DoSE: Deobfuscation based on Semantic Equivalence

Ramtine Tofighi-Shirazi  
Univ. Grenoble Alpes, CNRS, Institut Fourier, F-38000 Grenoble, France  
Trusted Labs, Meudon, France  
ramtine.tofighi-shirazi@trusted-labs.com

Philippe Elbaz-Vincent  
Univ. Grenoble Alpes, CNRS, Institut Fourier  
F-38000 Grenoble, France  
philippe.elbaz-vincent@univ-grenoble-alpes.fr

Maria Christofi  
Oppida  
Montigny-le-Bretonneux, France  
maria.christofi@opidia.fr

Thanh-ha Le  
Trusted Labs  
Meudon, France  
thanhh.le@trusted-labs.com

ABSTRACT
Software deobfuscation is a key challenge in malware analysis to understand the internal logic of the code and establish adequate countermeasures. In order to defeat recent obfuscation techniques, state-of-the-art generic deobfuscation methodologies are based on dynamic symbolic execution (DSE). However, DSE suffers from limitations such as code coverage and scalability. In the race to counter and remove the most advanced obfuscation techniques, there is a need to reduce the amount of code to cover. To that extend, we propose a novel deobfuscation approach based on semantic equivalence, called DoSE. With DoSE, we aim to improve and complement DSE-based deobfuscation techniques by statically eliminating obfuscation transformations (built on code-reuse). This improves the code coverage. Our method’s novelty comes from the transposition of existing binary differencing techniques, namely semantic equivalence checking, to the purpose of the deobfuscation of untreated techniques, such as two-way opaque constructs, that we encounter in surreptitious software. In order to challenge DoSE, we used both known malwares such as Cryptowall, WannaCry, Flame and BitCoinMiner and obfuscated code samples. Our experimental results show that DoSE is an efficient strategy of detecting obfuscation transformations based on code-reuse with low rates of false positive and/or false negative results in practice, and up to 63% of code reduction on certain types of malwares.

KEYWORDS
Obfuscation, deobfuscation, reverse engineering, malware analysis, symbolic execution, opaque predicate, control-flow graph, code cloning

1 INTRODUCTION
Context. Software obfuscation [8] is an information management strategy that aims at obscuring the meaning that can be drawn from a software or a code, while preserving its functionality. Obfuscation transformations can be used in specific contexts and for different purposes, such as the improvement of software security, the protection against software alteration or the protection of the intellectual property. However, since its main purpose is to protect software against reverse-engineering, obfuscation is also widely used by malwares to prevent their detection and analysis.

Thus, recent binary deobfuscation techniques [4, 12] based on dynamic symbolic execution emerged in order to face obfuscation techniques such as code virtualization [34, 35, 44] or control-flow flattening [25, 45]. Generic deobfuscation methods have appeared in order to deobfuscate protected binaries. Such techniques can either extract protected code [19] or reduce the complexity of a control-flow graph of an obfuscated binary [48], contributing in this way to an improved analysis. The first step of most generic deobfuscation methods consists in generating execution traces of a protected binary. Using forward and backward taint analysis [43], only the instructions manipulating the inputs are collected. Based on these traces, an initial control-flow graph is built, which can then be completed using dynamic symbolic execution (otherwise called concolic execution) combined with a constraint solver.

Yadegari et al. [49], in their methodology, also use control dependency analysis in order to handle obfuscation transformations such as implicit flow, or call/return tampering. Code optimizations and simplifications are then applied on the generated traces in order to build a reduced control-flow graph. Salwan et al. [19], [40] add transformations to the LLVM intermediate representation [24], allowing them to build a deobfuscated binary directly from the collected traces. This technique succeeds against most of the Tigress challenges [7]. Other deobfuscation techniques and tools [4, 28], specific to opaque predicates [10], are also based on dynamic symbolic execution. They consist in collecting traces of the binaries and slices of instructions depending on a given predicate to test its opaqueness. Using a constraint solver, they verify if the branches of a predicate is feasible or infeasible in order to remove any unreachable paths within the binary.

Motivation. Generic deobfuscation techniques based on DSE often needs execution trace, which requires inputs generation. This may be time consuming and make code coverage and scalability the main issues of those techniques. Moreover, in the context of malware analysis, DSE is confronted to network event based components and conditions (e.g. connection to a command and control server) which makes the deobfuscation more difficult in terms of scalability. Besides, novel obfuscation techniques exploit these limitations to further hinder the analyses [3]. Their goal is to divide the number of paths, forcing dynamic symbolic execution engines to slow down when trying to cover all the code.

Contributions. We propose a novel deobfuscation method based on semantic equivalence, called DoSE. The novelty of our contribution is built on the application of binary differencing techniques based
on semantic equivalence to deobfuscate binaries. Our transposition of existing binary timing techniques allows us to provide a concrete methodology to statically detect and remove protections based on code-reuse (c.f. Sections 2.1 to 2.3). Some of these protections are not handled by current deobfuscation methodologies, while others aim at preventing generic ones. Our approach, in contrast to the current deobfuscation techniques, threat also novel obfuscation transformations based on code-reuse and detects two-way opaque predicates constructs for which no deobfuscation methodology exists. We implemented DoSE as an IDA plug-in and applied it to different families of recent malwares in order to illustrate the significant reduction of the amount of code to cover. We also discuss how it can be used to combine and complement existing generic deobfuscation techniques.

This paper is organized as follows:

- First, we present background information about obfuscation techniques based on code-reuse such as range dividers and two-way opaque predicates, highlighting the need to analyze and deobfuscate them. We also discuss the utility of such methods in other use cases such as white-box cryptography (Section 2).
- Second, we propose our methodology of Deobfuscation based on Semantic Equivalence (i.e. DoSE). Formal definitions of our core methodology are given (Section 3) along with some enhancements improving the efficiency and precision of DoSE (Section 3.3).
- Third, we present concrete applications of DoSE, namely control-flow graph reduction (Section 4.1), two-way opaque predicate removal (Section 4.2) and cloned sub-functions detection (Section 4.3). Each application contains a detailed explanation of our approach and an evaluation on real-world malwares.
- Finally, we close this paper with a discussion on our perspectives and conclusions regarding DoSE (Sections 5 and 6).

2 PROBLEM SETTING

Collberg et al. [9] define code obfuscation as follows:

Let \( P \xrightarrow{T} P' \) be a transformation \( T \) of a source program \( P \) into a target program \( P' \). We call \( P \xrightarrow{T} P' \) an obfuscating transformation if \( P \) and \( P' \) have the same observable behavior.

Consequently, the following conditions must be fulfilled for an obfuscating transformation : if \( P \) fails to terminate, or terminates with an error condition, then \( P' \) may or may not terminate; otherwise, \( P' \) must terminate and produce the same output as \( P \).

Several obfuscation transformations exist, each of them having their own purposes : obfuscate the layout, the data or the control-flow of a program. Other obfuscation techniques aim at counterering existing tools or known deobfuscation methods. A classification of all the obfuscation techniques, as well as known deobfuscation methods with their different purposes, has been provided by S. Schrittwieser et al. [41].

In this section, we will study code cloning and its combination with obfuscation techniques such as range dividers [3], before showing why they cannot be detected by existing techniques and how DoSE can contribute. We will then present the benefits of such detection to the simplification of control-flow graphs, or even the removal of bogus branch functions. In the next sections, we will also focus on opaque predicates [10, 36] and more precisely on two-way constructs since most recent opaque predicate detection analyses and tools [4, 28] do not handle such type of constructions.

2.1 Code cloning

Code cloning or copying is a widely used obfuscation technique [42] consisting in diversifying paths of the program in order to increase the amount of code an attacker has to analyze. The cloned parts of the code are often syntactically different but shall remain semantically equivalent, meaning that, from a functional point of view, the original portion of the code and its clone are the same.

To prevent them from being syntactically equivalent, code cloning is often combined with other obfuscation transformations such as instruction re-ordering or dead code insertion.

Code cloning, as an obfuscation technique, can also be used implicitly with other obfuscation transformations such as control-flow flattening or opaque predicates [8]. Other uses of code cloning consist in duplicating small functions or in creating semantically equivalent input-dependent paths within a binary in order to prevent state-of-the-art generic deobfuscation techniques [48].

A good example of code cloning as an obfuscation transformation can be found in the most resilient challenge of the CHES 2017 "Capture the flag" WhibOx Contest [11], which consists in building and evaluating white-box AES-128 [13] implementations. This challenge\(^1\), in order to prevent reverse-engineering, implements over 1200 small functions, referred to as sub-functions (i.e. branch functions), among which 1180 are semantically equivalent (i.e. clones). It also implements virtualization, dummy operations and renaming to further obfuscate the code.

```c
1  void wGzZ(uint oEHmwk, uint KCZu, uint MtCA) { ooGoRv[ ( kIKfgI+oEHmwk)&262143]= ooGoRv[ ( kIKfgI+KCZu) &262143](ooGoRv[ ( kIKfgI+MtCA)&262143]);}
2  ...
3  void pzwSZ(uint eCFI, uint picb, uint aqUUV) { ooGoRv[ [ kIKfgI+eCFI]&262143]= ooGoRv[ ( kIKfgI+picb)&262143]*
 \  ooGoRv[ ( kIKfgI+aqUUV)&262143];}
```

Listing 1: Example of two cloned sub-functions from the challenge adoring_poitras of the WhibOx contest.

Listing 1 illustrates two of these cloned sub-functions. We will show in Section 4.3 that extending our approach to detect semantically equivalent sub-functions, can allow us to statically simplify and deobfuscate binary code, and in this example, proceed to the key extraction of the challenge.

For readers convenience, the next paragraphs recall the obfuscation techniques based on semantically equivalent code that are discussed in the paper, namely range dividers and two-way construct opaque predicates.

2.2 Range dividers

Range dividers is a novel obfuscation transformation, introduced by Banescu et al. [3], which exploits the limitations of generic deobfuscation techniques, such as path explosion, code coverage and complex constraints. Range dividers are input-based condition

---

\(^1\)Source code is available at https://run.whibox.cvpr.to/5443/show/candidate/777.
branches that cause symbolic execution engines to explore more feasible paths, thus slowing it down.

However, in order to preserve the functionality property of an obfuscator, equivalent instruction sequences are used in all branches of range dividers, as illustrated in Listing 2. Such construction illustrates that being able to detect and merge cloned blocks allows the deobfuscation of these obfuscation transformations, along with reducing the number of paths to explore and the number of inputs to generate. These properties are crucial for the construction of a generic deobfuscation technique in order to have a wide code coverage and prevent too much slowdown from the symbolic execution engine.

```c
unsigned char *str = argv[1];
unsigned int hash = 0;
for(int i=0; i<strlen(str); str++, i++) {
    char chr = *str;
    if (chr > 42) {
        hash = (hash << 7) ^ chr;
        // semantically equivalent to else case
    } else {
        hash = (hash * 128) ^ chr;
        // semantically equivalent to if case
    }
}
if (hash == 889267)
    printf ("win \n\n");
Listing 2: S. Banescu et al. illustration of range dividers [3]
```

Our approach aims at removing this novel obfuscation technique by detecting and grouping clones.

2.3 Two-way opaque predicates

Opaque predicates [10, 28, 36] are a fundamental illustration of the implication of code-reuse in software obfuscation. Such transformations are defined as expressions whose values are known by the defender, but hard to deduce for an attacker. There are different kinds of opaque predicates. Collberg et al. defined $P^E$, $P^T$ and $P^F$ as being opaque predicates that are always evaluated to false, true or unknown (either true or false) respectively.

The latter construction of opaque predicates $P^F$ are called two-way opaque predicates and are a current limitation to state-of-the-art analysis and tools that only handle predicates of type $P^T$ and $P^F$. Moreover, since they use constraint solvers to check feasibility or infeasibility of each path, they are currently limited to arithmetic-based predicates, while other types of opaque predicates (e.g. MBA-based [51]) cannot be analyzed.

Figure 1 illustrates an example of a two-way predicate where the value of $(p)\%2$ depends on the allocated memory area. The predicate can be evaluated to either true or false. However, both branches are semantically equivalent, meaning that no matter the value of the predicate, a same entry will produce the same output for both branches. We will present how semantic-based comparison can be extended to detect and remove such constructions of opaque predicates which are currently not handled by state of the art deobfuscation techniques [4, 28].

2.4 Binary diffing techniques

Detecting clones between binaries has a wide variety of applications such as software development [37, 47], software plagiarism detection [26, 46], vulnerabilities exploration [6, 32, 33] and malware variant detection [2, 15, 18]. Different comparison approaches have been published, either syntax-based (i.e. text-based) for example by measuring instruction sequences [31] or using byte n-grams [30, 20], metrics-based [15, 39] or structure-based [22, 52]. While the previous comparison techniques can be defeated with obfuscation or even with code optimizations, more recent methods use semantic-based approaches since, by definition, an obfuscation transformation should preserve the logic of the original program.

**Semantic-based comparison.** Semantic-based comparison methods disassemble the binaries to be compared before extracting the logic of their instructions (i.e. the semantics) using an intermediate representation of the assembly language. From this intermediate representation, one first analyzes the basic blocks to express their inputs to outputs behavior using symbolic execution [21]. Once the input to output expressions are generated, a constraint solver is used to check the equivalence between the basic blocks. This method has been first introduced by Gao et al. [16] as a static analysis in order to detect plagiarism between a set of binaries. It has since been modified, optimized [23] and extended to dynamic analysis combined with taint techniques, either to accept more noise [26], or to be more efficient [27, 29]. Our work is built on these approaches for the purpose of deobfuscating binaries. The novelty of DoSE comes from the transposition and a combination of binary diffing techniques, used statically and optimized for the purpose of deobfuscation, as presented in the following section.

3 DOSE: DEOBFUSCATION BASED ON SEMANTIC EQUIVALENCE

In this section, we present a new method for deobfuscation using semantic equivalence comparisons. We call our methodology DoSE, for Deobfuscation based on Semantic Equivalence. DoSE consists in several steps: syntactic equivalence, semantic equivalence and conditional equivalence. We start by formalizing syntax-based basic blocks comparisons to afterwards introduce the semantic-based approach. Then, we present our improvements based on conditional equivalence.
equivalence checking to prevent false positives, combined with normalization and optimizations steps to eliminate false negatives, and prevent too much slowdown. DoSE, in one hand simplifies and deobfuscates the code and on the other hand, makes generic DSE-based deobfuscation techniques more scalable and efficient.

3.1 Syntax-based basic blocks comparison

Syntax-based comparison relies on the assembly code of the basic blocks. In order to define the syntactic equivalence between two basic blocks, we start by defining the inclusion of a basic block into another. Furthermore, we define an inclusion score in order to quantify the number of included instructions. In the following definitions we use the notations syn for syntax, sem for semantic and cond for conditional.

Definition 1: Syntactic Inclusion. Let B and B’ be two basic blocks and let \( \text{I}_n \) be the \( n \)-th instruction of B and \( \text{I}_m \) the \( m \)-th instruction of B’, \( m, n \in \mathbb{N} \). We say that B is syntactically included in B’ if for all \( \text{I}_n \in B \), there exists a \( \text{I}_m \in B’ \) such that \( \text{I}_m = \text{syn} \text{I}_n \), with \( m = n \), and we set \( B \subseteq \text{syn} B’ \). In other words, \( \text{I}_m = \text{syn} \text{I}_n \) with \( m = n \) means that we have exactly the same instruction at the same position (i.e. same order).

Definition 2: Syntactic Inclusion Score. In order to measure the inclusion of two basic blocks B and B’, we need to define a score. Let \( \sigma_{\text{syn}}(B, B’) \) be the syntactic inclusion score of B compared to B’, \( N \) the number of equivalent instructions between B and B’, and \(|B|\) and \(|B’|\) the number of instructions of B and B’ respectively. Then \( \sigma_{\text{syn}}(B, B’) = \frac{N}{|B’|} \). As an example, \( \sigma_{\text{syn}}(B, B’)=1 \) means that all the instructions of B are included in B’.

Definition 3: Syntactic Equivalence. Let B and B’ be two basic blocks. If \( B \subseteq \text{syn} B’ \) and \( B’ \subseteq \text{syn} B \) then we write \( B = \text{syn} B’ \), meaning that both basic blocks are equivalent (i.e. B is a clone of B’ and vice versa). Syntactic equivalence between two basic blocks can also be represented by \( \sigma_{\text{syn}}(B, B’)=\sigma_{\text{syn}}(B’,B)=1 \).

Obviously, such method is not resilient to obfuscation techniques and the probability that we will find equivalent basic blocks based on their syntax may be low. However, in the context of an evaluation, starting by simple methods is coherent since it can sometimes discard semantic-based analysis, which requires more resources and more time.

3.2 Semantic-based basic blocks comparison

As opposed to syntactic equivalence, comparisons based on semantic equivalence rely on an intermediate representation of a basic block, since it uses symbolic execution combined with a constraint solver in order to verify the equivalence between the computed expressions. Thus, the inputs of basic blocks are treated as symbols while the output of the symbolic execution returns a set of expressions that represents the input-output relations of these basic blocks.

Definition 4: Semantic Inclusion. Let B and B’ be two basic blocks and let \( IR_B \) and \( IR_{B’} \) be the intermediate representation of B and B’ respectively after their symbolic execution. Let \( X_B \) and \( Y_{B’} \) be two sets of all outputs expressions of \( IR_B \) and \( IR_{B’} \) respectively. Let \( x_i \in X_B \) be the i-th output expression of \( IR_B \) and \( y_j \in Y_{B’} \) be the j-th output expression of \( IR_{B’} \), \( i, j \in \mathbb{N} \) (note that \( i = j \) or \( i \neq j \)). We can say that B is semantically included in B’ if \( \forall x_i \in X_B \), there exists a unique \( y_j \in Y_{B’} \) such that \( y_j = \text{sem} x_i \) and we set \( B \subseteq \text{sem} B’ \). The semantic inclusion between two expressions is verified using a SMT solver.

Definition 5: Semantic Inclusion Score. Based on the same principle as for the syntax-based comparison, we define a score for semantic-based basic block inclusion. Let \( \sigma_{\text{sem}}(B, B’) \) be the semantic inclusion score function of B compared to B’, \( N \) the number of equivalent output expressions and \(|X_B|\) and \(|Y_{B’}|\) the number of output expressions of B and B’ respectively. Then \( \sigma_{\text{sem}}(B, B’)=\frac{N}{|X_B|} \).

Definition 6: Semantic Equivalence. As in the definition of syntactic equivalence, two semantically equivalent basic blocks, or cloned basic blocks, can be represented by \( \sigma_{\text{sem}}(B, B’)=\sigma_{\text{sem}}(B’, B)=1 \), meaning that \( B \subseteq \text{sem} B’ \) and \( B’ \subseteq \text{sem} B \). Our approach tries all possible pairs to find if there exists a bijective mapping between the output expressions of B and B’.

In order to achieve a complete analysis of two basic blocks, we start by comparing their syntax. If the syntax-based comparison fails, we use the semantic equivalence along with our conditional equivalence step. The latter is an improvement which is introduced in Section 3.3.

3.3 Minimizing false positive/negative rates

As it is the case for any analysis, false positive or false negative results may occur. Our objective is to reduce them as much as possible.

3.3.1 False positive prevention: Conditional-equivalence. A false positive means that two basic blocks labeled as clones may actually have different purposes. Since our context requires strict equivalence in order to remove cloned blocks within a function, it is important to have a good correctness. Our semantic equivalence step is efficient in finding functionality-equivalent portion of code but regardless of the memory area used, or of the function called within the blocks. Thus, in some cases, functionality-equivalent codes may use different values which may generate different outputs. Such example is given in Figure 2, where the two blocks compute the same operations using different memory areas.

![Figure 2: Example of two functionally equivalent basic block using different memory areas, from Vipasana ransomware.](image)

We choose to treat such type of code as false positives, since it is statically undecidable whether two different memory areas used
contain the same values, or two calls to different functions return the same values. Thus, our conditional equivalence step consists in replacing all inputs (e.g., memory areas, registers, return value of a function) by randomly generated concrete values in order to verify whether two blocks compute the same outputs. If it is the case, then we can conclude that the two blocks are equivalent under a given condition (i.e., the concrete value). A similar technique is already used in the context of binary diffing [29]. Definitions of the conditional inclusion score and equivalence are similar to the semantic definitions and are given below.

Definition 7: Conditional Inclusion. Let \( B \) and \( B' \) be two basic blocks and let \( IR_B \) and \( IR_{B'} \) be the intermediate representation of \( B \) and \( B' \) respectively after their symbolic execution. Let \( X_B \) and \( Y_{B'} \) be two sets of all outputs expressions of \( IR_B \) and \( IR_{B'} \) respectively. Let \( x_i \in X_B \) be the \( i \)-th output expression of \( IR_B \) and \( y_j \in Y_{B'} \) be the \( j \)-th output expression of \( IR_{B'} \), \( i, j \in \mathbb{N} \) (note that \( i = j \) or \( i \neq j \)). Let \( C \) be a concretization function which replaces all symbols of a given output expression \( x \) by random concrete values. We say that \( B \) is conditionally included in \( B' \) if for all \( x_i \in X_B \), there exists a unique \( y_j \in Y_{B'} \) such that \( C(y_j) = C(x_i) \) and set \( B \sqsubseteq_{\text{cond}} B' \).

Definition 8: Conditional Inclusion Score. Let \( \sigma_{\text{cond}}(B, B') \) be the conditional inclusion score function of \( B \) compared to \( B' \). \( N \) the number of equivalent output expressions injected with concrete values and \( |X_B| \) and \( |Y_{B'}| \) (the number of output expressions of \( B \) and \( B' \) respectively) such that \( \sigma_{\text{cond}}(B, B') = \frac{N}{|X_B|} \). As an example, if \( \sigma_{\text{cond}}(B, B') = 1 \), then we say that \( B \sqsubseteq_{\text{cond}} B' \), meaning that \( B \) is conditionally included in \( B' \).

Definition 9: Conditional Equivalence. Two conditionally equivalent basic blocks can be represented by \( \sigma_{\text{cond}}(B, B') = \sigma_{\text{cond}}(B', B) = 1 \), meaning that \( B \sqsubseteq_{\text{cond}} B' \) and \( B' \sqsubseteq_{\text{cond}} B \) under the condition of the injected concrete values.

Conditional equivalence step can be added after the comparisons based on semantic equivalence in order to confirm that two given basic blocks do represent clones. This step allows DoSE to find codes that are equivalent with respect to the values used and also to prevent false positives. Since DoSE aims at contributing and completing generic deobfuscation techniques based on dynamic analysis, we can note that the blocks that are semantically but not conditionally equivalent need further analysis during the dynamic analysis in order to verify their equivalence.

3.3.2 False negative prevention: Normalization and optimizations.
False negatives are another downside of comparisons based on semantic equivalence. They represent basic blocks that are not considered as clones (i.e., semantically equivalent) when in fact they are. This limitation does not impact on the quality of our approach as all results will indeed be real clones, but its efficiency may be questioned as some clones may not be detected. In order to prevent this issue, we add some normalization steps for both syntax and semantic equivalence comparisons.

The normalization step for syntax-based comparisons aims at removing any unnecessary instructions (such as \texttt{nop} instructions) or destination addresses for \texttt{jmp} instructions (since two cloned basic blocks may jump to different blocks located at different addresses).

Moreover, we symbolize the registers used by the instructions in order to handle register substitution without using the semantic equivalence step. This allows us to have better performances since we do not query the SMT solver. An illustration of the syntactic normalization is given in Figure 3.

**Figure 3:** Normalization of the syntax of two blocks which are cloned.

The normalization phase for semantic equivalence comparisons consists in the following steps:

- symbolize all variables, registers, memory access used by the basic blocks;
- keep the concrete values of immediate values;
- use constant propagation on the intermediate language;
- use arithmetic simplifications on the intermediate language.

These optimization and simplification techniques allow us to improve the precision of DoSE in the purpose of preventing false negative results, as well as optimizing the performances. Table 1 illustrates the differences in execution time and false negatives and positives results of our method, before and after our improvements.

Our improvements are an important step toward the detection and removal of obfuscation transformations based on code-reuse. The next section presents some applications of our methodology along with their evaluation.

4 APPLICATIONS

In this section we present some concrete applications of DoSE. We show how it can be used to reduce control-flow graphs, detect and remove two-way opaque predicates as well as cloned sub-functions. For each application, we illustrate our process based on DoSE, along with their respective evaluations. DoSE is implemented as an IDA Pro [17] plug-in, based on the reverse-engineering framework Miasm [14] in order to be easily integrated in other reverse-engineering and deobfuscation frameworks. All our evaluations are done on a Windows 7 virtual machine, using 8gb of RAM, and a Intel vPro i7 CPU.

4.1 Reducing control-flow graphs
Reducing control-flow graphs by grouping similar nodes can ease the understanding of the code and eliminate some paths for further dynamic analysis, thus contributing to generic deobfuscation techniques. Since some obfuscation transformations generate equivalent basic blocks, we extended our methodology to the static reduction of control-flow graphs by detecting and grouping such blocks. Moreover, in another context, e.g., the evaluation of cryptographic white-box implementations, there is a need for clone removal.

4.1.1 Methodology. Our methodology for reducing control-flow graphs is based on static clone detection and is divided in two parts. The first part collects needed information about the obfuscated
function to analyze. This information is then transmitted to the second step which performs the comparisons in order to detect clones. In the remaining of this section, we will describe these steps.

Basic blocks collection. Given a function \( F \) that we want to analyze, we start by collecting all basic blocks of the function. For each basic block \( B \) of \( F \), we gather both its instructions \( I_B \) and its associated intermediate representation \( IR_B \). The collected instructions will be normalized in order to be compared syntactically. Their intermediate language will be first simplified, to prevent any false positive results, before being used as input for the symbolic execution engine. The latter will return the expressions that illustrate the inputs and outputs behavior (i.e. functionality) of a basic block. These expressions, that we note \( X_B \), will then be processed by our normalization phase before being compared to find semantic equivalences. All of the basic blocks are represented by a structure that will contain all gathered information (i.e. \( I_B, IR_B \) and \( X_B \)). Based on this structure, we initialize a list \( L \) containing the collected information for each \( B \), so that it can be used as input for the comparison method.

Algorithm 1 illustrates the pseudo-code for our static clone detection technique, given an obfuscated function \( F \). More precisely, it shows how information is gathered and analyzed in order to perform syntactic along with semantic equivalence comparisons. Moreover, it includes both simplification and normalization steps in order to prevent any misleading results (i.e. false positives and false negatives).

Basic blocks comparisons. Once the first step is done, we proceed to the comparisons, using the list \( L \) of all basic block structures. Once the two basic blocks named \( B \) and \( B' \) are selected, we check whether they are located at the same addresses within the binary or if they already have been analyzed in order to avoid unnecessary computations. Since we require a bijective mapping between \( B \) and \( B' \), we can also verify whether these two blocks have the same number of instructions (i.e. \( |B| = |B'| \)) or the same number of output expressions (i.e. \( |X_B| = |X_{B'}| \)). If two blocks pass those tests, we proceed to the syntactic comparison. If the syntactic inclusion score is 1 for \( B \) compared to \( B' \), and vice-versa, then we assume that these blocks are clones and we add them to our dictionary \( C \) which groups all detected cloned blocks. However, if the syntax-based comparison fails at determining that \( B \) and \( B' \) are equivalent, we proceed to the semantic equivalence comparison in order to verify the inclusion between the selected blocks. If those blocks are semantically equivalent, we use the concretization function in order to prevent false positives. This function replaces the symbols of each expression by concrete values in order to check for a conditional equivalence. Only if \( B \) and \( B' \) are equivalent both semantically and conditionally, we assume that the basic blocks are clones and update the dictionary \( C \). If one of those verification steps fails, we consider that the selected basic blocks are not clones and move on to the next couple of basic blocks. The different verification steps are described in Section 3.

Algorithm 2 illustrates the second part of our methodology. It returns the dictionary \( C \) of detected clones in order to remove them. The next section will present the evaluation of semantic equivalence comparison for the purpose of reducing control-flow graphs.

4.1.2 Evaluations. To illustrate the efficiency of our analysis, we used several malware samples\(^3\) among Flame [5] and Cryptowall [50] as shown in Table 2. These malwares were selected according to their availability. We analyzed some functions of these samples, with their entry-points listed in column “Function EP” for reproducibility. These functions have been selected for their large sizes in order to measure the scalability of DoSE. Column “\# Nodes” indicates the number of basic-blocks of each function before the application of DoSE whereas “\% Reduction” illustrates the efficiency of our approach for detecting and grouping semantically equivalent basic blocks within the control-flow graph of each function. Finally, the last columns show a pair representing the false positive and

---

Table 1: Differences of false positives, false negatives results and execution time before and after our improvements (i.e. conditional-equivalence, normalization and optimizations), based on control-flow graph reduction of several malware functions.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Function</th>
<th>(#FP, #FN) before</th>
<th>time (s) before</th>
<th>(#FP, #FN) after</th>
<th>time (s) after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asprox</td>
<td>0x10009b82</td>
<td>(5,0)</td>
<td>48.03s</td>
<td>(0,0)</td>
<td>18.14s</td>
</tr>
<tr>
<td>Asprox</td>
<td>0x1000b3e35</td>
<td>(32,2)</td>
<td>1851.09s</td>
<td>(0,0)</td>
<td>243.32s</td>
</tr>
<tr>
<td>Flame</td>
<td>0x1003177b</td>
<td>(6,1)</td>
<td>230.84s</td>
<td>(0,0)</td>
<td>26.14s</td>
</tr>
<tr>
<td>Wanna.Cry</td>
<td>0x04043b6</td>
<td>(20,0)</td>
<td>124.06s</td>
<td>(0,0)</td>
<td>23.48s</td>
</tr>
<tr>
<td>CryptoWall</td>
<td>0x0011000</td>
<td>(5,11)</td>
<td>227.57s</td>
<td>(0,3)</td>
<td>67.21s</td>
</tr>
<tr>
<td>Vipasaña</td>
<td>0x029954</td>
<td>(6,7)</td>
<td>106.95s</td>
<td>(0,5)</td>
<td>24.40s</td>
</tr>
</tbody>
</table>

Algorithm 1 Control-flow graph reduction

1: procedure CLONE DETECTION(\( F \): a function)
2: Initialize a dictionary \( C \) to store clones
3: Initialize a list \( L \) of basic block structures
4: for each basic block \( B \) in \( F \) do
5: \( I_B \) \( \leftarrow \) GetInstructions(\( B \))
6: \( \text{NormalizeInstructions}(I_B) \)
7: \( IR_B \) \( \leftarrow \) GetIntermediateLanguage(\( I_B \))
8: \( \text{Simplify}(IR_B) \)
9: \( X_B \) \( \leftarrow \) SymbolicExecution(\( IR_B \))
10: \( \text{NormalizeSemantics}(X_B) \)
11: \( L[B] \) \( \leftarrow \) \( I_B, IR_B, X_B \)
12: end for
13: \( C \) \( \leftarrow \) Syntactic and semantic equivalence comparisons(\( L \))
14: // see Algorithm 2.
15: return \( C \)
16: end procedure

---

\(^3\)Samples are available at https://github.com/lanaram/DoSE
### Table 2: Evaluation of static control-flow graph reduction using DoSE

<table>
<thead>
<tr>
<th>Sample</th>
<th>Type</th>
<th>Function EP</th>
<th># Nodes</th>
<th>% Reduction</th>
<th>(#FP, #FN)</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitCoinMiner</td>
<td>Trojan</td>
<td>0x40a900</td>
<td>97</td>
<td>52.58%</td>
<td>(0,0)</td>
<td>25.42s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x407240</td>
<td>697</td>
<td>47.06%</td>
<td>(0,0)</td>
<td>933.49s</td>
</tr>
<tr>
<td>Hupigon</td>
<td>Backdoor</td>
<td>0x49935c</td>
<td>321</td>
<td>58.57%</td>
<td>(0,0)</td>
<td>141.00s</td>
</tr>
<tr>
<td>Asprox</td>
<td>Trojan</td>
<td>0x1000be55</td>
<td>436</td>
<td>41.97%</td>
<td>(0,0)</td>
<td>243.32s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10009f82</td>
<td>57</td>
<td>45.61%</td>
<td>(0,0)</td>
<td>18.14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100096a5</td>
<td>67</td>
<td>20.90%</td>
<td>(0,0)</td>
<td>14.01s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100091ac</td>
<td>33</td>
<td>39.39%</td>
<td>(0,0)</td>
<td>1.38s</td>
</tr>
<tr>
<td>Asprox</td>
<td>Trojan</td>
<td>0x1009c70</td>
<td>113</td>
<td>33.63%</td>
<td>(0,0)</td>
<td>13.14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10096c0</td>
<td>44</td>
<td>18.18%</td>
<td>(0,0)</td>
<td>3.39s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1006d0</td>
<td>30</td>
<td>23.33%</td>
<td>(0,0)</td>
<td>2.57s</td>
</tr>
<tr>
<td>Vipasana</td>
<td>Ransomware</td>
<td>0x429954</td>
<td>95</td>
<td>25.26%</td>
<td>(0,5)</td>
<td>24.40s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x425b50</td>
<td>80</td>
<td>40.00%</td>
<td>(0,0)</td>
<td>6.46s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x424fc0</td>
<td>64</td>
<td>25.00%</td>
<td>(0,0)</td>
<td>7.51s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x4278a0</td>
<td>63</td>
<td>23.81%</td>
<td>(0,0)</td>
<td>20.14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x42d578</td>
<td>60</td>
<td>33.30%</td>
<td>(0,0)</td>
<td>6.09s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x4399f8</td>
<td>123</td>
<td>63.41%</td>
<td>(0,0)</td>
<td>43.52s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x42be04</td>
<td>59</td>
<td>50.85%</td>
<td>(0,0)</td>
<td>6.20s</td>
</tr>
<tr>
<td>Cryptowall</td>
<td>Ransomware</td>
<td>0x401100</td>
<td>179</td>
<td>44.13%</td>
<td>(0,3)</td>
<td>67.21s</td>
</tr>
<tr>
<td>Flame</td>
<td>Worm</td>
<td>0x100586ea</td>
<td>365</td>
<td>21.64%</td>
<td>(0,0)</td>
<td>58.44s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x100317b</td>
<td>157</td>
<td>29.30%</td>
<td>(0,0)</td>
<td>26.14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x10023fd6</td>
<td>29</td>
<td>31.03%</td>
<td>(0,0)</td>
<td>4.14s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1006e7b9</td>
<td>100</td>
<td>36.00%</td>
<td>(0,0)</td>
<td>15.60s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1004949f</td>
<td>54</td>
<td>37.04%</td>
<td>(0,0)</td>
<td>3.26s</td>
</tr>
<tr>
<td>WannaCry</td>
<td>Ransomware</td>
<td>0x4043b6</td>
<td>123</td>
<td>16.26%</td>
<td>(0,0)</td>
<td>23.48s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x403bfc</td>
<td>98</td>
<td>35.71%</td>
<td>(0,0)</td>
<td>12.30s</td>
</tr>
<tr>
<td>Dexter</td>
<td>Trojan</td>
<td>0x1004a0</td>
<td>86</td>
<td>27.91 %</td>
<td>(0,0)</td>
<td>25.55s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1002050</td>
<td>33</td>
<td>18.18%</td>
<td>(0,0)</td>
<td>5.00s</td>
</tr>
<tr>
<td>OnionDuke</td>
<td>Trojan</td>
<td>0x10005b00</td>
<td>76</td>
<td>38.16%</td>
<td>(0,0)</td>
<td>11.56s</td>
</tr>
</tbody>
</table>

**Algorithm 2** Basic blocks comparisons

1. procedure Syntax and semantic equivalence comparisons(L: List of basic blocks)
2. Initialize a dictionary C of clones
3. for each basic block B in L do
4.   for each basic block B’ in L do
5.     if AlreadyCompared(B’, B) = False then
6.       if σ_{syn}(B, B’) = σ_{cond}(B’, B) = 1 then
7.         C[B] ← B’ // add B’ as a clone of B
8.       C[B] ← B // add B as a clone of B’
9.     else if σ_{syn}(B, B’) = σ_{cond}(B’, B) = 1 then
10.        C[B] ← B’ // add B’ as a clone of B
11.        C[B] ← B // add B as a clone of B’
12.     else
13.         pass // B’ is not a clone of B.
14.     end if
15.   end for
16. end if
17. end procedure

For each application of DoSE, positive and negative results were verified by using heuristics based on the transitivity property of an equivalence. The inclusion scores are also used to facilitate the detection of false negatives. We also proceed with mainly manual reverse engineering to verify our results.

As shown in Table 2, DoSE can reduce in most of the cases one-third of the malware functions control-flow graphs with no false positives in practice and only a few false negative results. In some cases, such as Vipasana sample, we reduced 63.41% of a function’s control-flow graph. Figure 4 illustrates the application of DoSE on Cryptowall main function. The tagged CFG illustrates the detected cloned blocks (with one color for each group). We can see that DoSE is quite efficient in reducing the amount of paths to cover (46 similar paths are removed), and grouping cloned blocks (78 of the 179 basic blocks are clones) in an acceptable amount of time (approximately 1 minute). DoSE can also scale to more complex functions, as illustrated with the BitCoinMiner sample, on which we are able to reduce 47.06% of the 697 basic-blocks with no false positive/negative results, in approximately 15 minutes.

**4.1.3 Limitations.** One limitation of DoSE is its block-centric approach. Indeed, some malware such as the Vipasana ransomware combine opaque predicates with code cloning, thus some clones are divided into several basic blocks with no direct successors. Since DoSE compares each basic block, such type of clones is not detected which explains the false negatives results in our evaluations. We believe that by extending our analysis on paths, it will be possible...
we present some notations. We denote by $\phi$ a two-way opaque predicate, then $\omega$ is the opaque predicate as illustrated in Algorithm 3. Indeed, if $\phi$ is a two-way opaque predicate, then $\omega_T$ and $\omega_F$ must end either on a common block or on a returning block\(^4\). Moreover, since we aim at comparing basic blocks, we iterate only once over an encountered loop.

### Algorithm 3 Two-way predicate detection

1. procedure **Two-way predicate detection**($D$: disassembly of the targeted binary)
2. Initialize a dictionary \( R \) to store the results
3. for each $\phi_n$ in \( D \) do
4. \( \omega_T \leftarrow \text{GetTruePaths}(\phi_n) \)
5. \( \omega_F \leftarrow \text{GetFalsePaths}(\phi_n) \)
6. \( R[\phi_n] \leftarrow \text{PathEquivalenceChecking}(\omega_T, \omega_F) \)
7. end for
8. return \( R \)
9. end procedure

**Checking path equivalence.** Our final step consists in comparing all basic blocks of the same depth from $\omega_n^T$ and $\omega_n^F$. However, we do not only check for semantic and conditional equivalence, but also for inclusions (cf. Section 3). For these purposes, let us note $S_n$ the equivalence score between two given $\omega_n^T$ and $\omega_n^F$, and let us note $S_m$ their inclusion score. $S_n$ and $S_m$ represent the amount of coupled basic blocks that are equivalent and included respectively. We define by $\max$ and $\min$ functions that return respectively the maximum and minimum value between two numbers. Moreover, we define a total score $S_{tot}$ such that:

\[
S_{tot} = S_n + S_m
\]

We note that if an equivalence is detected between two paths, we increment $S_n$ without studying their inclusion. Thus, $S_{tot}$ equals at most the number of $\omega_n^T$ or $\omega_n^F$.

\[
S_{tot} \leq \min(\#\omega_n^T, \#\omega_n^F).
\]

In order to check the paths equivalence and inclusion, we compare all $B_m$ with $B'_m$ such that $B_m \in \omega_n^T$ and $B'_m \in \omega_n^F$, and $m \in [1, \min(\#\omega_n^T, \#\omega_n^F)]$.

Then, three cases could occur:

- $B_m$ and $B'_m$ are syntactically equivalent; then we increment the score $S_n$.
- $B_m$ and $B'_m$ are semantically and conditionally equivalent; then we increment the score $S_m$.
- $B_m$ is semantically and conditionally included (but not equivalent) to $B'_m$, then we increment the score $S_{inc}$ (likewise if $B'_m$ is included in $B_m$).

Algorithm 4 describes this process. Based on the calculated score, we can verify if a given predicate is a two-way opaque construct:

- if $S_n \geq S_m$ and $S_m = \max(\#\omega_n^T, \#\omega_n^F)$ then we mark the predicate as a **two-way** opaque construct.
- if $S_n < S_m$ and $S_m = \max(\#\omega_n^T, \#\omega_n^F)$ then we mark the predicate as a **probable** two-way opaque construct. This label means that, since there is more inclusions than equivalences, a false positive is likely. Thus, we suggest in case of a probable two-way opaque predicate to verify the result manually.
- if $S_{inc} < \max(\#\omega_n^T, \#\omega_n^F)$ and $S_{inc} > 0$ then we mark the predicate as **normal** and we propose to group equivalent basic blocks to reduce the control-flow graph.

---

\(^4\)A returning block refers to a basic block that exits a function. 

---

**Figure 4:** Example of CryptoWall main function control-flow graph (CFG) reduction.

Figure 4 shows the original and tagged control-flow graphs (a and b), and the reduced control-flow graph (c). The reduction process involves identifying and merging equivalent basic blocks to minimize the size of the control-flow graph.

**4.2 Detecting two-way opaque predicates**

As discussed in Section 2.2, $P^T$ are excluded from known analysis. In this section, we propose a methodology, based on DoSE in order to handle two-way opaque predicates. The aim of this methodology is to detect and remove all $P^T$ without even making any assumption on their type. Since we do not try to solve the predicate but rather check for semantic equivalence between the paths generated from it, this means that the opaque predicate can be of any construct (e.g. MBA-based, arithmetic-based, alias-based, etc.).

**4.2.1 Methodology.** Our methodology to detect two-way opaque predicates is composed of three steps. Before presenting these steps, we present some notations. We denote by $\phi_n$ the $n$-th predicate of a binary $B$, such that $\phi_n \in B, n \in \mathbb{N}$. Let $\phi_n^T$ be the false branch of a given $\phi_n$ and let $\phi_n^F$ be its true branch. We denote by $\omega_n^T$ and $\omega_n^F$ all paths generated from respectively $\phi_n^T$ and $\phi_n^F$ to a common basic-block within their function. Based on these notions, we proceed as follows:

**Path-constraints collection.** The first step consists in identifying all $\phi_n, n \in \mathbb{N}$, within $B$. If an identified $\phi_n$ is a two-way predicate, then all paths $\omega_n^T$ generated from the true branch $\phi_n^T$, are semantically equivalent to all paths $\omega_n^F$, generated from the false branch $\phi_n^F$. We will use this property afterwards.

**Generating paths.** After collecting all paths constraints (i.e. predicates), we want to generate all paths $\omega_n^T$ and $\omega_n^F$ from respectively $\phi_n^T$ and $\phi_n^F$ to their first common basic-block, using a depth-first search algorithm as illustrated in Algorithm 3. Indeed, if $\phi_n$ is a two-way opaque predicate, then $\omega_n^T$ and $\omega_n^F$ must end either on a common block or on a returning block\(^4\). Moreover, since we aim at
Table 3: Evaluation on the generated use cases with Tigress.

<table>
<thead>
<tr>
<th>Case study</th>
<th>$P^T$</th>
<th>$P^F$, EncD</th>
<th>EncA</th>
<th>EncL</th>
<th>Flat</th>
<th>Virt</th>
<th>(#OP, #FP, #FN)</th>
<th>time avg.(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1 (A, B, C, D)</td>
<td>×10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(10,00)</td>
<td>4.54s</td>
</tr>
<tr>
<td>Case 2 (A, B, C, D)</td>
<td>×4</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,00)</td>
<td>2.32s</td>
</tr>
<tr>
<td>Case 3 (A, B, C, D)</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,00)</td>
<td>2.38s</td>
</tr>
<tr>
<td>Case 4 (A, B, C, D)</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,00)</td>
<td>4.04s</td>
</tr>
<tr>
<td>Case 5 (A, B, C, D)</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,00)</td>
<td>3.52s</td>
</tr>
<tr>
<td>Case 6 (A, B, C, D)</td>
<td>×6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6,00)</td>
<td>4.46s</td>
</tr>
<tr>
<td>Case 7 (A, B, C, D)</td>
<td>×6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6,00)</td>
<td>5.16s</td>
</tr>
<tr>
<td>Case 8 (B, C, D)</td>
<td>×8</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(7,1,1)</td>
<td>12.45s</td>
</tr>
<tr>
<td>Case 9 (A)</td>
<td>×8</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(8,00)</td>
<td>13.28s</td>
</tr>
<tr>
<td>Case 10 (A, B, C, D)</td>
<td>×10</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(8,1,2)</td>
<td>29.09s</td>
</tr>
<tr>
<td>Case 11 (B, C)</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(4,00)</td>
<td>4.54s</td>
</tr>
<tr>
<td>Case 11 (A, D)</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(3,0,1)</td>
<td>3.31s</td>
</tr>
<tr>
<td>Case 12 (A, B, C, D)</td>
<td>×8</td>
<td>×8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6,0,2)</td>
<td>9.13s</td>
</tr>
<tr>
<td>Case 13 (B, C)</td>
<td>×10</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6,0,2)</td>
<td>31.21s</td>
</tr>
<tr>
<td>Case 13 (A)</td>
<td>×10</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(9,2,1)</td>
<td>32.28s</td>
</tr>
<tr>
<td>Case 13 (D)</td>
<td>×10</td>
<td>×4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(7,1,3)</td>
<td>31.48s</td>
</tr>
</tbody>
</table>

Algorithm 4 Paths equivalence checking

1. procedure Paths equivalence checking($\omega^T_n$, true path, $\omega^F_n$: false path)
2. $S_{eq}, S_{inc}, S_{max} = 0, 0, 0$
3. for each basic blocks $B, B'$ in $\omega^T_n$, $\omega^F_n$ do
4. if $\sigma_{cond}(B, B') = \sigma_{cond}(B', B) = 1$ then
5. $S_{eq} =$ + +
6. else if $\sigma_{cond}(B, B') = \sigma_{cond}(B', B) = 1$ then
7. if $\sigma_{cond}(B, B') = 1$ or $\sigma_{cond}(B, B') = 1$ then
8. $S_{eq} =$ + +
9. end if
10. else if $B \subseteq B'$ or $B' \subseteq B$ then
11. if $\sigma_{cond}(B, B') = 1$ or $\sigma_{cond}(B, B') = 1$ then
12. $S_{inc} =$ + +
13. end if
14. end if
15. end for
16. $S_{eq} = S_{eq} + S_{inc}$
17. if $S_{eq} \geq S_{inc}$ and $max(\#, \#o^T_n, \#, \#o^F_n) =$ then
18. return two-way
19. else if $S_{eq} < S_{inc}$ and $max(\#o^T_n, \#o^F_n)$ then
20. return probable
21. else if $< max(\#o^T_n, \#o^F_n)$ and $> 0$ then
22. propose Control-flow graph reduction() // see Algorithm 1
23. end if
24. return normal
25. end procedure

4.2.2 Evaluations. For the evaluation, we used the Tigress obfuscator [7] which implements these opaque predicates. We have selected four C code samples (Huffman as sample A, bubble sort as sample B, binary sort as sample C and matrix multiplication as sample D) which are obfuscated using two-way opaque predicates constructs. We combined them with other obfuscation techniques implemented in Tigress, such as control-flow flattening (Flat), encodings of respectively data (EncD), arithmetics (EncA) and literals (EncL) and finally code virtualization (Virt). These combinations allow us to measure the efficiency as well as the limitations of DoSE for two-way opaque predicates detection. Table 3 groups our evaluations of the four code samples listed above, in a way to present results according to the obfuscation techniques that they use. For example, "Case 1" represents the application of ten opaque predicates $P^T$ to our samples A, B, C, and D with the corresponding evaluation; "Case 2" represents the application of four $P^F$ combined with four $P^T$ or $P^F$ in all samples, etc. The column "(#OP, #FP, #FN)" represents a tuple in which "#OP" is the number of detected two-way predicates, "#FP" is the number of false positive results and "#FN" the number of false negatives. As we can see, we are able to detect all two-way opaque predicates with no false positives and no false negatives in the majority of the cases. The reasons for the few false positive and negative results are the block-centric approach of DoSE and the insertion of infeasible paths. We present these limitations in the following paragraph.

We also evaluated our implementation against real world malwares. Table 4 illustrates our results. We analyzed some functions of these samples with their entry-points listed in column "Function EP" in order to ease the detection of any false positive or negative results. Column 4 shows the number of detected two-way predicates, false positive and false negative results as a tuple whereas column 5 shows the execution time. As we can see, two-way opaque predicates are efficiently detected, within an acceptable amount of time. Further, in some cases, such as the Vipasana malware, specific patterns are used (based on an additional subtraction with 0 within their cloned blocks) to construct their two-way opaque predicates. Such information can be used to create more detection rules for these malwares.

4.2.3 Limitations. The performed evaluations underline the problematic of inserting infeasible paths with opaque predicates of types $P^T$ or $P^F$ within a path generated from a two-way opaque predicate. Such combination inserts bogus blocks that will never be reached within equivalent path derived from a $P^F$ opaque predicate.
### Table 4: Evaluation on malwares for two-way opaque predicates detection and removal.

<table>
<thead>
<tr>
<th>Sample</th>
<th>Function EP</th>
<th>(#OP, #FP, #FN)</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vipasana</td>
<td>0x4371a4</td>
<td>(10,0,0)</td>
<td>1.63s</td>
</tr>
<tr>
<td>Vipasana</td>
<td>0x434df0</td>
<td>(10,0,0)</td>
<td>21.76s</td>
</tr>
<tr>
<td>ZeuS</td>
<td>0x437814</td>
<td>(20,0)</td>
<td>196.04s</td>
</tr>
<tr>
<td>GuoGuo</td>
<td>0x441b510</td>
<td>(10,0)</td>
<td>2.04s</td>
</tr>
<tr>
<td>Kryptik</td>
<td>0x40e6d0</td>
<td>(40,0)</td>
<td>22.10s</td>
</tr>
<tr>
<td>Romberik</td>
<td>0x4c2c5d</td>
<td>(40,0)</td>
<td>1.46s</td>
</tr>
<tr>
<td>Ixsh</td>
<td>0x40106d</td>
<td>(10,0)</td>
<td>3.58s</td>
</tr>
</tbody>
</table>

This limitation shows that our approach must be considered as an additional analysis to state-of-the-art opaque predicate tools in order to first check infeasible paths by detecting $P^T$ and $P^F$, and afterwards complete the analysis by detecting $P^C$ predicates.

Another limitation is due to the insertion of branch functions. These functions are cloned but their entry point addresses are different. This causes our conditional equivalence step to generate dissimilar values for each function. Since both functions have different addresses, they will also have distinct symbols, thus causing some false negative results. However, being able to detect these cloned branch functions (i.e. sub-functions) beforehand prevents such limitations. The next paragraph will introduce the extension of DoSE for the purpose of detecting these cloned sub-functions.

#### 4.3 Detecting cloned sub-functions

In the case of opaque predicates or control-flow flattening, another kind of obfuscation transformation may be applied: replacing a basic block by a function to be called. We refer to these functions as *sub-functions* since they represent only one basic block. In such a case, we need to extend our methodology to the detection of these cloned sub-functions.

##### 4.3.1 Methodology

Such analysis is based on the following process: we take as inputs two different sub-functions $F_1$ and $F_2$, and we compare all basic blocks of $F_1$ with all basic blocks of $F_2$, as it is presented in the following definition:

**Definition 10: Sub-functions Semantic Inclusion.** Let $F_1$ and $F_2$ be two sub-functions. We say that $F_1$ is semantically equivalent to $F_2$ (i.e. cloned) if for every basic block of $F_1$ there exists a unique semantically and conditionally equivalent basic block in $F_2$.

Thus, for each $B$ in $F_1$ and $B'$ in $F_2$, we can apply a similar approach as the one illustrated in Algorithm 2 in order to check for their syntactic, semantic, and conditional equivalence. The only difference is that the algorithm takes two lists of basic-blocks, one for each function. All detected clones are added in a dictionary $C$. Afterwards, $C$ is given to a function which verifies whether our definition for the sub-functions semantic inclusion is satisfied and it returns a boolean value accordingly. Thus, it allows us to confirm if $F_1$ and $F_2$ are cloned or semantically different. However, the above comparison needs to be adapted in order to properly compare two functions containing more complex structures.

##### 4.3.2 Evaluations

We evaluated the detection of statically equivalent sub-functions against known malwares as illustrated in Table 5. Column 3 represents the number of functions before our analysis whereas column 4 illustrates the number of detected cloned sub-functions. Column "(#FP, #FN)" shows the number of false positive and false negative results. Our evaluation shows that some malwares use what we defined as *sub-functions*, notably the worm Flame for which we were able to detect 1954 clones with neither false positives nor false negative results. Such a detection is important, specially towards the reduction of control-flow graphs or the detection of two-way opaque predicates which contain jumps or calls to these cloned functions.

#### 4.3.3 Limitations

For now, our approach is limited to small *sub-functions* with no complex structure (e.g. loops). We are looking to extend this application of DoSE to more complex functions while preserving efficiency and an acceptable time of execution.

### 5 PERSPECTIVES

**Opaque predicate deobfuscation framework.** Using our approach to detect two-way opaque predicates constructs combined with existing opaque predicate detection tools can contribute not only to counter the limitations of these tools, but also prevent DoSE limitation due to infeasible branches. Indeed, if prior to detect two-way opaque predicates, we detect and remove $P^T$ and $P^F$ constructs, then we will no longer have our current limitation.

*Hybrid analysis*. Even if our current approach is evaluated statically, it is straightforward to use it dynamically through DSE. Moreover, using our approach dynamically would prevent limitations due to emulation of memory access since their concrete values are available at run-time. However, limitations of dynamic analysis would still be relevant and it will prevent us of contributing to generic de-obfuscation techniques by statically reducing the amount of code to cover. Thus, we are looking forward to a clever combination of static and dynamic analysis in order to keep our goal of contributing to generic deobfuscation techniques statically while improving our accuracy dynamically, thus preserving DoSE scalability to real-world use-cases.

### 6 CONCLUSION

Obfuscated softwares raise many issues during their reverse engineering or evaluation. Most of the deobfuscation techniques come with limitations since they are based on dynamic symbolic execution. We have proposed a novel deobfuscation method based on semantic equivalence, called DoSE. We applied binary diffing methods based on semantic equivalence to deobfuscate binaries in order to provide a methodology to statically detect and remove protections based on code-reuse. We presented this approach by formalizing and improving it for a better correctness and efficiency. Several applications of DoSE were also presented: detect and remove two-way opaque predicates, reduce control-flow graphs by detecting range dividers and code-reuse and detect cloned sub-functions. The benefits of DoSE are also demonstrated with several realistic classes of opaque predicates using Tigress, along with existing malwares. Our evaluations show that DoSE can efficiently reduce control-flow graphs of malwares such as Flame up to 63%, or even detect 1954 sub-functions, with an acceptable amount of time. Moreover, we demonstrated that DoSE can be efficiently extended.
to the detection of two-way opaque predicates, which until then were not detected by any known technique. Therefore, this work paves the way for combining semantic equivalence methodologies with existing generic deobfuscation techniques, in order to improve their efficiency and scalability.

ACKNOWLEDGMENTS

This work is supported by the French National Research Agency in the framework of the Investissements d’Avenir program (ANR-15-IDEX-02).

REFERENCES


Selected Papers (Lecture Notes in Computer Science), Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.), Vol. 7839. Springer, 92–109. https://doi.org/10.1007/978-3-642-37682-5_8


