row is able to overcome the obfuscation technique which is noted in its column. If it cannot reverse that transformation, the cell is marked with × and if it cannot be reversed completely but is partially considered in the solution, the ∼ mark is used. The - mark indicates that mentioned feature is inapplicable to that solution.

All solutions support packings except ROPMEMU which starts its work with a memory dump assuming that ROP chains exist in that dump. Three solutions considered arithmetical and logical obfuscations by means of compiler transformations and/or symbolic expression equivalence tests. 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 counterexample is X-Force which will exhibit exponential execution time by forcing analysis of deadcode portions.

Except Twinner and the Rolles method which finds a correspondence between used binary portions and previously reverse engineered VM parts, other solutions suffice to learn some model such as CFG and do not provide a source code representation which can be used as input of other analysis tools. Among solutions which support the VO-protection scheme, Rolles requires priori knowledge of the used VM. This knowledge needs to be updated per VM language instance. Rotalume and Kinder need a specific VM metamodel in order to detect it as a binary pattern and extract VM and protected codes based on it. And Yadegari and Coogan focus on some given execution traces and do not have a systematic approach for maximizing the code analysis coverage.

Next feature is about analysis of program loader itself. Malware can hide with techniques such as ELF weird machines out of the common executable area and run before beginning of the actual program. X-Force and Twinner support program analysis from the entry point and so can detect such tricks. Also Yadegari and Coogan employ Ether low-level instruction traces and so can 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 about analysis of multi-threaded programs. Except X-Force [16] which serializes all threads by replacing thread creation API with a direct call to the thread entry function, other solutions follow the OS scheduler decisions about order of threads.

Finally, the last two columns of Table 5 indicate resistance of solutions against anti-assembly and anti-debugging techniques. Kinder [4] as a static analysis method suffers more from the anti-disassembly stopping 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, Twinner, which can dynamically analyze an arbitrary Windows/Linux executable program. Presented framework maps the deobfuscation problem into three components. First, a concolic execution engine (CEE) which instruments the given binary and captures its runtime behavior in a series of symbolic expressions and constraints. Second, a path search strategy (PSS) which learns a behavioral model of program by running it through different execution paths iteratively. In each run, an independent instance of CEE is employed to run the binary along an specific path and produce its corresponding trace object. Third, a library for solving symbolic constraints (SMTS) which is 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 is presented and evaluated to measure the deobfuscation effectiveness and performance using different real-world programs. The proposed method is not dependent on any obfuscation process or structure of the obfuscated program. CEE which is realized as twintool library is implemented using the ldumbi architecture based on the Intel Pin. By instrumenting assembly instructions, it finds out about memory symbols before their first use and so can guide the program along a specific execution path by modifying their concrete values. PSS which is realized as twinner provides a command line interface for configuring the deobfuscation parameters and combines traces which are received from twintool to update an execution trace graph (ETG). In each round, twinner selects the next execution path by DFS searching the ETG and obtains a list of constraints for satisfying all branches along the selected path. The CVC4 SMT solver is used to find a concrete solution for those constraints and feed the twintool.

As code is analyzed concolically along all paths, anti-debugging techniques aren’t an obstacle and as the behavior is tracked symbolically, the complete functionality of the program is captured in the generated twincode. Used concepts such as trace and guided executions are formally defined and used to prove properties about deobfuscation process. To evaluate its effectiveness, several programs are protected with virtualization obfuscation and then used as test inputs. ETG graphs of analyzed programs are drawn and compared with each other to see how much of details of original programs are restored after deobfuscation. A graph similarity measure is defined for this purpose which shows that obtained ETGs after deobfuscation are considerably similar to the original CFGs and can distinguish test inputs with at least 18% margin. ETGs match CFGs of original programs except for rearrangement of graph nodes and are encoded as twincode in C language which can be used for further analysis. To evaluate the performance of deobfuscation process, the SPEC test programs are used. Analysis of complex programs such as gcc demonstrated that the additional overhead of twintool is in the order of 10 times which is considerably lower than the additional overhead of Pin which is 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 such as DNG algorithm of a bot which is impossible without its deobfuscation, generation of comprehensive behavioral signatures for categorizing malware in their related families, debugging benign software which are obfuscated so correctness of used obfuscation transformation can be examined, and analysis of close sourced and obfuscated software for presence of possible backdoor.
List of Figures

1. An example code with several obfuscations such as a while loop which does not stop before an overflow, an unreachable code which calls a complicated function, and some arithmetical obfuscations. ................................................................. 22
2. The deobfuscation framework in which components are shown as rectangles and products are drawn as ellipses. ................................................................. 22
3. Operational semantics for execution of program in CEE. ....................... 22
4. Sketch of the PSS algorithm using SMTS for solving symbolic constraints and CEE for executing the $P$ through the intended paths. ....................... 23
5. Sketch of the twincode encoding algorithm ........................................ 23
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. ....................... 24
8. Implementation of the ADD assembly instruction analysis routine. ............... 24
9. Control flow graphs of test input programs set ..................................... 25
10. The interpreter used for virtualization obfuscation protection of test programs. ....................... 26
11. Execution trace graphs of input binaries before obfuscating them ............... 27
12. Execution trace graphs of output binaries which were VO-protected ............... 28
13. Sample twincode and its corresponding CFG. ....................................... 29

List of Tables

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 gives the bit length; $\text{len}(.)$ function gives number of used tokens. .................................................. 30
2. Similarity degree of pre/post-obfuscation ETG figures ................................ 31
3. Execution times and relative overheads for SPEC programs analysis. The Native column presents the execution times in milliseconds without any instrumentation. The Count column indicates the execution times in seconds when an instruction counting pintool is used. The IAO columns reports the Instruction-counting Added Overhead as a factor of native execution times. The Twintool column presents the measured times when the Twintool pintool is used to instrument each SPEC test program. The TAO column reports the Twintool Added Overhead as a factor of instruction counting pintool execution times. ....................... 31
5. Comparison of Analysis and Deobfuscation Solutions. The ✓ symbol indicates that mentioned solution can overcome the obfuscation technique, the × shows an unsupported feature, the - indicates inapplicable features, and ~ is for obfuscation features which cannot be reversed completely but are partially considered. ....................... 32
```c
int getsecret (unsigned int key, int salt) {
    if (key > 5)
        while (key > 0)
            key++;  
    else if (salt > 5)
        if (key > 6)
            return complicatedFunction (key);
        else
            return (key>>24)
| (key>>16) | (key>>8) | key;
    return (key ˆ salt) % 1024;
}
```

Figure 1: An example code with several obfuscations such as a while loop which does not stop before an overflow, an unreachable code which calls a complicated function, and some arithmetical obfuscations.

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

Figure 3: Operational semantics for execution of program in CEE.
function PSS($P$)  
  $T \leftarrow$ CEE($P$)  
  ETG $\leftarrow$ \{T\}  
  while ETG contains an uncovered path do  
    $\zeta_i \leftarrow$ next uncovered execution path to analyze  
    $C_i \leftarrow$ SMTS($\zeta_i$)  
    if $C_i \neq$ Unsatisfiable then  
      $T_i \leftarrow$ CEE($P$, $C_i$)  
      ETG $\leftarrow$ ETG $\cup$ \{$T_i$\}  
    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 the $P$ through the intended paths.

function EncodeAsTwincode($g : ETG$)  
  u $\leftarrow$ 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 $c \in$ Children($u$) do  
    $C \leftarrow$ Condition($c$)  
    if IsSegmentTerminator($c$) then  
      $TS \leftarrow$ Segment($c$)  
      Output MemChanges($TS$)  
      Output RegChanges($TS$)  
      $SC \leftarrow$ Syscall($TS$)  
      Output "InvokeSyscall($SC$)"  
      i $\leftarrow$ 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

Figure 6: Proposed framework reference implementation component diagram.
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.

Figure 8: Implementation of the ADD assembly instruction analysis routine.
Figure 9: Control flow graphs of test input programs set
int aux; // for intermediate values such as return value

int end_of_execution(const char *&ptr);
void printf_command(const char *&ptr);
void strcmp_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[]);

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 interpreter code
(b) Corresponding control flow graph

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

26
Figure 11: Execution trace graphs of input binaries before obfuscating them
Figure 12: Execution trace graphs of output binaries which were VO-protected
int main (int argc, char *argv[]) {
    struct RegistersSet regs;
    const UINT64 rax_0 = regs.rax, rbx_0 = regs.rbx, ...;
    const UINT64 rdx_0 = regs.rdx, rsi_0 = regs.rsi, ...;
    if ((signExtend_0x80_0x20 ((UINT64 (rdi_0) & 0xffffffff)) >= UINT128 (0x0, 0x0, 0x0, 0x3))) {
        regs.rcx = UINT64 (0x14);
        regs.rax = strcmp (/*"--wrongopt"*/ (const char *) argv[1], /*"--option"*/ (const char *) 0x400e80);
        const UINT64 rax_1 = regs.rax, rbx_1 = regs.rbx, ...;
        const UINT64 xmm0_1 = UINT128 (regs.xmm0), ...;
        const UINT64 m7fffffffe1a0_1_64 = *((UINT64 *) 0x7fffffffe1a0);
        const UINT32 m7fffffffe1b0_1_32 = *((UINT32 *) 0x7fffffffe1b0);
        if (((UINT64 (rax_1) & 0xffffffff) /*0x8*/ != 0)) {
            regs.rax = printf (/*"unknown option!
"*/ (const char *) 0x400e89);
        } else {
            regs.rax = printf (/*"Usage: program <command codes>
"*/ (const char *) 0x400e60);
        }
    }
    const UINT64 rdx_1 = regs.rdx, rsi_1 = regs.rsi, ...;
    if (istringEqual_0x80_0x20 (true, "--option", argv[2], NULL)) {
        regs.rax = printf (/*"Usage: program <command codes>
"*/ (const char *) 0x400e60);
    } else {
        regs.rax = printf (/*"Wrong option/value pair is given!"*/ (const char *) 0x400eb8);
        const INT64 rax_2 = regs.rax, rbx_2 = regs.rbx, ...;
        const UINT64 m7fffffffe1a0_2_64 = *((UINT64 *) 0x7fffffffe1a0);
        if (((UINT64 (rax_2) & 0xffffffff) /*0x8*/ != 0)) {
            regs.rax = printf (/*"Invalid value!"*/ (const char *) 0x400e2b);
        } else {
            regs.rax = printf (/*"Correct option/value pair is given!"*/ (const char *) 0x400eb8);
        }
    }
    const INT64 rax_3 = regs.rax, ...;
    const INT64 rdx_3 = regs.rdx, ...;
    const INT64 rdx_3 = regs.rdx, ...;
    const INT64 rdx_3 = regs.rdx, ...;
    if (false) {
        regs.rax = printf (/*"Usage: program <command codes>
"*/ (const char *) 0x400e60);
    } else {
        regs.rax = printf (/*"Wrong option/value pair is given!"*/ (const char *) 0x400eb8);
    }
    regs.rax = printf (/*"Usage: program <command codes>
"*/ (const char *) 0x400e60);
}

(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.
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; \( | \cdot | \) function gives the bit length; \( \text{len}() \) function gives number of used tokens.

<table>
<thead>
<tr>
<th>Visited expression</th>
<th>Simplified formula</th>
<th>Required context</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \land a )</td>
<td>0</td>
<td>( (2^{\lceil S \rceil} - 1) \land a = 0 )</td>
</tr>
<tr>
<td>( S \land a )</td>
<td>( S )</td>
<td>( (2^{\lceil S \rceil} - 1) \land a = (2^{\lceil S \rceil} - 1) )</td>
</tr>
<tr>
<td>( Z \pm a + b )</td>
<td>( Z \pm c )</td>
<td>( c = a \pm b )</td>
</tr>
<tr>
<td>( Z \land 0 )</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( Z^+_0 )</td>
<td>( Z )</td>
<td></td>
</tr>
<tr>
<td>( (X \lor Y) \land a )</td>
<td>( (X' \lor Y') \land a )</td>
<td>( X' = X \land a, Y' = Y \land a )</td>
</tr>
<tr>
<td>( \text{len}(X') \leq \text{len}(X), \text{len}(Y') \leq \text{len}(Y) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( (X + Y) \land a )</td>
<td>( (X' + Y') \land a )</td>
<td>( a = 2^b - 1 )</td>
</tr>
<tr>
<td>( X' = X \land a, Y' = Y \land a )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{len}(X') \leq \text{len}(X), \text{len}(Y') \leq \text{len}(Y) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( (X \land Y) \gg a )</td>
<td>( X' \gg a )</td>
<td>( X' = X \gg a, Y' = Y \gg a )</td>
</tr>
<tr>
<td>( \text{len}(X') \leq \text{len}(X), \text{len}(Y') \leq \text{len}(Y) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z \lor a \land b )</td>
<td>( b )</td>
<td>( a \land b = b )</td>
</tr>
<tr>
<td>( Z \land c )</td>
<td>( Z \land c )</td>
<td>( c = a \land b )</td>
</tr>
<tr>
<td>( (Z \land a) \gg b )</td>
<td>( Z \gg b )</td>
<td>( Z' = Z \land (b^2 a) )</td>
</tr>
<tr>
<td>( \text{len}(Z') \leq (Z) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( Z \gg b )</td>
<td>( Z \gg b )</td>
<td>( c = a \pm b )</td>
</tr>
<tr>
<td>( ((Z \land a) \lor b) \land c )</td>
<td>( (Z \lor d) \land e )</td>
<td>( d = b \land c )</td>
</tr>
<tr>
<td>( e = (a \lor d) \land c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ((Z \land a) \gg b) \land c )</td>
<td>( (Z \gg b) \land c )</td>
<td>( a = 2^d - 1, c = a \land c )</td>
</tr>
<tr>
<td>( ((Z \land a) \times b) \land c )</td>
<td>( (Z \land a) \times b )</td>
<td>( c = 2^d - 1,</td>
</tr>
<tr>
<td>( (Z \otimes a) \gg b )</td>
<td>( Z \gg b )</td>
<td>( c = a \gg b )</td>
</tr>
<tr>
<td>( (Z \land a) \ll b )</td>
<td>( Z \ll b )</td>
<td>( c = a \ll b )</td>
</tr>
<tr>
<td>( (Z \gg b) \gg c )</td>
<td>( Z \gg d )</td>
<td>( a = 2^d, d = b \pm c, d &gt; 0 )</td>
</tr>
</tbody>
</table>

Table 2: Virtual opcodes of the VO-protection interpreter

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Reads an argument and calls exit syscall on it</td>
</tr>
<tr>
<td>0x01</td>
<td>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</td>
</tr>
<tr>
<td>0x02</td>
<td>Reads a format string and calls printf on it</td>
</tr>
<tr>
<td>0x03</td>
<td>Reads an offset and jumps to it unconditionally</td>
</tr>
<tr>
<td>0x04</td>
<td>Reads two pointer args, compares their corresponding C strings by calling strcmp, and stores the result in the aux variable</td>
</tr>
</tbody>
</table>
Table 3: Similarity degree of pre/post-obfuscation ETG figures

<table>
<thead>
<tr>
<th>VO-Protected / Pre-Obfuscation</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0’</td>
<td>0.96</td>
<td>0.56</td>
<td>0.47</td>
<td>0.5</td>
</tr>
<tr>
<td>P1’</td>
<td>0.61</td>
<td>0.90</td>
<td>0.63</td>
<td>0.44</td>
</tr>
<tr>
<td>P2’</td>
<td>0.5</td>
<td>0.70</td>
<td>0.88</td>
<td>0.39</td>
</tr>
<tr>
<td>P3’</td>
<td>0.52</td>
<td>0.41</td>
<td>0.37</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Table 4: Execution times and relative overheads for SPEC programs analysis. The Native column presents the execution times in milliseconds without any instrumentation. The Count column indicates the execution times in seconds when an instruction counting pintool is used. The IAO columns reports the Instruction-counting Added Overhead as a factor of native execution times. The Twintool column presents the measured times when the Twintool pintool is used to instrument each SPEC test program. The TAO column reports the Twintool Added Overhead as a factor of instruction counting pintool execution times.

<table>
<thead>
<tr>
<th>Program</th>
<th>Native (ms)</th>
<th>Count (s)</th>
<th>IAO (x)</th>
<th>Twintool (s)</th>
<th>TAO (x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>astar</td>
<td>3.602</td>
<td>2.496</td>
<td>691.8</td>
<td>5.999</td>
<td>1.4</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.961</td>
<td>2.514</td>
<td>848.2</td>
<td>3.768</td>
<td>0.5</td>
</tr>
<tr>
<td>gcc</td>
<td>8.328</td>
<td>25.312</td>
<td>3038.4</td>
<td>89.03</td>
<td>2.5</td>
</tr>
<tr>
<td>gnugo</td>
<td>48.816</td>
<td>23.863</td>
<td>487.8</td>
<td>41.64</td>
<td>0.7</td>
</tr>
<tr>
<td>grover</td>
<td>2.39</td>
<td>2.222</td>
<td>928.7</td>
<td>10.968</td>
<td>3.9</td>
</tr>
<tr>
<td>hmmer</td>
<td>5.135</td>
<td>3.828</td>
<td>744.5</td>
<td>11.015</td>
<td>1.9</td>
</tr>
<tr>
<td>jm-h264ref</td>
<td>84.939</td>
<td>8.977</td>
<td>104.7</td>
<td>136.287</td>
<td>14.2</td>
</tr>
<tr>
<td>omnetpp</td>
<td>36.468</td>
<td>45.752</td>
<td>1253.6</td>
<td>1543.674</td>
<td>32.7</td>
</tr>
<tr>
<td>perl</td>
<td>5.221</td>
<td>28.234</td>
<td>5406.8</td>
<td>28.793</td>
<td>0.02</td>
</tr>
<tr>
<td>sjeng</td>
<td>85.513</td>
<td>3.814</td>
<td>43.6</td>
<td>50.094</td>
<td>12.1</td>
</tr>
<tr>
<td>xalan</td>
<td>6.058</td>
<td>15.2</td>
<td>2508.1</td>
<td>70.982</td>
<td>3.7</td>
</tr>
</tbody>
</table>
Table 5: Comparison of Analysis and Deobfuscation Solutions. The ✓ symbol indicates that mentioned solution can overcome the obfuscation technique, the × shows an unsupported feature, the - indicates inapplicable features, and ~ is for obfuscation features which cannot be reversed completely but are partially considered.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Packing</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constant Obfuscation</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Logical Obfuscation</td>
<td>×</td>
<td>~</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>~</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Branch Obfuscation</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Analysis Type: Static (S), Dynamic (D), or Both (B)</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>S</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Source Code Presentation</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadcode Elimination</td>
<td>×</td>
<td>~</td>
<td>×</td>
<td>~</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>VO- Protection</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VO’s VM Language Independence</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>VO’s VM Metamodel Independence</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Coverage/Completeness</td>
<td>✓</td>
<td>~</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>~</td>
<td>✓</td>
</tr>
<tr>
<td>Loader Analysis</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>Multi-thread Programs</td>
<td>~</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Anti-Disassembly</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Anti-Debugging</td>
<td>✓</td>
<td>×</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Biographies

Behnam Momeni received B.E. and M.Sc. degrees in computer engineering and information technology (with first rank) from Sharif University of Technology, Tehran, Iran, in 2010 and 2012, respectively. He then joined the Ph.D. program in Sharif University of Technology. He is currently a Ph.D. candidate in Safety and Security in Software and Systems Laboratory (S4Lab), Department of Computer Engineering, Sharif University of Technology, Tehran, Iran. His current research interests include information, operating system, and software security and techniques of software obfuscation and deobfuscation.

Mehdi Kharrazi received B.E. degree in electrical engineering from the City College of New York, New York, in 1999, and M.Sc. and Ph.D. 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.