DoSE: Deobfuscation based on Semantic Equivalence

Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, Thanh-Ha Le

DoSE: Deobfuscation based on Semantic Equivalence

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

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

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

Thanh-ha Le
Trusted Labs
Meudon, France
thanhha.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 diffing 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 diffing techniques based

on semantic equivalence to deobfuscate binaries. Our transposition of existing binary diffing 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 contrary to the current deobfuscation techniques, threats 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 countering 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.

```
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 aqQiUv) {ooGoRv[(
      kIKfgI+eCFI)&262143]=ooGoRv[(kIKfgI+picb)&262143]^
      ooGoRv[(kIKfgI+aqQiUv)&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.cr.yp.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.

```
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 == 809267)
  printf ("win \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^F, P^T and $P^?$ as being opaque predicates that are always evaluated to *false*, *true* or *unknown* (either true or false) respectively.

The latter construction of opaque predicates $P^?$ 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].

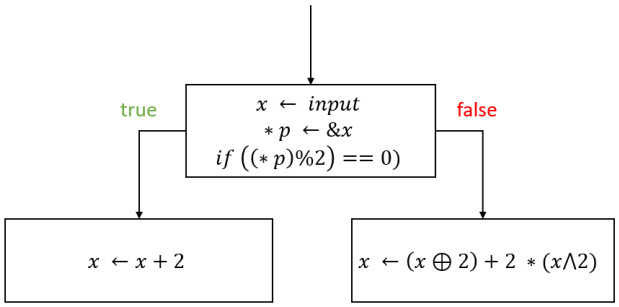

Figure 1: Example of a two-way opaque predicate.

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[2] 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

[2] A basic block is a straight-line code sequence with only one entry point and one exit point.

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 I_n be the n-th instruction of B and 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 $I_n \in B$, there exists a *unique* $I_m \in B'$ such that $I_m =_{\text{syn}} I_n$, with $m = n$, and we set $B \subset_{\text{syn}} B'$. In other words, $I_m =_{\text{syn}} I_n$ 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 $B \subset_{\text{syn}} B'$ and $B' \subset_{\text{syn}} B$ 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 $B \subset_{\text{sem}} B'$. 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 $B \subset_{\text{sem}} B'$ and $B' \subset_{\text{sem}} B$. 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 functionally equivalent portion of code but regardless of the memory area used, or of the function called within the blocks. Thus, in some cases, functionally 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.

```
loc_439B45:
mov     eax, edi
mov     edx, offset dword_439F84
call    sub_404A70
mov     eax, [ebp+var_8]
call    sub_404A68
mov     esi, eax
test    esi, esi
jle     loc_439C3E
```

```
loc_439C3E:
mov     eax, edi
mov     edx, offset dword_439F84
call    sub_404A70
mov     eax, [ebp+var_C]
call    sub_404A68
mov     esi, eax
test    esi, esi
jle     loc_439D37
```

Figure 2: Example of two functionally equivalent basic block using different memory areas, from Vipasana ransomware.

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 $C(y_j) =_{\text{cond}} C(x_i)$ and set $B \subset_{\text{cond}} B'$.

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 $B \subset_{\text{cond}} B'$, 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 $B \subset_{\text{cond}} B'$ and $B' \subset_{\text{cond}} B$ 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 *nop* instructions) or destination addresses for *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.

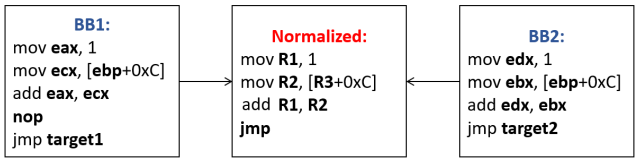

Figure 3: Normalization of the syntax of two blocks which are cloned.

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

Sample	Function	(#FP, #FN) before	time (s) before	(#FP, #FN) after	time (s) after
Asprox	0x10009b82	(5,0)	48.03s	(0,0)	18.14s
Asprox	0x1000be35	(32,2)	1851.09s	(0,0)	243.32s
Flame	0x1003177b	(6,1)	230.84s	(0,0)	26.14s
WannaCry	0x4043b6	(2,0)	124.06s	(0,0)	23.48s
CryptoWall	0x401100	(3,11)	227.57s	(0,3)	67.21s
Vipasana	0x429954	(6,7)	106.95s	(0,5)	24.40s

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.

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 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.* $|X_B| = |Y_{B'}|$). 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

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 ← GetInstructions(B)
6:          NormalizeInstructions(I_B)
7:          IR_B ← GetIntermediateLanguage(I_B)
8:          Simplify(IR_B)
9:          X_B ←SymbolicExecution(IR_B)
10:         NormalizeSemantics(X_B)
11:         L[B] ←< I_B, IR_B, X_B >
12:     end for
13:     C ←Syntactic and semantic equivalence comparisons(L)
14:     // see Algorithm 2.
15:     return C
16: end procedure
```

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

[3]Samples are available at https://github.com/lamaram/DoSE

Sample	Type	Function EP	# Nodes	% Reduction	(#FP, #FN)	time (s)
BitCoinMiner	Trojan	0x40a900	97	52.58%	(0,0)	25.42s
		0x407240	697	47.06%	(0,0)	933.49s
Hupigon	Backdoor	0x49935c	321	58.57%	(0,0)	141.00s
Asprox	Trojan	0x1000be35	436	41.97%	(0,0)	243.32s
		0x10009b82	57	45.61%	(0,0)	18.14s
		0x100096a5	67	20.90%	(0,0)	14.01s
		0x100091ac	33	39.39%	(0,0)	1.38s
Dircrypt	Trojan	0x409c70	113	33.63%	(0,0)	13.14s
		0x4060c0	44	18.18%	(0,0)	3.39s
		0x406da0	30	23.33%	(0,0)	2.57s
Vipasana	Ransomware	0x429954	95	25.26%	(0,5)	24.40s
		0x425b50	80	40.00%	(0,0)	6.46s
		0x424fc8	64	25.00%	(0,0)	7.51s
		0x4278a8	63	23.81%	(0,0)	20.14s
		0x42d578	60	33.30%	(0,0)	6.09s
		0x4399f8	123	63.41%	(0,0)	43.52s
		0x42be04	59	50.85%	(0,0)	6.20s
Cryptowall	Ransomware	0x401100	179	44.13%	(0,3)	67.21s
Flame	Worm	0x100586ea	365	21.64%	(0,0)	58.44s
		0x1003177b	157	29.30%	(0,0)	26.14s
		0x10023fd6	29	31.03%	(0,0)	4.14s
		0x1006e7b9	100	36.00%	(0,0)	15.60s
		0x1004949f	54	37.04%	(0,0)	3.26s
WannaCry	Ransomware	0x4043b6	123	16.26%	(0,0)	23.48s
		0x403cfc	98	35.71%	(0,0)	12.30s
Dexter	Trojan	0x404ad0	86	27.91 %	(0,0)	25.55s
		0x402050	33	18.18%	(0,0)	5.00s
OnionDuke	Trojan	0x10005b60	76	38.16%	(0,0)	11.56s

Table 2: Evaluation of static control-flow graph reduction using DoSE

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** AlreadyComputed(B', B) = $False$ **then**
6: **if** $\sigma_{\text{syn}}(B, B') = \sigma_{\text{syn}}(B', B) = 1$ **then**
7: $C[B] \leftarrow B'$ // add B' as a clone of B
8: $C[B'] \leftarrow B$ // add B as a clone of B'
9: **else if** $\sigma_{\text{sem}}(B, B') = \sigma_{\text{sem}}(B', B) = 1$ **then**
10: **if** $\sigma_{\text{cond}}(B, B') = \sigma_{\text{cond}}(B', B) = 1$ **then**
11: $C[B] \leftarrow B'$
12: $C[B'] \leftarrow B$
13: **else**
14: pass // B' is not a clone of B.
15: **end if**
16: **else**
17: pass // B' is not a clone of B.
18: **end if**
19: **end if**
20: **end for**
21: **end for**
22: **return** C
23: **end procedure**

false negative results and also the execution time of the analysis.

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 proceeded 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

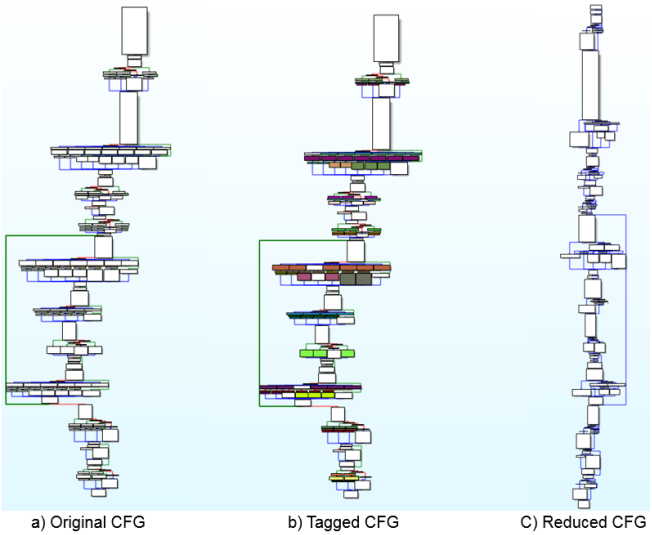

Figure 4: Example of CryptoWall main function control-flow graph (CFG) reduction.

to handle such limitation. However, the cost of such analysis will be greater and could lead to path explosion issues.

4.2 Detecting two-way opaque predicates

As discussed in Section 2.2, $P^?$ 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 $P^?$ 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 ϕ_n the n-th predicate of a binary $\mathcal{B}$, such that $\phi_n \in \mathcal{B}$, $n \in \mathbb{N}$. Let ϕ_n^F be the *false* branch of a given ϕ_n and let ϕ_n^T be its *true* branch. We denote by ω_n^F and ω_n^T all paths generated from respectively ϕ_n^F and ϕ_n^T 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 ϕ_n, $n \in \mathbb{N}$, within $\mathcal{B}$. If an identified ϕ_n is a two-way predicate, then all paths ω_n^T, generated from the true branch ϕ_n^T, are semantically equivalent to all paths ω_n^F, generated from the false branch ϕ_n^F. We will use this property afterwards.

Generating paths. After collecting all paths constraints (*i.e.* predicates), we want to generate all paths ω_n^T and ω_n^F from respectively ϕ_n^T and ϕ_n^F to their first common basic-block, using a depth-first search algorithm as illustrated in Algorithm 3. Indeed, if ϕ_n is a two-way opaque predicate, then ω_n^T and ω_n^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 φ_n in D do
4:         ω_T ← GetTruePaths(φ_n)
5:         ω_F ← GetFalsePaths(φ_n)
6:         R[φ_n] ← PathEquivalenceChecking(ω_T, ω_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 ω_n^T and ω_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 $\mathcal{S}_{\text{eq}}$ the equivalence score between two given ω_n^T and ω_n^F, and let us note $\mathcal{S}_{\text{inc}}$ their inclusion score. $\mathcal{S}_{\text{eq}}$ and $\mathcal{S}_{\text{inc}}$ 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 $\mathcal{S}_{\text{tot}}$ such that:

$$\mathcal{S}_{\text{tot}} = \mathcal{S}_{\text{eq}} + \mathcal{S}_{\text{inc}}$$

We note that if an equivalence is detected between two paths, we increment $\mathcal{S}_{\text{eq}}$ without studying their inclusion. Thus, $\mathcal{S}_{\text{tot}}$ equals at most the number of ω_n^T or ω_n^F.

$$\mathcal{S}_{\text{tot}} \leqslant min(\#\omega_n^T, \#\omega_n^F).$$

In order to check the paths equivalence and inclusion, we compare all B_m with B'_m such that $B_m \in \omega_n^T$, $B'_m \in \omega_n^F$ 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 $\mathcal{S}_{\text{eq}}$.
- B_m and B'_m are semantically and conditionally equivalent; then we increment the score $\mathcal{S}_{\text{eq}}$.
- B_m is semantically and conditionally included (but not equivalent) to B'_m; then we increment the score $\mathcal{S}_{\text{inc}}$ (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 $\mathcal{S}_{\text{eq}} \geqslant \mathcal{S}_{\text{inc}}$ and $\mathcal{S}_{\text{tot}} = max(\#\omega_n^T, \#\omega_n^F)$ then we mark the predicate as a **two-way** opaque construct.
- if $\mathcal{S}_{\text{eq}} < \mathcal{S}_{\text{inc}}$ and $\mathcal{S}_{\text{tot}} = max(\#\omega_n^T, \#\omega_n^F)$ 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 $\mathcal{S}_{\text{tot}} < max(\#\omega_n^T, \#\omega_n^F)$ and $\mathcal{S}_{\text{tot}} > 0$ 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.

Case study	$P^?$	P^T, P^F	EncD	EncA	EncL	Flat	Virt	(#OP, #FP, #FN)	time avg.(s)
Case 1 (A, B, C, D)	×10							(10,0,0)	4.54s
Case 2 (A, B, C, D)	×4	×4						(4,0,0)	2.32s
Case 3 (A, B, C, D)	×4		✓					(4,0,0)	2.38s
Case 4 (A, B, C, D)	×4			✓				(4,0,0)	4.04s
Case 5 (A, B, C, D)	×4				✓			(4,0,0)	3.32s
Case 6 (A, B, C, D)	×6			✓	✓			(6,0,0)	4.46s
Case 7 (A, B, C, D)	×6		✓	✓	✓			(6,0,0)	5.16s
Case 8 (B, C, D)	×8	×4	✓	✓	✓			(7,1,1)	12.45s
Case 8 (A)	×8	×4	✓	✓	✓			(8,0,0)	13.28s
Case 9 (A, B, C, D)	×6					✓		(6,0,0)	7.84s
Case 10 (A, B, C, D)	×10	×4	✓	✓	✓	✓		(8,1,2)	29.09s
Case 11 (B, C)	×4						✓	(4,0,0)	4.54s
Case 11 (A, D)	×4						✓	(3,0,1)	3.31s
Case 12 (A, B, C, D)	×8					✓	✓	(6,0,2)	9.13s
Case 13 (B, C)	×10	×4	✓	✓	✓	✓	✓	(8,0,2)	31.21s
Case 13 (A)	×10	×4	✓	✓	✓	✓	✓	(9,2,1)	32.28s
Case 13 (D)	×10	×4	✓	✓	✓	✓	✓	(7,1,3)	31.48s

Table 3: Evaluation on the generated use cases with Tigress.

Algorithm 4 Paths equivalence checking

1: **procedure** PATHS EQUIVALENCE CHECKING(ω_n^T: true path, ω_n^F: false path)
2: $\mathcal{S}_{\text{eq}}$, $\mathcal{S}_{\text{inc}}$, $\mathcal{S}_{\text{tot}}$ = 0, 0, 0
3: **for** each basic blocks B, B' in ω_n^T, ω_n^F **do**
4: **if** $\sigma_{\text{syn}}(B, B') = \sigma_{\text{syn}}(B', B) = 1$ **then**
5: $\mathcal{S}_{\text{eq}}$ + +
6: **else if** $\sigma_{\text{sem}}(B, B') = \sigma_{\text{sem}}(B', B) = 1$ **then**
7: **if** $\sigma_{\text{cond}}(B, B') = \sigma_{\text{cond}}(B', B) = 1$ **then**
8: $\mathcal{S}_{\text{eq}}$ + +
9: **end if**
10: **else if** $B \subset_{\text{sem}} B'$ or $B' \subset_{\text{sem}} B$ **then**
11: **if** $\sigma_{\text{cond}}(B, B') = 1$ or $\sigma_{\text{cond}}(B', B) = 1$ **then**
12: $\mathcal{S}_{\text{inc}}$ + +
13: **end if**
14: **end if**
15: **end for**
16: $\mathcal{S}_{\text{tot}} = \mathcal{S}_{\text{eq}} + \mathcal{S}_{\text{inc}}$
17: **if** $\mathcal{S}_{\text{eq}} \geqslant \mathcal{S}_{\text{inc}}$ and = $max(\#\omega_n^T, \#\omega_n^F)$ **then**
18: return **two-way**
19: **else if** $\mathcal{S}_{\text{eq}} < \mathcal{S}_{\text{inc}}$ and = $max(\#\omega_n^T, \#\omega_n^F)$ **then**
20: return **probable**
21: **else if** < $max(\#\omega_n^T, \#\omega_n^F)$ 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[5]. 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 $P^?$ to our samples A, B, C, and D with the corresponding evaluation; "Case 2" represents the application of four $P^?$ 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 $P^?$ opaque predicate.

[5]Tigress refers to two-way opaque predicates as *question* opaque predicates (*i.e.* $P^?$).

Sample	Function EP	(#OP, #FP, #FN)	time (s)
Vipasana	0x437fa4	(1,0,0)	1.63s
Vipasana	0x434df0	(10,0,0)	21.76s
ZeuS	0x437814	(2,0,0)	196.04s
GuaGua	0x41b510	(1,0,0)	2.04s
Kryptik	0x40fe00	(4,0,0)	22.10s
Rombertik	0x4c2c3d	(1,0,0)	1.46s
Ixesh	0x40106d	(1,0,0)	3.58s

Table 4: Evaluation on malwares for two-way opaque predicates detection and removal.

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 $P^?$ 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 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 toward 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

Sample	Type	# Functions	# Clones	(#FP, #FN)	time (s)
Flame	Worm	8464	1954	(0,0)	1866.16s
LoadMoney	Trojan	78	3	(0,0)	2.01s
Skylock	Trojan	1212	10	(0,2)	321.93s
Vipasana	Ransomware	1715	45	(0,0)	358.85s
WannaCry	Ransomware	142	2	(0,0)	19.92s
OnionDuke	Trojan	755	67	(0,0)	113.93s
Polip	Trojan	2458	246	(1,0)	648.93s
Dircrypt	Trojan	232	13	(0,0)	39.63s

Table 5: Evaluation of sub-functions detection

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

[1] Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis (Eds.). 2009. *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*. ACM.

[2] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. 2016. DroidClone: Detecting android malware variants by exposing code clones. In *Sixth International Conference on Digital Information and Communication Technology and its Applications, DICTAP 2016, Konya, Turkey, July 21-23, 2016*. IEEE, 79–84. https://doi.org/10.1109/DICTAP.2016.7544005

[3] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, Stephen Schwab, William K. Robertson, and Davide Balzarotti (Eds.). ACM, 189–200. https://doi.org/10.1145/2991079

[4] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 633–651. https://doi.org/10.1109/SP.2017.36

[5] Boldizsar Bencsath. 2012. Duqu, Flame, Gauss: Followers of Stuxnet. https://www.rsaconference.com/writable/presentations/file$_{u}$pload/br-208$_{b}$encsath.pdf. [Online; accessed 30-08-2018].

[6] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 143–157. https://doi.org/10.1109/SP.2008.17

[7] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. [n. d.]. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/index.html. [Online; accessed 30-01-2018].

[8] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (1st ed.). Addison-Wesley Professional.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A Taxonomy of Obfuscating Transformations.

[10] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 184–196. https://doi.org/10.1145/268946.268962

[11] An ECRYPT White-Box Cryptography Competition. 2017. CHES 2017 Capture the Flag Challenge - The WhibOx Contest. https://whibox.cr.yp.to/. [Online; accessed on october 2017].

[12] Kevin Coogan, Gen Lu, and Saumya K. Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). ACM, 275–284. https://doi.org/10.1145/2046707.2046739

[13] Joan Daemen and Vincent Rijmen. 2000. Rijndael for AES. In *AES Candidate Conference*. 343–348.

[14] Fabrice Desclaux. 2012. Miasm : Framework de reverse engineering. https://github.com/cea-sec/miasm. [Online; accessed 09-08-2017].

[15] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting Code Clones in Malware. In *Eighth International Conference on Software Security and Reliability, SERE 2014, San Francisco, California, USA, June 30 - July 2, 2014*. IEEE, 78–87. https://doi.org/10.1109/SERE.2014.21

[16] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, October 20-22, 2008, Proceedings (Lecture Notes in Computer Science)*, Liqun Chen, Mark Dermot Ryan, and Guilin Wang (Eds.), Vol. 5308. Springer, 238–255. https://doi.org/10.1007/978-3-540-88625-9$_{1}$6

[17] Hex-Rays. [n. d.]. IDA Pro : Interactive DisAssembler. https://www.hex-rays.com/products/ida/index.shtml. [Online; accessed 30-01-2018].

[18] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs, See [1], 611–620. https://doi.org/10.1145/1653662.1653736

[19] Jonathan Salwan, Sebastien Bardin and Marie-Laure Potet. 2017. Desobfuscation binaire : Reconstruction de fonctions virtualisees. Symposium sur la securite des technologies de l'information et des communications, SSTIC, 2017.

[20] Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. 2005. Malware phylogeny generation using permutations of code. *Journal in Computer Virology* 1, 1-2 (2005), 13–23. https://doi.org/10.1007/s11416-005-0002-9

[21] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252

[22] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. 2005. Polymorphic Worm Detection Using Structural Information of Executables. In *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers (Lecture Notes in Computer Science)*, Alfonso Valdes and Diego Zamboni (Eds.), Vol. 3858. Springer, 207–226. https://doi.org/10.1007/11663812$_{1}$1

[23] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, Jeffrey Todd McDonald and Mila Dalla Preda (Eds.). ACM, 5:1–5:6. https://doi.org/10.1145/2430553.2430558

[24] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[25] Timea Lazlo and Akos Kiss. 2009. Obfuscating C++ Programs via Control Flow Flattening. https://www.inf.u-szeged.hu/~akiss/pub/pdf/laszlo$_{o}$bfuscating.pdf.

[26] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 389–400. https://doi.org/10.1145/2635868.2635900

[27] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised*

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$_8$

[28] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code, See [38], 757–768. https://doi.org/10.1145/2810103.2813617

[29] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized Semantics-Based Binary Diffing with Application to Malware Lineage Inference. In *ICT Systems Security and Privacy Protection - 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings (IFIP Advances in Information and Communication Technology)*, Hannes Federrath and Dieter Gollmann (Eds.), Vol. 455. Springer, 416–430. https://doi.org/10.1007/978-3-319-18467-8$_2$8

[30] Ginger Myles and Christian S. Collberg. 2005. K-gram based software birthmarks. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA, March 13-17, 2005*, Hisham Haddad, Lorie M. Liebrock, Andrea Omicini, and Roger L. Wainwright (Eds.). ACM, 314–318. https://doi.org/10.1145/1066677.1066753

[31] Alexios Mylonas and Dimitris Gritzalis. 2012. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. *Computers & Security* 31, 6 (2012), 802–803. https://doi.org/10.1016/j.cose.2012.05.004

[32] Beng Heng Ng, Xin Hu, and Atul Prakash. 2010. A Study on Latent Vulnerabilities. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*. IEEE Computer Society, 333–337. https://doi.org/10.1109/SRDS.2010.47

[33] Jeongwook Oh. 2009. Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries. Black Hat USA 2009.

[34] Oreans Technologies. [n. d.]. Code virtualizer: Total obfuscation against reverse engineering. http://www.oreans.com/codevirtualizer.php. [Online; accessed 30-01-2018].

[35] Oreans Technologies. [n. d.]. Themida, Advanced Windows Software Protection System. http://www.oreans.com/themida.php. [Online; accessed 30-01-2018].

[36] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Y. Zhang. 2000. Experience with Software Watermarking. In *16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA*. IEEE Computer Society, 308–316. https://doi.org/10.1109/ACSAC.2000.898885

[37] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2017. Cross-architecture bug search in binary executables. *it - Information Technology* 59, 2 (2017), 83. https://doi.org/10.1515/itit-2016-0040

[38] Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). 2015. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM. http://dl.acm.org/citation.cfm?id=2810103

[39] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, Gregg Rothermel and Laura K. Dillon (Eds.). ACM, 117–128. https://doi.org/10.1145/1572272.1572287

[40] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic Deobfuscation: From Virtualized Code Back to the Original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings (Lecture Notes in Computer Science)*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.), Vol. 10885. Springer, 372–392. https://doi.org/10.1007/978-3-319-93411-2$_1$7

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

[42] Sandro Schulze and Daniel Meyer. 2013. On the robustness of clone detection to code obfuscation. In *Proceeding of the 7th International Workshop on Software Clones, IWSC 2013, San Francisco, CA, USA, May 19, 2013*, Rainer Koschke, Elmar Jürgens, and Juergen Rilling (Eds.). IEEE Computer Society, 62–68. https://doi.org/10.1109/IWSC.2013.6613045

[43] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. https://doi.org/10.1109/SP.2010.26

[44] VMProtect Software. [n. d.]. VMProtect, New-generation software protection. http://vmpsoft.com/products/vmprotect/. [Online; accessed 30-01-2018].

[45] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. 2001. Protection of Software-Based Survivability Mechanisms. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (formerly: FTCS), 1-4 July 2001, Göteborg, Sweden, Proceedings*. IEEE Computer Society, 193–202. https://doi.org/10.1109/DSN.2001.941405

[46] Xinran Wang, Yoon-chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior based software theft detection, See [1], 280–290. https://doi.org/10.1145/1653662.1653696

[47] Zheng Wang, Ken Pierce, and Scott McFarling. 2000. BMAT - A Binary Matching Tool for Stale Profile Propagation. *J. Instruction-Level Parallelism* 2 (2000). http://www.jilp.org/vol2/v2paper2.pdf

[48] Babak Yadegari. 2016. *Automatic Deobfuscation and Reverse Engineering of Obfuscated Code*. Ph.D. Dissertation. University of Arizona, Tucson, USA. http://hdl.handle.net/10150/613135

[49] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code, See [38], 732–744. https://doi.org/10.1145/2810103.2813663

[50] Yonathan Klijnsma. 2015. The Story of CryptoWall: a historical analysis of a large scale cryptographic ransomware threat. https://www.botconf.eu/wp-content/uploads/2015/12/OK-P14-Yonathan-Klijnsma-The-Story-of-CryptoWall-a-historical-analysis-of-a-large-scale-cryptographic-ransomware-threat.pdf. [Online; accessed 30-08-2018].

[51] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers (Lecture Notes in Computer Science)*, Sehun Kim, Moti Yung, and Hyung-Woo Lee (Eds.), Vol. 4867. Springer, 61–75. https://doi.org/10.1007/978-3-540-77535-5$_5$

[52] Zynamics. 2013. BinDiff. https://www.zynamics.com/bindiff.html. [Online; accessed 10-07-2017].