Analysis of obfuscation transformations on binary code

Matthieu Tofighi Shirazi

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de
DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Mathématiques et Informatique**

Arrêté ministérial : 25 mai 2016

Présentée par

Ramtine TOFIGHI SHIRAZI

Thèse dirigée par **Philippe ELBAZ-VINCENT**

préparée au sein du Laboratoire **Institut Fourier**
dans l'École Doctorale **MSTII**

Evaluation des méthodes d'obscurcissement de binaire

Thèse soutenue publiquement le **16 décembre 2019**,
devant le jury composé de :

M, Louis GOUBIN
Professeur à l'Université Versailles-Saint-Quentin-en-Yvelines, Versailles, Rapporteur

M, Aurélien FRANCILLON
Professeur associé à EURECOM, Biot, Rapporteur

M, Sébastien JOSSE
Chercheur à la Direction générale de l'armement, Rennes, Examinateur

M, Igor MUTTIK
Professeur invité à la Royal Holloway Université de Londres, Londres, Examinateur

Mme, Marie-Laure POTET
Professeur à l'Université Grenoble Alpes, Grenoble, Présidente

M, Guénaël RENAULT
Professeur à l'Ecole Polytechnique, Paris, Examinateur

Mme, Irina-Mariuca ASAVOAE
Consultante à Trusted Labs, Thales Group, Meudon, Encadrante de thèse

M, Philippe ELBAZ-VINCENT
Professeur à l'Université Grenoble Alpes, Grenoble, Directeur de thèse

Remerciements

Je souhaite remercier en premier lieu mon directeur de thèse, M. Philippe Elbaz-Vincent, pour m'avoir encadré et soutenu tout au long de cette thèse. Je lui suis également reconnaissant pour le temps conséquent qu'il m'a accordé, ses qualités pédagogiques et scientifiques, sa franchise et sa sympathie. J'ai beaucoup appris à ses côtés et je lui adresse ma gratitude pour tout cela.

J'adresse de chaleureux remerciements à ma co-encadrante de thèse, Irina-Marica Asavoae, pour son attention de tout instant sur mes travaux, pour ses conseils avisés et son écoute qui ont été prépondérants pour la bonne réussite de cette thèse.

Je remercie également mes anciennes co-encadrantes, Maria Christofi et Thanh Ha Le, avec lesquelles l'experience fut courte mais très enrichissante.

Un grand merci à Louis Goubin et à Aurelien Francillon pour avoir accepté de rapporter ma thèse et pour tous leurs conseils, ainsi qu'aux autres membres du jury. Je désire églament remercier Sebastien Josse pour ses remarques, son ouverture d'esprit, sa franchise, et sa gentillesse qui m'ont permis d'atteindre mes objectifs dans le cadre du doctorat.

Bien sûr, atteindre ces objectifs n'aurait pas été possible sans l'aide de nombreux collègues. Je souhaite notamment remercier Aline Gouget, pour son soutien tout au long de cette thèse.

Enfin, je remercie ma famille pour leur soutien au cours de ces trois années, et plus particulièrement ma mère sans laquelle je n'en serais pas là aujourd'hui.

Résumé long

Introduction

Contexte

L'ingénierie inverse, également appelée la rétro-ingénierie, est le processus d'analyse d'un système qui permet d'identifier les composants et leur relation afin de créer des représentations du système à un niveau d'abstraction supérieur ou différent. L'ingénierie inverse est utilisée sur des systèmes logiciels afin de comprendre leur fonctionnement interne. Cette pratique est courante concernant le développement de logiciels, l'étude de programmes malveillants, l'audit de sécurité et également dans le but de mettre en échec les systèmes de protection logicielle. Dans ce dernier cas, lorsque des utilisateurs malveillants ont un accès complet à un logiciel, l'ingénierie inverse est utilisée pour les attaques de type *Man-At-The-End* (MATE).

Afin de se protéger contre l'ingénierie inverse, et plus généralement contre les attaques de type MATE, les techniques de protection des logiciels sont largement utilisées de sorte à préserver la valeur commerciale du logiciel. Les algorithmes de protection logicielle appartiennent à quatre catégories:

- l'obscurcissement du code pour rendre l'ingénierie inverse plus difficile en les rendant inintelligibles ;
- des méthodes d'inviolabilité utilisées pour se protéger contre des modifications illicites d'un logiciel;
- le tatouage numérique insérant des messages de vérification afin d'identifier le propriétaire et ses droits;
- le marquage, dit *de naissance*, utilisé pour extraire les caractéristiques inhérentes au programme d'origine de maière à en détecter l'originalité.

Dans cette thèse, nous nous concentrons principalement sur l'obscurcissement de code en tant que technique de protection des logiciels.

Défis Scientifiques

L'obscurcissement de code est perçu comme une stratégie de gestion de l'information visant à masquer le sens pouvant être tiré d'un logiciel, tout en préservant ses fonctionnalités d'origines. Actuellement, l'obscurcissement est utilisé comme mécanisme de protection de la propriété intellectuelle, mais aussi pour dissimuler les comportements malveillants de certain code binaire. Par conséquent, l'évaluation des méthodes d'obscurcissement est une question ouverte à laquelle il est souvent répondu par des méthodologies de dés-obscurcissement. Le processus de dés-obscurcissement est constitué par des méthodes d'ingénierie inverse qui évaluent la force des protections d'obscurcissement appliquées. Cependant, ces méthodes se concentrent souvent sur des protections spécifiques.

Cette thèse porte sur l'évaluation des transformations d'obscurcissement appliquées aux codes binaires. L'objectif est de fournir différentes études et méthodologies afin d'aider les évaluateurs et les rétro-concepteurs durant l'analyse des logiciels obscurcis.

Contributions

Le processus de dés-obscurcissement peut être vu sous différentes approches, telles que la suppression d'une ou plusieurs transformations, la simplification du programme ou encore la collecte d'informations dites « méta-données » à propos du code obscurci.

Dans cette thèse, nous contribuons à chaque approche de dés-obscurcissement comme décrit dans les paragraphes suivants.

Contribution 1 – DoSE, Dés-obscurcissement basée sur l'équivalence sémantique de code

La première contribution consiste en une approche de simplification de programme. Il s'agit d'une méthodologie de dés-obscurcissement basée sur l'équivalence sémantique de code, appelée `DoSE`. `DoSE` permet principalement de simplifier le code binaire en vérifiant l'équivalence syntaxique et sémantique de portions d'un code binaire. Cette vérification permet de supprimer les nouvelles transformations d'obscurcissement qui entravent les analyses de dés-obscurcissement de pointe, basées sur l'analyse dynamique et symbolique d'un code.

Contribution 2 – L'évaluation des prédicats opaques de manière statique grâce à l'apprentissage automatique et à l'analyse binaire

La deuxième contribution consiste en une approche de suppression de transformation d'obscurcissement. Basée sur une méthodologie d'apprentissage automatique et supervisée, notre approche vise à détecter puis supprimer des schémas d'obscurcissement spécifiques, mais largement utilisés,

nommé prédicat opaque. À notre connaissance, il s'agit de la première méthodologie d'élimination de transformation d'obscurcissement utilisant des techniques d'apprentissage automatique.

Contribution 3 – La détection des transformations d'obscurcissement basées sur un ensemble de modèle d'apprentissage et le raisonnement sémantique

La troisième contribution de cette thèse se base sur une méthode de collecte de méta-données du code protégé. En utilisant des techniques avancées d'apprentissage automatique et de raisonnement sémantique, la méthodologie proposée permet aux analystes d'identifier plusieurs couches de transformations d'obscurcissement appliquées au code binaire, ce qui représente une étape importante précédent la suppression de ces protections.

État de l'art

Obscurcissement du code

L'obscurcissement de code est une stratégie de gestion de l'information visant à obscurcir la signification du code, tout en préservant ses fonctionnalités. L'objectif principal est de protéger la propriété intellectuelle des auteurs de logiciels. Cependant, l'obscurcissement est également très utilisé par les auteurs de logiciels malveillants pour empêcher la détection et l'analyse de leurs codes. Par conséquent, la capacité d'évaluation de telles transformations d'obscurcissement est une étape importante vers une meilleure protection des logiciels. De ce fait, nous introduisons par la suite différentes définitions de l'obscurcissement. Puis nous décrivons brièvement différentes méthodes d'obscurcissement et de dés-obscurcissement, avant de présenter les outils en lien avec ce domaine d'étude.

Obscurcissement en boîte noire virtuelle

Une première définition de l'obscurcissement prend en compte la propriété d'indiscernabilité entre deux programmes. La propriété essentielle, appelée boîte noire virtuelle, nécessite que tout ce qui est efficacement calculable avec le programme obscurci puisse également être calculé avec un accès oracle au programme initial. Cependant, cette définition étant trop restrictive, elle ne permet pas sa mise en pratique.

Obscurcissement indiscernable

Cette définition de l'obscurcissement repose sur deux conditions: le programme obscurci doit obtenir les mêmes résultats que le programme original. Le programme obscurcit ne doit pas être distinguable du programme non obscurcit (les deux programmes ayant la même fonctinnalité).

Toutefois, la définition ne fournit aucune garantie quant à l'obscurcissement de deux circuits dotés de fonctionnalités différentes.

Obscurcissement empirique

Cette définition consiste essentiellement en des propriétés moins strictes, de manière à ce que le programme obscurci doit avoir le même comportement observable. L'obscurcissement empirique évite la complexité transformationnelle des deux définitions précédentes et est largement utilisé par des outils d'obscurcissement à des fins de protection de logiciels.

Les outils d'obscurcissement empiriques, ou pratiques, sont basés sur des compositions de différentes transformations de programmes. Chacune de ces transformations ajoute sa propre complexité et leur combinaison contribue à la résilience globale du programme protégé. Un outil d'obscurcissement peut appliquer ces transformations à différentes représentations d'un programme, telles que le code source, la représentation intermédiaire ou le langage assembleur.

Les transformations par obscurcissement sont évaluées selon quatre critères:

1. La furtivité décrivant la capacité de détection de la transformation par un adversaire;
2. La résistance permettant de mesurer si la transformation appliquée affaiblit d'autres transformations;
3. La résilience indiquant si le processus d'inversion de la transformation nécessite plus de ressources que sa création, soit sa force contre un adversaire;
4. Le coût, révélant les pénalités en termes de temps d'exécution ou de taille du code qu'implique l'application de la transformation.

Chacun de ces critères peut être mesuré à l'aide de métriques de complexité telles que:

- métriques basées sur le flux de contrôle: complexité cyclomatique, niveaux de nidification, nœuds;
- mesures basées sur les flux de données: fan-in / fan-out, flux de données;
- métriques basées sur les instructions: nombre d'opérandes, vocabulaire du programme, volume du programme.

Des études récentes tentent de mesurer les critères d'un programme obscurci (furtivité, puissance et résilience) sur la base d'un modèle d'attaquant. Le modèle d'attaquant repose sur des attaques automatiques basées sur des techniques génériques de dés-obscurcissement fondées

sur des moteurs d'exécution symboliques dynamiques ou concoliques. De plus, des techniques d'apprentissage automatique sont mises en place pour évaluer la furtivité de la transformation par obscurcissement, appelée attaque via la récupération de méta-données (*metadata recovery attacks*). Néanmoins, la capacité à mesurer avec précision les critères de transformations d'obscurcissement reste une question ouverte.

Il existe plusieurs transformations d'obscurcissement, chacune d'elles ayant des objectifs différents. Elles peuvent être classées en différentes catégories, telles que l'obscurcissement des données, l'obscurcissement de code statique et l'obscurcissement de code dynamique.

Les méthodes d'obscurcissement des données modifient la forme dans laquelle les données sont stockées dans un programme, afin de se prémunir contre des analyses directes. En général, l'obscurcissement des données nécessite la modification du code du programme, de sorte que la représentation des données d'origine puisse être reconstruite durant l'exécution. Ces modifications sont effectuées par ré-ordonnancement, codage ou conversion de données statiques en procédures.

Les transformations statiques d'obscurcissement de code sont similaires aux techniques d'optimisation des compilateurs. Elles modifient le code du programme mais la sortie sera exécutée sans aucune modification. La modification du code peut être effectuée par certaines des techniques suivantes : substitution d'instructions, duplication de code, insertion de code mort ou non pertinent, prédicats opaques, réorganisation, transformation en boucle, division ou fusion de fonctions, chevauchement de code, repli de code, aplatissement de flux de contrôle, modifications de noms de fonctions ou de variables, parallélisation de code et suppression des appels de bibliothèque statiques.

L'obscurcissement de code dynamique est caractérisé par le fait que le code exécuté diffère de ce qui est statiquement visible dans le binaire. Les techniques utilisées pour l'obscurcissement dynamique du code sont le chiffrement ou la compression du code, la virtualisation, l'insertion de mesures empêchant le débogage mais encore l'exécution de code dépendant du matériel.

Dés-obscurcissement

Le terme de dés-obscurcissement regroupe toutes les techniques visant à évaluer les protections de logiciels. À savoir, le processus de dés-obscurcissement peut être vu comme:

1. La suppression des transformations d'obscurcissement appliquées;
2. La simplification du code obscurci;

3. La collecte d'informations sur le code protégé.

Il existe plusieurs approches et méthodologies de dés-obscurcissement des programmes: statique, dynamique et symbolique.

Les techniques de dé-obscurcissement statiques utilisent l'analyse statique visant à déduire des informations sur un programme en raisonnant sur toutes les exécutions possibles de celui-ci. Plusieurs analyses de flux de données statiques ont été proposées pour traiter les transformations d'obscurcissement, par exemple l'analyse de dépendance des données ou d'alias. Cependant, elles sont sujettes à des limitations, car l'analyse statique peut perdre en précision face à des techniques d'obscurcissement telles que l'aplatissement du flux de contrôle.

L'analyse dynamique est une partie importante de l'étude des programmes malveillants. Elle permet l'étude des exécutions réelles d'un programme, que ce soit en direct (pendant l'exécution) ou en différé (en utilisant des traces d'exécutions enregistrées). Cependant, une analyse dynamique peut manquer certains chemins d'exécution lorsque le nombre de voies possibles dans un programme est trop important pour être testé de manière exhaustive. Cette limitation, également appelée « problème de couverture de code », est en général inévitable en raison de son caractère indécidable. À cette fin, alors que l'analyse statique opte pour la précision par rapport au coût, l'analyse dynamique privilégiera la couverture plutôt que le coût.

Les méthodologies de dés-obscurcissement les plus courantes et les plus utilisées sont basées sur des techniques d'exécution symboliques. Alors que les autres approches existantes sont essentiellement divisées en méthodes statiques et dynamiques, les approches symboliques offrent un équilibre précieux entre les deux. Les techniques d'exécution symboliques statiques capturent la sémantique d'un programme. Un interpréteur est utilisé afin de suivre le programme, tout en supposant des valeurs symboliques pour les entrées plutôt que d'obtenir des valeurs concrètes comme le ferait une exécution normale. L'exécution symbolique dynamique, également appelée exécution concolique, est largement utilisée pour le dés-obscurcissement. L'exécution symbolique nécessite les avantages d'un chemin d'exécution concret.

Les méthodologies de dés-obscurcissement sont créées pour des transformations d'obscurcissement spécifiques et utilisent les deux approches suivantes :

- L'utilisation de techniques d'analyse avancées contre les transformations générales pour simplifier un programme ou récupérer des éléments clés;
- L'utilisation de techniques d'analyse spécifiques pour cibler une transformation d'obscur-

cissement précise.

Les méthodologies de dés-obscurcissement se concentrent généralement sur l'évaluation d'aspects clés du code obscurci tels que les prédicats opaques, l'aplatissement du flux de contrôle ou la virtualisation. Cependant, il existe également des méthodologies génériques de dés-obscurcissement qui commencent par générer des traces d'exécution. Puis, elles reconstruisent le graphe de flux de contrôle parcouru au moyen d'une exécution symbolique dynamique.

Contribution 1 – DoSE, Dés-obscurcissement basée sur l'équivalence sémantique de code

En vue de vaincre les techniques d'obscurcissement récentes, les méthodologies de dés-obscurcissement dites génériques sont basées sur l'exécution symbolique dynamique. Ces approches nécessitent souvent une trace d'exécution, ce qui requiert la génération d'entrées pour un programme, ce qui est coûteux en temps. De ce fait la couverture de code et l'applicabilité sont deux des leurs principaux problèmes. En outre, dans le contexte de l'analyse des programmes malveillants, l'exécution symbolique dynamique est confrontée à des composants et conditions basées sur des événements de réseau (par exemple, la connexion à un serveur de commande et de contrôle), ce qui permet de rendre le dés-obscurcissement plus difficilement applicable. En outre, les nouvelles techniques d'obscurcissement exploitent ces limitations pour entraver davantage les analyses. Leur objectif est de diviser le nombre de chemins et de forcer les moteurs d'exécution symboliques dynamiques à ralentir lors de la tentative de couverture complète du code.

Dans la course pour contrer et supprimer les techniques d'obscurcissement les plus avancées, il est nécessaire de réduire la quantité de code à couvrir. De ce fait, nous proposons notre première contribution, consistant en une nouvelle approche de dés-obscurcissement basée sur l'équivalence sémantique, appelée `DoSE`. Avec `DoSE`, nous visons à améliorer et à compléter les techniques de dés-obscurcissement dynamiques basées sur l'exécution symbolique dynamique en éliminant statiquement les transformations d'obscurcissement construites à partir de la duplication de code. Notre transposition des techniques existantes de différenciation de binaire nous permet de fournir une méthodologie concrète pour détecter et supprimer de manière statique les protections basées sur la duplication de code. Certaines de ces protections ne sont pas traitées par les méthodologies actuelles de dés-obscurcissement, alors que d'autres visent à prévenir les approches génériques. Notre contribution, contrairement aux techniques actuelles, vise également de nouvelles techniques d'obscurcissement basées sur la réutilisation de code et détecte les constructions à prédicats opaques bidirectionnels pour lesquelles il n'existe pas de méthodologie de dés-obscurcissement.

Nous avons implémenté DoSE et l'avons appliqué à différentes familles de logiciels malveillants récents pour montrer comment elle réduit considérablement la quantité de code à couvrir. Afin d'évaluer DoSE, nous avons utilisé des logiciels malveillants connus tels que Cryptowall, WannaCry, Flame et BitCoinMiner, ainsi que des exemples de code obscurci. Nos résultats expérimentaux montrent que DoSE est une stratégie efficace de détection et suppression des transformations d'obscurcissement basées sur la réutilisation de code. Nous obtenons de faibles taux de faux positifs et faux négatifs dans nos résultats, et parvenons jusqu'à 63% de réduction du code sur certains types de programmes malveillants. Nous discutons également de la façon dont il peut être utilisé pour combiner et compléter les techniques de dés-obscurcissement génériques existantes.

Contribution 2 – L'évaluation des prédicats opaques de manière statique grâce à l'apprentissage automatique et à l'analyse binaire

Les techniques et les outils actuels de dés-obscurcissement des prédicats opaques présentent quelques limitations. Les techniques qui évaluent les prédicats opaques se concentrent sur des constructions spécifiques et manquent donc de généricité pour tous les schémas d'invariant existants. De plus, les techniques de dés-obscurcissement les plus récentes sont basées sur une exécution symbolique dynamique qui nécessite la génération de traces d'instructions. Par conséquent, la possibilité de couvrir tous les chemins du programme est un problème qui empêche, dans certains cas, le dés-obscurcissement complet du code. Enfin, les solveurs SMT utilisés dans les analyses d'atteignabilité de chemin souffrent de plusieurs limitations en fonction de la construction des prédicats opaques. Certaines constructions basées sur des pointeurs ou des expressions mixtes booléennes et arithmétiques empêchent généralement les solveurs SMT de prédire la faisabilité d'un chemin.

De ce fait, notre deuxième contribution consiste en une nouvelle approche qui relie les techniques d'analyse de binaire à la classification par apprentissage supervisé et automatique. Notre objectif est de fournir une technique d'évaluation statique et générique pour les prédicats opaques, quelle que soit leur construction. Nous utilisons notre méthodologie comme un outil de dés-obscurcissement automatisé et statique afin d'éliminer les prédicats opaques introduits par les transformations d'obscurcissement. Notre travail a pour objectif de réintroduire l'analyse statique pour l'évaluation et la dés-obscurcissement de logiciels obscurcis.

Nous présentons donc plusieurs études en vue de la construction de modèles d'apprentissage

automatique capables de détecter un prédicat opaque ou de prédire sa valeur invariante sans exécuter le code. Nous étendons également notre conception au dés-obscurcissement, quelle que soit leur construction, en créant un outil d'analyse statique. Pour évaluer plus en profondeur notre méthodologie, nous la comparons aux outils disponibles basés sur l'exécution symbolique statique et dynamique pour le dés-obscurcissement des prédicats opaques. Nous menons d'autres évaluations contre des obscurcissements tels que `Tigress` et `OLLVM`. Selon nos résultats expérimentaux, nos modèles ont une précision pouvant atteindre 98% pour la détection et la dés-obscurcissement de prédicats opaques. En revanche, les méthodes de dés-obscurcissement basées sur une exécution symbolique montrent moins de précision, principalement en raison des contraintes liées aux solveurs SMT.

Les conséquences de cette contribution montrent que la combinaison de techniques d'apprentissage automatique et d'analyse symbolique statique fournit une méthodologie générique, automatique et précise pour l'évaluation des prédicats opaques. Notre travail montre que l'apprentissage automatique permet une efficacité et une généricité meilleures pour cette application, tout en nous permettant de faire abstraction des solveurs SMT.

Contribution 3 – Détection des transformations d'obscurcissement basées sur un ensemble de modèles d'apprentissage et le raisonnement sémantique

Parmi la diversité des techniques et méthodes d'obscurcissement, la capacité de détecter efficacement les protections logicielles utilisées est primordiale. De ce fait, les travaux récents de Salem et Banescu se concentrent sur la détection des transformations d'obscurcissement. Leur objectif est de faciliter la sélection et l'application de techniques de dés-obscurcissement adéquates. À notre connaissance, il sont les premiers à étudier la détection des méthodes d'obscurcissement par l'apprentissage automatique. Cependant, leur méthodologie est également sujette à certaines limitations. Tout d'abord, l'apprentissage automatique et le raisonnement syntaxique utilisés pour la détection des transformations peuvent conduire à une forte dépendance entre la fonctionnalité du code étudié et leur modèle, diminuant ainsi la précision des résultats. De plus, la méthodologie utilisée repose sur des problèmes de classification comportant plusieurs classes. Par conséquent, ils considèrent qu'un binaire ne peut être obscurci avec plus d'une transformation d'obscurcissement. Cependant, les transformations peuvent être combinées, d'où la nécessité de pouvoir détecter plusieurs couches de transformation.

C'est pourquoi la troisième contribution de cette thèse consiste en une nouvelle approche combinant des techniques de raisonnement sémantique et un ensemble de modèle de classification dans le but de fournir un cadre de détection statique des transformations par obscurcissement. La capacité à détecter efficacement les protections de logiciel utilisées est primordiale pour faciliter la sélection et l'application de techniques de dés-obscurcissement adéquates. Ainsi, nous fournissons plusieurs études sur les meilleures pratiques d'utilisation des techniques d'apprentissage supervisée pour un modèle évolutif et efficace. De plus, nous étendons notre travail à la détection de constructions de transformations d'obscurcissement, fournissant ainsi une fine méthodologie. Nous pensons que le raisonnement sémantique empêchera notre modèle de dépendre de la fonctionnalité des codes étudiés. De plus, nous appliquons un modèle d'ensemble multi-étiquettes et multi-sorties qui nous permet de détecter plusieurs couches de transformations combinées.

Selon nos résultats expérimentaux et nos évaluations sur des outils d'obscurcissement tels que `Tigress` et `OLLVM`, nos modèles ont une précision pouvant atteindre 91% sur les transformations d'obscurcissement. La précision de notre modèle pour la détection leurs constructions va jusqu'à 100% de précision. Contrairement aux travaux existants, nous proposons des solutions qui exploitent le raisonnement sémantique, par opposition au code désassemblé.

Notre approche souligne l'efficacité du raisonnement sémantique combiné à des techniques avancées d'apprentissage automatique, telles que des ensembles de modèles ainsi que l'approche multi-étiquettes avec multi-sorties.

Conclusions

Dans cette thèse, nous avons étudié différentes approches de dés-obscurcissement en vue d'une évaluation statique des transformations d'obscurcissement. Nous nous sommes principalement concentrés sur le raisonnement sémantique statique, en le combinant avec des techniques bien connues issues d'autres domaines de recherche, telles que la différenciation binaire et l'apprentissage automatique. Nous avons également étudié et développé plusieurs quadriciels de dés-obscurcissement, un pour chacune des approches suivantes: simplifier le code obscurci, supprimer les transformations d'obscurcissement ou collecter des informations sur les protections appliquées. Nos méthodologies et nos outils ont été évalués sur des logiciels malveillants bien connus et des outils d'obscurcissement mettant en œuvre des transformations d'obscurcissement complexes ainsi que largement utilisées.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Reverse engineering is the mechanism of extracting the knowledge or design blue-prints from anything man-made. Many reasons exists to perform reverse engineering, which takes it origins in the analysis of hardware for commercial or military advantages [40]. Reverse engineering is usually performed to obtain missing ideas, knowledge or designs. In some cases, these informations are owned and not shared. In other cases, they can be simply lost or destroyed. In 1990, the Institute of Electrical and Electronics Engineers (*IEEE*) provided a definition for reverse engineering as :

"the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of system in another form or at a higher level of abstraction". (IEEE, 1990).

Such a definition can also be applied to software, which are nowadays omnipresent technologies in our everyday lives. The usage of software is vast, from critical applications to entertainment ones, and software reverse engineering can help improving, maintaining and even securing them.

1.1 Software reverse engineering

The application of software reverse engineering can be seen as two main categories, namely software development and security.

On one hand, software developers can employ reverse engineering techniques to study interoperability with undocumented software. It can also be used to evaluate third-party code, or to improve internal technologies by extracting valuable informations on competitors products.

On the other hand, reverse engineering is employed also for software evaluation and security auditing. The discovery of vulnerabilities, bugs or bad implementations can be reported for developers to fix their products. Reverse engineering is also widely used against cryptographic algorithms. In this case, understanding the internal design of the algorithm can help discover some secrets, or security issues, putting at harm the end users once the software is deployed. Many famous case of reverse engineering can be found throughout the history. In 1987, `Bell Laboratories` reverse-engineered the `Mac OS` System 4.1 so they could run it on RISC machines of their own. More recently in 2011, the software `Skype` was reverse-engineered for the purpose of creating an open-source tool similar to it. This led to the revealing of the inner workings of `Skype`, endangering the security of more than 600 million people's communications.

Malware developers often use reversing techniques to locate weaknesses in operating systems and other software. These vulnerabilities can be used to penetrate a system defense layers and allow its infection. Moreover, in order to detect, analyze, and prevent malicious software, reverse engineering is also required. By tracing every step taken by a malware, the expected rate of infection, and how it can be removed from a system, security analysts and anti-virus developers employ reverse engineering techniques.

Finally, reverse engineering is very popular to defeat software protection schemes, also referred to as *cracking*. Nowadays, media content providers control the distribution of digital media content throughout developed or acquired technologies. These are also referred to as *Digital Right Management* (DRM) technologies. Their goal is to prevent cracking by controlling the use, modification, and distribution of copyrighted works and is of common use by the entertainment industry (*e.g.* `Apple's iTunes Store`). Despite these technological progressions, reverse engineering is often employed when malicious users have complete access over a software or hardware. Such approach allows them to examine, modify, probe, or tamper at will [37, 64]. It is considered as the problem of the *Man-At-The-End* (MATE) attacks [2], as discussed in the next paragraph.

1.2 Man-at-the-end attacks

MATE attacks exists under several forms [2]. In each case, the adversary has physical and authorized access to its target. These attacks are considered an open problem since they are difficult to resolve. Existing counter-measures do not aim at preventing an attack, but rather at slowing it down [46]. Subsequently, no software is considered likely to stay secure for a long period of time.

Therefore, the Digital Asset Protection Association (DAPA) was launched in July 2011 to address the particular challenges of MATE attacks. By addressing the general public, software developers, politicians, and even government agencies, they created some awareness regarding the seriousness of MATE attacks.

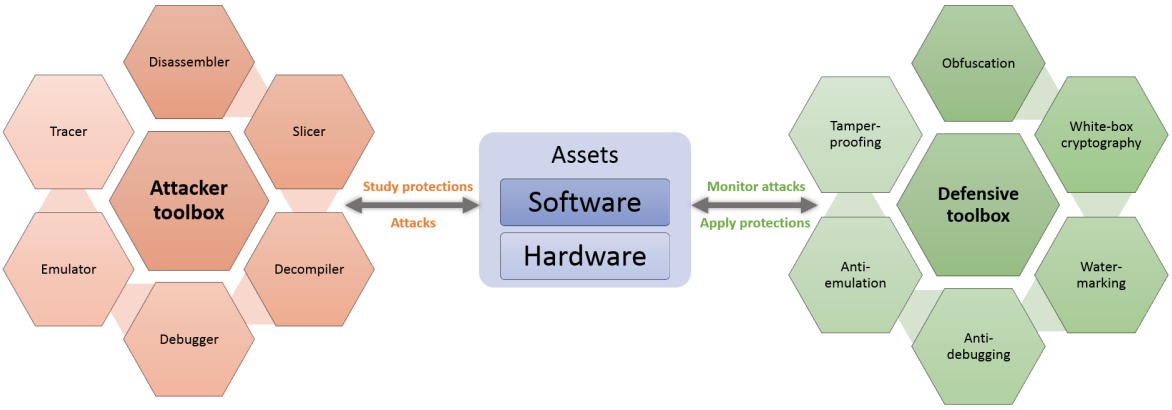

Figure 1.1: MATE attacks scheme

As presented in Figure 1.1, MATE attacks schemes usually involve both the attacker and the defender of the assets. Assets are generally considered as secrets within a software or a device, such as cryptographic keys, or proprietary designs. The attacker, who has complete access to the software or hardware, can study the assets protections and try to attack the software. However, monitoring techniques and tools, as well as updating services, allows the defender to protect the assets.

Different kinds of MATE attack scenarios exist in the literature, *e.g.* [64, 92, 122]. Thus, software protections techniques are widely used to protect against reverse engineering, and more generally against MATE attacks, as discussed in the followings.

1.3 Software protection

The term software protection is typically used to refer to protection of software against piracy, overuse, and reverse engineering. The purpose of software protection is to safeguard the commercial value of the software, regardless of whether any intellectual property contained within it has been compromised. This requires a combination of techniques related to anti-piracy, licensing, and anti-reverse engineering. Often combined with Intellectual Property (IP) protection, software protections can be achieved in several different ways. According to [64], software protection algorithms fall into four basic categories:

1. *Code obfuscation* which makes a program harder to reverse-engineer;
2. *Tamper-proofing* applied to render a program harder to modify;
3. *Watermarking* that allows program to be tracked;
4. *Birth-marking* used for the detection of code lifting.

Many well-known companies are showing interest in software protection schemes [44]. Such information can be deduced by the number of owned or applied patents. `Microsoft` for example owns several software watermarking, obfuscation, and birth-marking patents, *e.g.* [56, 67, 68, 126, 205]. `Apple` also holds patents in code obfuscation, *e.g.* [102]. Other companies such as `Cloakware`, sold to `Irdeto`, hold patents on *white-box cryptography* and also propose their own solutions for software protections, such as [178, 179].

In this thesis, we mainly focus on code obfuscation as a software protection technique. Introduced by Frederick B. Cohen [42], code obfuscation was first considered as a technique to automatically create multiple versions of the same program. The goal was to enhance the difficulty for malware to analyze and modify each generated versions. Ironically, malicious software nowadays employ code diversity to avoid detection. Currently, code obfuscation [45] is perceived as an information management strategy that aims at obscuring the meaning that can be drawn from a software or a code, while preserving its functionality. Many commercial programs use obfuscation as a protective layer to protect themselves against the duplication of their codes, or to hide details of their implementation. As examples, `Skype` or `Dropbox` combine code obfuscation and cryptography to fortify their communication protocols [147]. Other uses of obfuscation can be found on malwares. Their goal is to avoid engine analysis, anti-virus detection, as well as reverse engineering. Overall, code obfuscation is now a widely and commonly used software protection scheme.

The next paragraph will introduce this thesis context, related to code obfuscation as a software protection strategy.

1.4 Context

The evaluation and certification of software sometimes requires reviews from analysts within a short amount of time. Thus, the easy access to evaluation frameworks, and efficient analyses are at a prime. The goal is to find poor designs, vulnerabilities and other issues in the code that may facilitate attacks and cause harm to the end-users.

In the context of code obfuscation, the transformations evaluation is an open question which is often answered by proposing deobfuscation methodologies. The deobfuscation process evaluates the strength of the applied protections, often focusing on specific ones. Other approaches, based on code metrics, can be used to measure the stealth, cost, and quality of the applied protections.

This thesis focuses on the evaluation of obfuscation transformation applied on binary codes. The goal is to provide efficient and scalable methodologies or frameworks to support evaluators, and reverse-engineers, for the analysis of obfuscated programs. In the next section, we briefly introduce our contributions before presenting an overview of the thesis.

1.5 Contributions

The evaluation of obfuscated binaries can be seen as two main approach: either the ability to deobfuscate, or to measure some characteristics of the applied transformations. Furthermore, the deobfuscation process can be seen as different approaches, such as removing transformations, simplifying the program, or gathering metadata information about the obfuscated code. This thesis aims at contributing to each deobfuscation approaches, namely:

- *Program simplification approach*: our first contribution consists in a deobfuscation methodology based on semantic equivalence called `DoSE`. `DoSE` principally allows the simplification of binary code, with extension to some protections removal. These additions allow us to remove novel obfuscation transformations that cause harm to state-of-the-art deobfuscation analyses. Our methodology is also implemented as tool, and evaluated against well known malwares such as `Cryptowall` [206] or `Flame` [24].

- *Transformation removal approach*: our second contribution consists in a machine learning based methodology to remove specific, yet widely used, obfuscation schemes. This is, to the best our knowledge, the first transformation removal approach that uses machine learning

techniques. We provide several studies toward an efficient design of our methodology and illustrate how it can overcome the inherent limitations of these obfuscation schemes. The implementation of our design is also evaluated against state-of-the-art obfuscators and compare to existing tools that aim at removing the targeted obfuscation transformation.

- *Metadata gathering approach*: our third contribution is based on advanced machine learning techniques and semantic reasoning. It allows analysts to identify the applied obfuscation transformations on a binary code. The identification process is an important step in order to select the appropriate methodology for the protections removal. Our contribution underlines the efficiency of advanced machine learning techniques combined with semantic reasoning for that goal. The implementation of our design is also evaluated against state-of-the-art obfuscators.

Beforehand, this thesis provides an overview of state-of-the-art code obfuscation definitions and empirical transformations. We also present existing deobfuscation techniques, either generic or specific to some protection schemes. Finally, we describe some state-of-the-art tools related to the evaluation of code obfuscation.

This thesis is organized as follows: In Chapter 2 we introduce the state-of-the-art on code obfuscation and deobfuscation. We also present an overview of existing tools related to these subjects.

Chapter 3 is dedicated to our first contribution called `DoSE`. `DoSE` is a static **D**eobfuscation methodology based **O**n **S**emantic **E**quivalence, for the purpose of contributing existing deobfuscation techniques based on dynamic symbolic execution. Our implementation of `DoSE` as an `IDA Pro` plug-in is presented.

Chapter 4 presents this thesis second contribution. We describe a first methodology for static code deobfuscation based on machine learning techniques, implemented and evaluated against state-of-the-art obfuscators.

Chapter 5 focuses on our final contribution, also based on machine learning techniques. We propose a static and automated framework for the detection of obfuscation transformations in order to detect multiple obfuscation layers, and fine-tuned to also scale on their constructions.

We conclude this thesis with our perspectives and possible future work in Chapter 6.

Chapter 2

State-of-the-art on code obfuscation and deobfuscation

Contents

In this chapter we introduce code obfuscation, from the virtual black-box definition to the empirical description. Several obfuscation transformations are also presented and illustrated, either static, dynamic, or data-driven for software protection. Afterwards, state-of-the-art evaluations techniques, either specific to some transformations, or generic, are described. Finally, we close this chapter by introducing related tools used to apply or to evaluate code obfuscation transformations.

2.1 Code obfuscation

Code obfuscation [44] 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 goal is to protect software against reverse-engineering, obfuscation is also widely used by malware applications to prevent their detection and analysis. Thus, being able to evaluate such obfuscation transformations or to deobfuscate them is an important step towards a better protection of our intellectual properties and privacy.

In this section we introduce the different definitions of obfuscation namely, virtual black-box obfuscation, indistinguishability obfuscation, and empirical obfuscation.

2.1.1 Virtual black-box obfuscation

A first obfuscation definition that takes into account the *indistinguishability* property between two programs is presented in[80]. Afterwards, Barak *et al.* gave a less restrictive definition using the *virtual black-box* principle, as defined next.

Definition 2.1.1. *Let us denote by P a program. Barak et al. [15] defined an obfuscator O as a probabilistic compiler transformation of P in $O(P)$, which satisfies the following properties:*

- *Functionality: $O(P)$ computes the same function as P;*
- *Polynomial slow-down: for all P, $O(P)$ execution time is at most polynomially slower than P execution time, or polynomially bigger than P size;*
- *Virtual black-box: everything efficiently computable with $O(P)$ can also be computable with only an oracle access to P.*

With such definition, and by building a set of functions that cannot be obfuscated, it has been demonstrated that rendering a program unintelligible, as defined previously, is not possible [15, 16].

However, this does not mean that there are no solutions to make a program unintelligible with a less absolute definition given to the *obfuscation* term.

2.1.2 Indistinguishability Obfuscation

Another approach called *indistinguishability obfuscation* exists to define obfuscation, for which we know that it is achievable for all circuits [76]. The definition of the indistinguishability obfuscation is as follows:

Definition 2.1.2. *A uniform probabilistic polynomial time (i.e. PPT) machine $i\mathcal{O}$ is called an indistinguishability obfuscator for a circuit class $\mathcal{C}_\lambda$ if:*

- *For all security parameters $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$ and all inputs x we have that:*

$$Pr[C'(x) = C(x) | C' \leftarrow i\mathcal{O}(\lambda, C)] = 1.$$

  This is called the correctness condition.

- *For any PPT distinguisher D, there exists a negligible function ε such that, for all security parameters $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in C_\lambda$, we have that if $C_0 = C_1(x)$ for all inputs x then :*

$$|Pr[D(i\mathcal{O}(\lambda, C_0)) = 1] - Pr[D(i\mathcal{O}(\lambda, C_1)) = 1]| \leqslant \varepsilon(\lambda).$$

  This is called the security condition.

In other words, we first have the correctness condition which establishes that, for all inputs, the obfuscated version of the circuit computes the same results as the original circuit. This is similar to the *Functionality* property of the virtual black-box obfuscation. The second condition regarding the security definition of $i\mathcal{O}$ indicates that, for any pair of circuits which compute the same functionality, the respective obfuscated circuits should be indistinguishable.

However, the indistinguishability obfuscation only guarantees that two programs with the same functionality are indistinguishable when obfuscated. The definition does not provide any guarantees about obfuscating two circuits with different functionalities. Nevertheless, Barak *et al.* [16] state that, if it is possible to distinguish these circuits, then it must also be possible to find inputs on which they differ. Moreover, they also argue that $i\mathcal{O}$ is as good as any other obfuscator in such cases. Yet, the notion is considered tricky to be used in practice [8]. This underlines the necessity of empirical obfuscation techniques, as introduced in the next section.

2.1.3 Empirical obfuscation

In [45], Collberg, Thomborson and Low propose a definition of code obfuscation that avoids the paradigm of the virtual black-box definition.

Definition 2.1.3. *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 obfuscation transformation if P and P' have the same observable behavior. Consequently, the following conditions must be fulfilled for an obfuscation 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.*

Empirical or practical obfuscators are based on the composition of different program transformations. Each of these obfuscation transformations adds its own complexity and combining them contributes to the global resilience of the protected program. An obfuscator can apply its techniques on different representations of a program, such as:

- *Source code*: obfuscating a source code enables the use of obfuscation techniques that exploit some specificities of the input programming language. The source code obfuscation is also used for interpreted languages where the source code can be decompiled easily. Moreover, source-to-source obfuscators are easier to integrate into an existing compilation chain. They also allow a multi-architecture use of the protected program. Yet, source-to-source obfuscators implicitly use intermediate languages, such as `CIL` [136] for `Tigress`. The major issue of source-to-source obfuscation is the restriction on the input programming language.

- *Intermediate representation*: intermediate representations (IR) are widely used by compilers or reverse engineering frameworks to represent a code. Obfuscators based on an intermediate representation are more fitted to work on input program with different programming languages and can also target multiple assembly languages. However, the integration of an IR-based obfuscator into a compilation tool-chain can be trickier than for source-to-source obfuscators. Yet, several obfuscators are based on `LLVM` [114] or `Obfuscator-LLVM` [96] to facilitate their integration, *e.g.* [200].

- *Assembly language*: applying obfuscation transformations on the assembly code is more difficult than other approaches. It can lead to a loss of information compared to IR-based obfuscators. Yet, several obfuscation transformation such as code virtualization or anti-debugging (*e.g.* packing, *c.f.* 2.1.4) can be applied directly on the assembly language. Other approaches are more hybrid [156], providing patching mechanisms of the assembly code and source-to-source transformations.

Next, we describe some properties and metrics used to evaluate obfuscation transformations, as introduced in the work of Collberg, Thomborson and Low [45].

2.1.3.1 Metric-based obfuscation measurements

Obfuscation transformations are evaluated according to four criteria as described in [45]:

1. *Stealth* to describe the ability to detect the obfuscation transformation by an adversary;

2. *Potency* to report if the applied obfuscation transformation weaken other transforms;

3. *Resiliency* to express if the reversing process of the obfuscation transformation requires more resources than creating it, *i.e.* its strength against an adversary;

4. *Cost* to capture the overhead in space or time implied by the application of the obfuscation transformation.

Each of these criteria can be measured using complexity metrics, as presented by Collberg, Thomborson and Low, such as:

- *control-flow based metrics*: cyclomatic complexity [125], nesting levels [82], knots [197],
- *data-flow based metrics*: fan-in/fan-out [120], data-flow slicing [143],
- *instructions-based metrics*: number of operands, program vocabulary, program volume [81],

The previous list is not exhaustive, other program metrics exists (*e.g.* alias-based, data-based), for which a more detailed presentation can be found in [52]. The following paragraphs will provide detailed definitions of these criteria.

Stealth. While a resilient transformation may not be sensitive to attacks from automated deobfuscators, it may still be prone to attacks by humans. Particularly, if a transformation introduces new codes that differ from the original program by a large margin, it would be easy for a reverse engineer to detect it. In other words, it is essential that obfuscated code keeps enough similarities with the original code. If that is the case, the obfuscation transformation is considered *stealthy*.

Potency. Informally, an obfuscation transformation is potent if it succeed in confusing an adversary by hiding the intent of the original code. Formally, let T be a semantic-preserving obfuscation transformation such that $P \xrightarrow{T} P'$ transforms a program P into its obfuscated version P'. Let $E(P)$ be the complexity of P, as defined by several metrics (*e.g.* Halstead [81], McCabe [125], Harrison [82], etc.). Based on these metrics, Collberg, Thomborson and Low define properties for the transformation T to be a potent obfuscation technique as follows:

- T should increase the overall program size and introduce new methods and functions for code obfuscation;
- T should introduce new predicates and increase the nesting level of conditional and looping constructs;
- T should increase the number of method arguments and variables dependencies;
- T should increase the height of the inheritance tree.

Resiliency. Since some obfuscation transformation may be easily undone by automated deobfuscation analyses, Collberg, Thomborson and Low introduced the measure of resiliency. This concept captures how well a transformation holds up under attack from an automated deobfuscator. Thus, resiliency of a transformation T can be seen as the combination of two measures:

- the effort required to construct an automated deobfuscator that is able to effectively reduce the potency of T, *i.e. the programming effort*;
- the execution time and space required by such deobfuscator to effectively reduce the potency of T, *i.e. the deobfuscator effort.*

Some highly resilient obfuscation transformation are called *one-way* if they can never be undone (*e.g.* formatting variable names). Other transformations add useless information to the program without changing its observable behavior, but increase the quantity of information an analyst has to work with. These transformations can be undone with varying degrees of difficulty.

Cost. The cost of an obfuscation transformation is the execution time and space penalty it adds to a program. Some trivial transformations are free (*e.g.* variable scrambling) since they cause no additional runtime cost. However, other transformations may have a varying amount of overhead, depending on their use and location. Collberg, Thomborson and Low define the cost of a transformation as follows: let T be a behavior-conserving transformation such that $P \xrightarrow{T} P'$ transforms a source program P into a target program P'. $T_{cost}(P)$ is the extra execution time and space of P' compare to P such that:

- if executing P' requires $\mathcal{O}(1)$ more resources than P then the transformation's cost is *free*;
- if executing P' requires $\mathcal{O}(n)$ more resources than P then the transformation's cost is *cheap*;
- if executing P' requires $\mathcal{O}(n^p)$, $p > 1$, more resources than P then the transformation's cost is *costly*;
- if executing P' requires exponentially more resources than P then the transformation's cost is *dear*.

Quality. Based on the previous definitions of obfuscations metrics, Collberg, Thomborson and Low give a definition of the *quality* of an obfuscation transformation. Let $T_{qual}(P)$ be the quality of transformation T such as:

$$T_{qual}(P) = (T_{pot}(P), T_{res}(P), T_{cost}(P))$$

The quality of a transformation T is thus defined by the combination of the potency, resilience and cost of T.

Other definitions of these four criteria can be found, *e.g.* [7, 100]. However, by using software complexity metrics, those criteria definitions do not take into account the reverse engineering process. This is the reason why more recent work measures these criteria based on an attack model, as described next.

2.1.3.2 Attack model-based obfuscation measurements

Recent studies try to measure the criteria of an obfuscated program (*i.e.* stealth, potency, and resiliency) based on an attacker model. Ceccato *et al.* [37] define human-assisted attacks, *e.g.* debugging, to measure the potency. Their work is based on studies where test subjects are asked to perform specific reverse-engineering tasks on obfuscated codes. Dalla Preda [149] models attacks against obfuscation transformation as abstract domains expressing some properties about the program behaviors. Such methodology allows to compare obfuscation transformations with respect to their potency against various attackers.

More recent works from Banescu *et al.* [13, 14] propose to set-up an attacker model in order to measure the resiliency of obfuscated programs. The attacker model is based on state-of-the-art automatic attacks, *i.e.* generic de-obfuscation technique (*c.f.* Section 2.2.4.4) based on dynamic or concolic symbolic execution engines such as `KLEE` [34] or `Angr` [192] (for other tools, readers can refer to Chapter 2.3). Using two datasets of programs obfuscated with `Obfuscator-LLVM` and `Tigress`, they measured the symbolic execution slowdown applied to both protected and unobfuscated codes, based on an attacker model that aims to obtain a complete code coverage. Moreover, the second dataset is used to compare the different symbolic execution engines, based on another attacker model that aims to reach a specified path. Their work illustrates which obfuscation transformations, or combination of them, bring the higher resilience in their predefined context.

Furthermore, Salem and Banescu introduced the use of machine learning techniques to evaluate the stealth of obfuscation transformation by detecting them (otherwise called *meta-data recovery attack*). However, their approach remains dependent to the type of program evaluated.

Despite the above mentioned approaches, the ability to accurately measure obfuscation transformations criteria remains an open question. The next section will introduce a non-exhaustive list of existing obfuscation transformations.

2.1.4 Obfuscation transformations

Several obfuscation transformations exist, each of them having their own purposes. They can be classified into different categories, such as data obfuscation, static code obfuscation, and dynamic code obfuscation. Many of the described obfuscation techniques appeared first in malware samples, thus, it is difficult to reference the original source. However, early techniques are given by Collberg, Thomborson and Low [45], as well as Collberg and Nagra [44].

2.1.4.1 Data obfuscation

Code obfuscation related to data modifies the form in which they are stored in a program. The purpose of data obfuscation is to hide sensitive information from direct analysis of the code. Usually, these transformations require the program code to be modified so that the original data representation can be reconstructed dynamically, thus during execution of the code. Next, we present a non-exhaustive overview of existing transformation for data obfuscation.

Data reordering. Data variables can be split into several pieces in order to make it more difficult for an analyst to identify them. The mapping between an actual value of a variable and its split representation is managed by several functions. One function is used at obfuscation time whereas the other function reconstructs the original value at runtime. Different types of data, such as integers or string variables, can be obfuscated the same way. Based on the same principle, several data variables can also be combined together, *i.e.* merged, to avoid identification.

Arrays are also prone to reordering. They can be divided into two or more sub-arrays, or multiple arrays can be merged into one. Moreover, other techniques such as *folding* and *flattening* can respectively increase or decrease the number of dimensions of the array [47, 60].

Example 2.1.1. Splitting arrays can be seen as dividing an array A of size n into several arrays B_i, each of them having a size m_i. In this case, it is necessary to have a selection function that determines in which array B_i an element $A[j]$ must be. Moreover, other functions are needed in order to have the localization of each element in the newly created arrays. Figure 2.1 illustrates an example from [52] where an array A is split into two new arrays B and C.

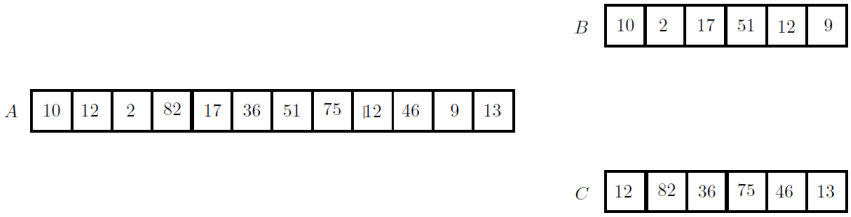

Figure 2.1: Illustration of data reordering transformations on split arrays from [52].

Encoding. Static data within binaries, such as strings or constant values, contain useful information for an analyst. The use of encodings as an obfuscation transformation converts data into a different representation. The transformation is based on special encoding functions to mitigate the need of storing the static data in clear text within the binary. During execution, the inverse function is used to decode the obfuscated data.

Definition 2.1.4. *To obfuscate a variable V in a program, one must convert it from its initial representation to another representation harder for an attacker to analyze, denoted by V'. Thus, any value that V can take during execution must be representable in the obfuscated representation V', as well as any operations performed on V that must be also performed on V'. Finally, two functions are needed to encode and decode V into V' and vice versa.*

Ideally, to prevent pattern-matching attacks, the obfuscated representation must be parameterized in order to have a *family* of representations. In other words, each representation will be different-looking obfuscated variables. However, they all will be based on the same obfuscating algorithm.

Example 2.1.2. Examples of variable encoding are numerous. Listing 2.1 from [44] shows an encoding transformation based on *number-theoretic tricks*. It is used to obfuscate an integer e into the representation $N * p + e$, where N is the product of two close primes, and p is a random value.

```
1  typedef int T4;
2  #define N (53*59)
3
4  T4 encode(int e, int p){
5    return p*N+e;
6  }
7  int decode(T4 e) {
8    return e%N;
9  }
10 T4 encoded_addition(T4 a, T4 b) {
11   return a+b;
12 }
13 T4 encoded_multiplication(T4 a, T4 b) {
14   return a*b;
15 }
16 BOOL encoded_lower_than(T4 a, T4 b) {
17   return decode(a)<decode(b);
18 }
```

Listing 2.1: Encoding transformation on an integer based on number-theoretic tricks

In this example, N must be larger than any integer V we want to represent. Decoding the transformation consists in removing $N * p$ by reducing modulo N. Moreover, addition and multiplication can, in this example, be performed on the encoded representation, unlike the comparison function `encode_lower_than` which requires the `decode` function first.

Converting static data to procedures. This obfuscation transformation replaces static data with a function that calculates the data at runtime. For example, specific constant values can be built during the execution of the binary so that an analyst cannot extract them statically.

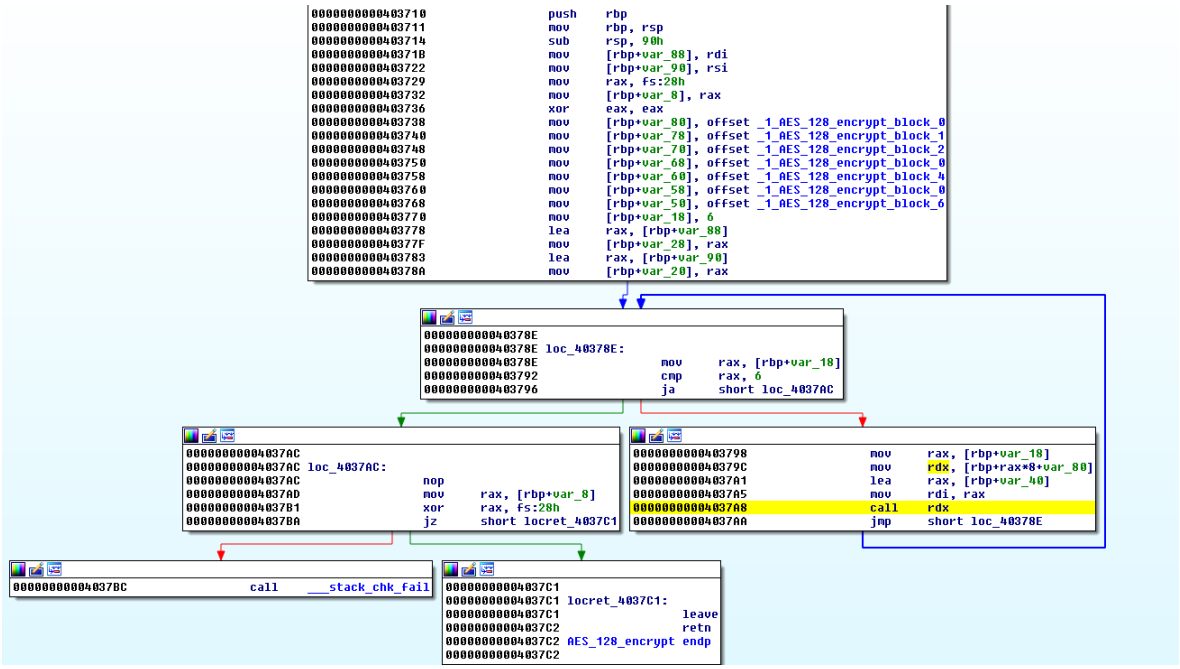

Figure 2.2: Dynamically computed address used for a call and generated with `Tigress`.

Example 2.1.3. One basic example of building static data during code execution can be observed when dynamically computed addresses are used to define where the instruction pointer has to go next. Figure 2.2 illustrates such transformation where the address of the function to be called during execution. We can observe, underlined in yellow, the `call` operation with the register `rdx` as operand, which will contain at runtime the address of the function to go to.

2.1.4.2 Static code obfuscation

Static code obfuscation transformations are similar in some cases to compiler optimizations. They modify the program code during the obfuscation process, or the compilation, but the output will be executed without any runtime (*i.e.* dynamic) modifications.

Instructions substitutions. Each behavior of a program can be implemented in multiple ways [195] and instructions or sequences of instructions can be replaced with syntactically different, yet semantically equivalent code. More complex obfuscations of this type include the replacement of call instructions with a combination of `push` and `ret` instructions [111]. De Sutter *et al* [183] substitute opcodes that are rarely used with more frequently used instructions. This transformation reduced the total number of different opcodes used in the code and normalized their frequency. Moreover, potentially malicious code can also be hidden in side effects of legit-looking sequences of instructions [170].

Example 2.1.4. One explicit example of instructions substitutions can be found with the `MoVfuscator` [1] tool that compiles a program into `mov` instructions.

```
<isprime>:
push   ebp
mov    ebp,esp
sub    esp,0x10
cmp    DWORD PTR [ebp+0x8],0x1
jne    0x8048490 <is_prime+0x13>
mov    eax,0x0
jmp    0x80484cf <is_prime+0x52>
cmp    DWORD PTR [ebp+0x8],0x2
jne    0x804849d <is_prime+0x20>
mov    eax,0x1
jmp    0x80484cf <is_prime+0x52>
mov    DWORD PTR [ebp-0x4],0x2
jmp    0x80484be <is_prime+0x41>
mov    eax,DWORD PTR [ebp+0x8]
cdq
// [[... ... LINES REMOVED ... ...]]
leave
ret
```

Listing 2.2: Un-obfuscated assembly code of an `isprime` function

Listing 2.2 illustrates the assembly code of a function, named `isprime`, which verifies if a given number is prime without any obfuscation transformation applied. Listing 2.3 shows one possible application of instructions substitutions, using the `MoVfuscator`, on the `isprime` function.

[1] the MoVfuscator : `https://github.com/xoreaxeaxeax/movfuscator` [Online; accessed the 01-10-2019]

```
38 <isprime>:
39 mov   eax,ds:0x83fc638
40 mov   edx,0x88048744
41 mov   ds:0x81fc4c0,eax
42 mov   DWORD PTR ds:0x81fc4c4,edx
43 mov   eax,0x0
44 mov   ecx,0x0
45 mov   edx,0x0
46 mov   al,ds:0x81fc4c0
47 mov   ecx,DWORD PTR [eax*4+0x8056ad0]
48 mov   dl,BYTE PTR ds:0x81fc4c4
49 mov   dl,BYTE PTR [ecx+edx*1]
50 mov   DWORD PTR ds:0x81fc4b0,edx
51 mov   al,ds:0x81fc4c1
52 mov   ecx,DWORD PTR [eax*4+0x8056ad0]
53 // [[... ... LINES REMOVED ... ...]]
54 mov   eax,ds:0x83fc628
55 mov   eax,DWORD PTR [eax*4+0x83fc620]
56 mov   DWORD PTR [eax],0x0
```

Listing 2.3: Obfuscated assembly code of the `isprime` function generated by the `MoVfuscator`

Code cloning. Code cloning or copying is a widely used obfuscation technique [171] 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. In other words and from a functional point of view, the original portion of the code and its clone are the same. To prevent the clones 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 [44]. Other uses of code cloning consist in duplicating small functions or of creating semantically equivalent input-dependent paths within a binary in order to prevent state-of-the-art generic deobfuscation techniques [201].

Example 2.1.5. An 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 [48]. This contest consists in building and evaluating white-box implementations of the `AES-128` [54] algorithm. This challenge[2], in order to prevent reverse-engineering, implements over 1200 small functions,

[2]Source code is available at `https://run.whibox.cr.yp.to:5443/show/candidate/777`.

that are 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. Listing 2.4 illustrates two of these cloned sub-functions.

```
void wGzZ(uint oEHmwk, uint KCZu, uint MtCA) {
  ooGoRv[(kIKfgI+oEHmwk)&262143]=ooGoRv[(kIKfgI+KCZu)&262143]^ooGoRv[(
    kIKfgI+MtCA)&262143];
}

// [[... ... LINES REMOVED ... ...]]

void pZwSZ(uint eCFI, uint picb, uint aqQiUv) {
  ooGoRv[(kIKfgI+eCFI)&262143]=ooGoRv[(kIKfgI+picb)&262143]^ooGoRv[(kIKfgI+
    aqQiUv)&262143];
}
```

Listing 2.4: Example of two cloned sub-functions from the challenge `adoring_poitras` of the WhibOx contest.

Opaque predicates. A predicate is a boolean-valued function. An opaque predicate, however, represents an obfuscated predicate with its outcome known at obfuscation time but difficult to determine for a deobfuscator.

Opaque predicates are used to confuse static reverse engineering by adding a evaluation problem that is difficult to solve without executing the code. Sometimes paired with bogus code, opaque predicates [47] are meant to encumber control-flow graphs with redundant infeasible paths. Compared to other control-flow obfuscation transformations such as control-flow flattening or call-stack tampering [112], opaque predicates are supposedly more stealthy (*i.e.* hard for an attacker to detect) because of the difficulty to differentiate an opaque predicate from original path conditions in a binary code. Several types and constructions of opaque predicates exist [128]. The following paragraphs give an overview of them.

Opaque predicate types. Opaque predicates exist under different types. Each of them has specificities and can be constructed in different manners. We denote by ϕ a predicate, *i.e.* a conditional jump within a piece of code. Such predicate can be evaluated to both *true* or *false* (*i.e. 0 or 1*). We denote by $\mathcal{O}$ the obfuscation function that generates opaque predicates, which takes as input a predicate ϕ such that $\mathcal{O}(\phi)$ is the obfuscated version of ϕ, *i.e.* the opaque predicate. By definition, $\mathcal{O}(\phi)$ should be stealthy (indistinguishable from any ϕ) and its value should not be easily known

by an attacker. There are two types of *invariant* opaque predicates and the *two-ways* opaque predicates.

Definition 2.1.5. *Invariant opaque predicates: Let $\mathcal{O}(\phi) : X \to \{0,1\}$ be an obfuscated predicate that evaluate to either 1 (i.e. true), or 0 (i.e. false), and $\mathcal{O}$ be the function that obfuscated the predicate. We denote by X the set of all possible inputs x. Then, we can say that:*

- *if $\forall x \in X, \mathcal{O}(\phi)(x) = 1$ then the predicate is **always true**.*
- *if $\forall x \in X, \mathcal{O}(\phi)(x) = 0$ then the predicate is **always false**.*

Thus, these opaque predicates are of an *invariant* type, because they always evaluate to the same value for all possible inputs.

Example 2.1.6. Figure 2.3 illustrates an *opaquely true* predicate, P^T, generated with the `Obfuscator-LLVM` [96] on a function that check if two given strings are anagrams.

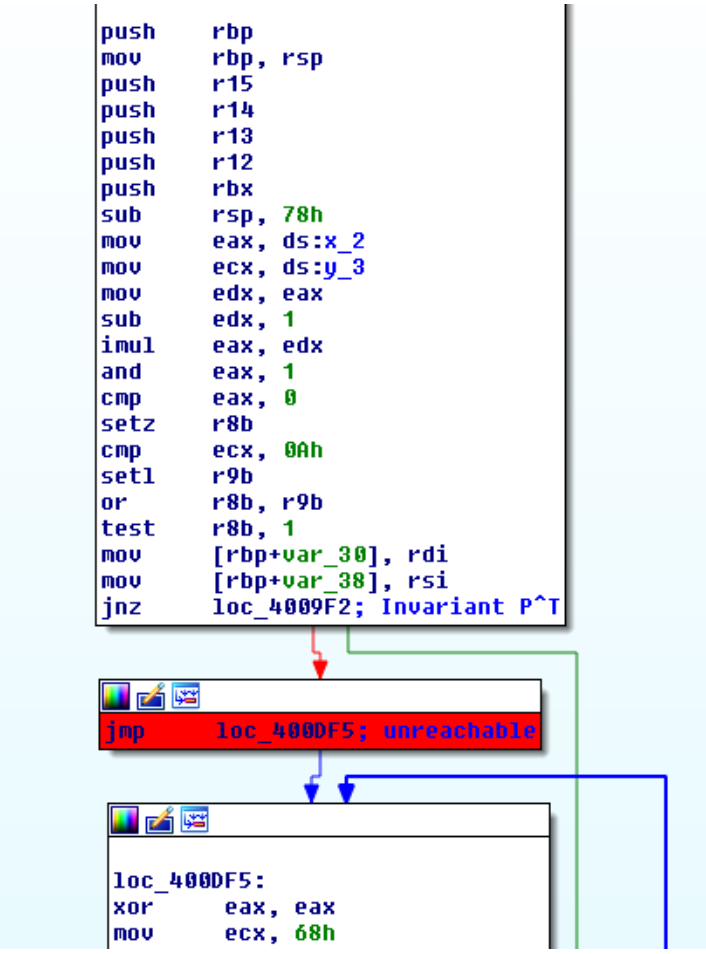

Figure 2.3: Invariant opaquely true predicate generated with `O-LLVM`.

Definition 2.1.6. *Two-ways opaque predicates: Another type of opaque predicate is referred to as two-way, which can be either true or false for all possible inputs. Such a construction requires both branches to be semantically equivalent in order to preserve the functionality of the code that will be executed. In other words we have:*

- *if* $\forall x \in X$, $Pr_{x \leftarrow X}[\mathcal{O}(\phi)(x) = 1] = \frac{1}{n}$ *then the predicate is* ***either true or false****, regardless of the input* x*, for all* $n \in \mathbb{N}^+$.

Example 2.1.7. Figure 2.4 illustrates a two-way predicate $P^?$ generated with the `Tigress` obfuscator [43]. The two basic blocks colored in green are semantically equivalent. Thus, regardless of the $P^?$ output, the same functionality will always be executed.

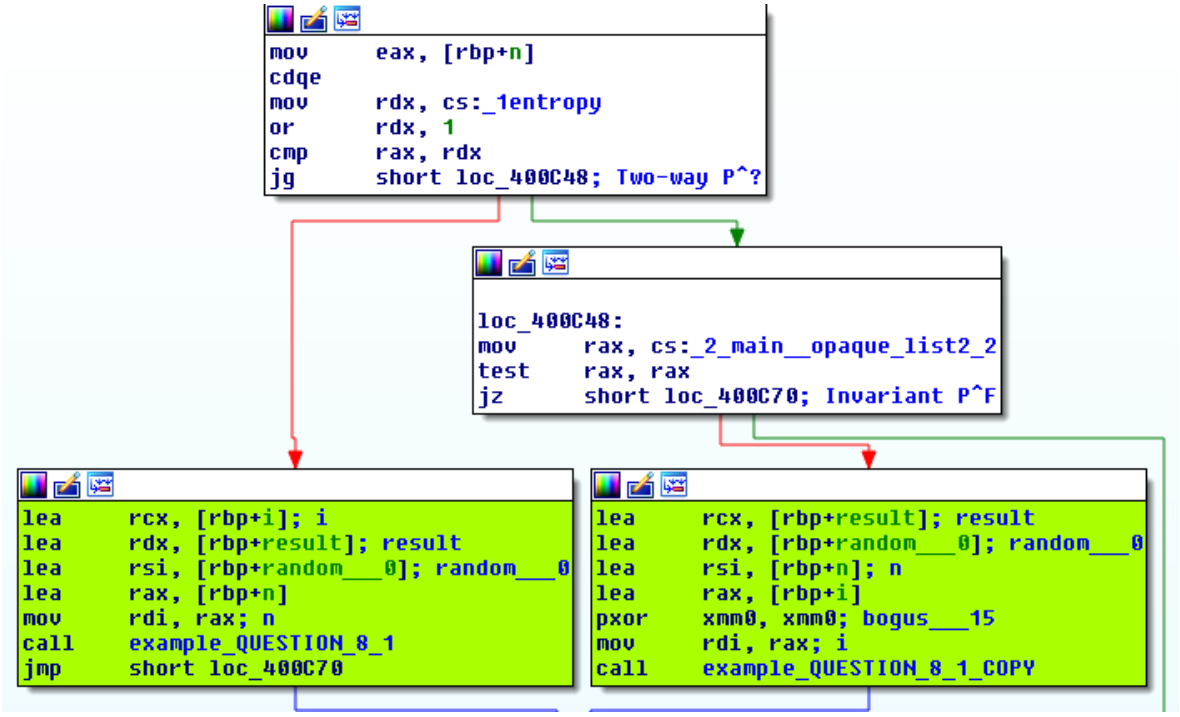

Figure 2.4: Two-way opaque predicate generated with `Tigress`.

Collberg, Thomborson and Low denoted these invariant and two-ways predicates by, respectively, P^T, P^F and $P^?$. Several works use two-ways opaque predicates constructs, either referred to as range-dividers [13] or as correlated opaque predicates [129, 199]. Regardless of their output, *e.g.* their type, there exist many different kinds of constructions in order to render these predicates opaque and make them more resilient against known analyses. We further elaborate on this matter in Chapter 4.

Dead code insertion. Dead code represents parts of a program that will never be reached and thus never executed. The application of dead code as an obfuscation transformation can make the analysis of a program more time consuming, as it increases the amount of code that has to be covered. In practice, dead code insertion if often based on code cloning and also on opaque predicates to make them unreachable during the execution of the program. Figure 2.3 can also be seen as an illustration of dead code insertion combined with an invariant opaque predicate, with the unreachable basic block colored in red. In this example, the instruction `jmp loc_400DF5` will never be executed.

Irrelevant code insertion. Irrelevant or *garbage* code represent sequences of instructions that do not have an effect on the execution of a program. This transformation aims at making analyses more complex. One simple example of irrelevant code consists in the `NOP` instruction as they do not have any effect on the program state. As opposed to dead code, irrelevant code can be reached by the control flow of the program and executed. However it has no effects on a program functionality.

Example 2.1.8. Listing 2.5 illustrates a function in `C` code with one irrelevant line that computes the square of the variable *a*. This line is irrelevant since it does not have any effect on the code functionality.

```
1 int function (int arg){
2   int a = arg;
3   int b = a*a; // irrelevant code
4   int c = a+a;
5   return c;
6 }
```

Listing 2.5: Example of an irrelevant instruction.

Reordering. Similarly to data structures, expressions and statements can be reordered to decrease locality if the reordering does not affect the program behavior. Such techniques were originally introduced for code optimization [11], but they can also be applied for code obfuscation. Moreover, this transformation can be pushed further to move parts of code, or functionalities, into different modules or programs, as studied in the Stuxnet malware [123].

Example 2.1.9. The reordering of instructions is often used by malware authors for metamorphism purposes. Such transformation is only possible if no dependencies exist between instructions. Listing 2.6 illustrates two basic instructions. In this simple case, swapping the two instructions is allowed if: `r1` does not equal `r4`, `r2` does not equal `r3` and `r1` does not equal `r3`.

```
1   add r1, r2
2   add r3, r4
```

Listing 2.6: Instructions snippets.

Loop transformations. Initially, loop transformations have been made to improve the performance and space usage [11]. However, some of these transformations increase the complexity of the code, which makes them potential candidates for code obfuscation. Other transformation can also be applied on loops, such as hiding its condition with the use of an opaque predicate.

Example 2.1.10. One example of obfuscation transformation on loops can be to split them. It consists in breaking down the body of the loop into several others that have the same iteration space. The decomposition must take into account the relationships between the instructions within the body of the loop. Moreover, such decomposition can be made random if the sequence of instructions allows it.

```
57 for (i=1; i<n, i++){
58   a[i] += c;
59   b[i+1] += d*b[i-1]*a[i];
60 }
```

Listing 2.7: Loop sample before the application of obfuscation transformations

Listing 2.7 illustrate a simple loop in `C` code with two instructions and without any transformations applied. Listing 2.8 shows the splitting tranformation of the first loop into two new loops. Each original instruction is dispatched in one of the two generated loops.

```
61 for (i=1; i<n, i++){
62   a[i] += c;
63 }
64
65 for (j=1; j<n; j++){
66   b[j+1] += d*b[j-1]*a[j];
67 }
```

Listing 2.8: Loop sample after the application of a splitting transformation

Function splitting or recombination. Functions cloning describes the concept of splitting the control-flow in two or more paths that look different to the analyst, while they are in fact semantically equivalent. Another type of obfuscation transformation merges the bodies of different functions. The new function will have the parameters of all merged functions, as well as an extra parameter that selects the function body to be executed.

Example 2.1.11. Listing 2.9 shows an example of function recombination. We have two different functions, namely `function1` and `function2`. The obfuscation transformation that merges both functions, *i.e.* `merged_functions_1_2`, takes the parameters of both functions and adds an extra one that selects the body to be executed.

```
void function1 (int arg1){
  // body of the function1
}

void function2 (char* arg2){
  // body of the function2
}

void merged_functions_1_2 (int arg1, char* arg2, int select_function){
  if (select_function == 1){
    // body of the function1
  }
  else{
    // body of the function2
  }
}
```

Listing 2.9: Example of function recombination

Overlapping codes. Overlapping functions reflects a part of the binary code where one function ends with bytes that also define the beginning of another function. Compilers usually use this strategy for optimization purposes, which may also confuse some disassemblers. More sophisticated methodologies where introduced, *e.g.* Jacob, Jakubowski and Venkatesan [90], where two independent code blocks are interleaved in a way that, depending on the entry and exit points of the merged code, a different functionality is executed.

Example 2.1.12. Overlapping instructions is a common anti-disassembly mechanism. The following example from [25] illustrates an execution sequences of bytes extracted from the packer `tELock0.99`.

```
0x1006e7a    fe   04   0b     inc byte [ebx+ecx]
0x1006e7d    eb   ff          jmp +1
0x1006e7e         ff   c9     dec ecx
0x1006e80    7f   e6          jg 0x1006e68
0x1006e82    8b   c1          mov eax, ecx
```

Listing 2.10: Example of overlapping assembly code of `tELock0.99`

In this example, the instruction at the address `0x1006e7d` is `jmp +1` and it is encoded with two bytes. The result is that it jumps to the second byte of its opcode, at address `0x1006e7d+1`, which corresponds to the instruction `dec ecx` that shares the opcode `ff` with the jump instruction. As a result, both instructions overlap each other.

Aliasing. Inserting spurious aliases (*i.e.* pointers to memory locations) can make code analysis more complex, as the number of possible ways for modifying a particular location in memory increases [87, 152]. These pointer-references can also be used as indirection to complicate the reconstruction of the control-flow of a program using static analysis scenarios [191]. Aliasing, as an obfuscation transformation, can also be used to strengthen other transforms.

Control-flow flattening. This class of obfuscation transformation aims at obscuring links between basic-blocks. Wang *et al.* [191] first referred to control-flow flattening as *chenxification*, which puts the basic-blocks of a program into a large switch-statement, called *dispatcher*, that decides where to jump next. Control-flow flattening using a central dispatcher was also described by Chow *et al.* [41]. A similar concept by Lynn and Debray [117] uses what is called *branch functions*, which directs the control-flow to the actual target based on a call table. Popov, Debray and Andrews [148] proposed to replace control transfer instructions by traps that cause signals. The signal handling code then performs the originally intended control-flow transfer. Further control-flow obfuscation techniques are described in [36, 51, 115, 169].

Definition 2.1.7. *The basic definition for flattening a function F consists in the following. First, the body of F is broke up into basic-blocks, that were originally at different nesting levels, into the same level. Then, each basic-blocks are encapsulated in a selective structure (e.g. a switch statement) with each block into separate cases. However, the correct control-flow of F must be ensured by a control variable c_v which represents the state of the program. At the end of each basic-blocks, c_v must be set and used in the predicates of the structure enclosing loop and selection.*

Several methods exists for the selective structure, the following list give an overview of some of them:

- *a **switch dispatch** where each block becomes a case in a switch statement which is wrapped inside an infinite loop,*
- *an **if-and-else dispatch** where each block is accessed trough a series of if and else conditions,*
- *a **goto dispatch** which uses direct gotos to jump between blocks,*
- *an **indirect dispatch** which uses indirect gotos through a jump table to select blocks,*
- *a **call dispatch** where each blocks becomes its own function for which indirect calls through a table of function pointers are used for selection.*

Example 2.1.13. Figure 2.5 illustrates the control-flow graph of a binary search function without any obfuscation applied. Figure 2.6 shows the same function with control-flow flattening. In the second illustration, we can observe a dispatcher combined with several if and else structures in order to select the next bloc of instructions to be executed.

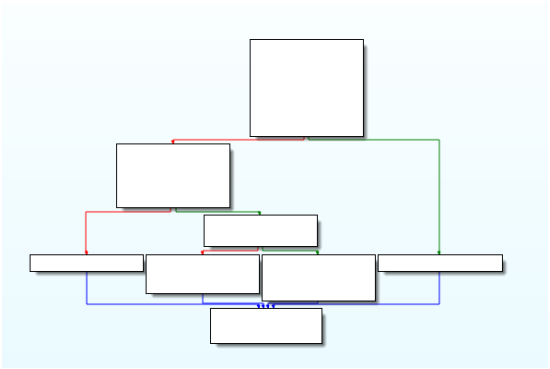

Figure 2.5: Control-flow graph of a binary search function without any obfuscation transformation.

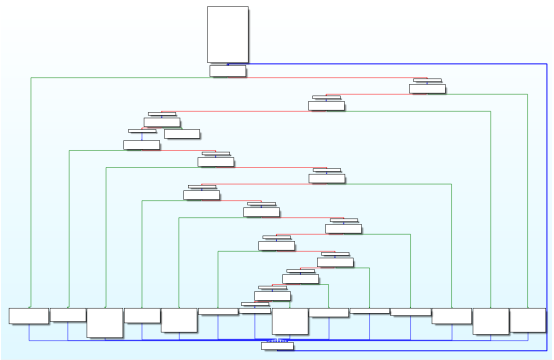

Figure 2.6: Control-flow graph of a binary search function after control-flow flattening.

Name scrambling. Modifying identifier names, such as the ones of variables or functions, and replacing them with random strings is a well known application of source code obfuscation.

Example 2.1.14. Many examples of name scrambling can be found. The illustration in Listing 2.11 is taken from an cryptographic white-box implementation of the `AES` algorithm, submitted during the 2019 WhibOx challenge of the CHES conference.

```
#define _ for
#define __ ++
#define ___ +=
#define ____ +
#define _____ (
#define ______ )
#define _______ {
#define ________ }
#define _________ *
#define __________ !=
#define ___________ ->
#define ____________ ,
#define _____________ [
#define ______________ ]
#define _______________ ;
#define ________________ ^
#define _________________ int
#define __________________ void
#define ___________________ char
#define ____________________ return
#define _____________________ switch
#define ______________________ case
#define _______________________ break
#define ________________________ unsigned
#define _________________________ -=
#define __________________________ %
#define ___________________________ if
#define ____________________________ AES_128_encrypt
#define _____________________________ struct
// [[... ... LINES REMOVED ... ...]]
```

Listing 2.11: Name scrambling in the challenge 21 from the 2019 CHES WhibOx contest

Parallelized code. Initially a technique for software optimization, parallelizing code also became common for code obfuscation. When used as a protective transformation, code parallelization makes the program more difficult to understand compared to any sequential code [45]. In addition, the insertion of useless processes into a program or the parallelization of portions of code that are not mutually dependent often raises additional complexity for analyses.

Removing library calls. The invocations of libraries used in programming languages provides to a reverse-engineer useful information. This is due to the fact that their names cannot be obfuscated. Nonetheless, these calls can be omitted by replacing generic libraries with custom copies, thus allowing their obfuscation. Dynamically tying libraries into the program or merging several small libraries into a large one is also seen as one application of library call obfuscation.

2.1.4.3 Dynamic code obfuscation

The dynamic code obfuscation transformations aim at modifying the code during execution in order to prevent static analysis approaches. In the followings, we introduce some of the existing dynamic obfuscation transformations.

Packing and encryption. Hiding a code by encoding or encrypting it is a dynamic code obfuscation transformation which is also referred to as *packing*. Not only used by legitimate software, packing is also widely used by malicious software to avoid static analyses. Packing a binary involves the usage of a routine that translates back the encrypted or encoded data into interpretable code, during execution. By changing the encryption or encoding keys, a packed program code can easily be rewritten upon distribution to complicate simple pattern matching analysis. The packing technique can also be combined with *code polymorphism* [135]. Such transformation is used for several reasons. First, since packing often employs compression, this technique renders reduction of the storage space. Second, it also employs other obfuscation transformations in order to obscure the code. Therefore, numerous commercial packers exist such as `VMProtect` [189], `ASPack`, `Armadillo`, or `Themida` [142]. Malware authors also widely use these tools in order to hide the malicious intents of their code.

Several works study packing as an obfuscation transformation. Cappaert *et al.* [35] present a different form of packing, where code can be decrypted at a fine granularity right before execution. To that purpose, they use a key which is built from other code sections. Wu *et al.* [198] discuss a polymorphism based concept called *mimimorphism*. Their approach aims at confusing the analysts by encoding data to make them look like code. Mavrogiannopoulos, Kisserli and Preneel *et al.* [124] also provide a taxonomy of transformations and packing techniques. Other work [164] surveys the different approaches used by malwares for packing.

Dynamic code modification. Obfuscation transformations based on dynamic code modification conceal the code by providing a general template in memory. The latter is patched right before its execution [45]. Therefore, evaluation methodologies based on static analysis are inefficient as the program functionality is only available during its execution. Other works [98, 121] construct dynamic code modification by deliberately adding incorrect erroneous of code and correcting them just before their execution.

Environmental requirements. Riordan and Schneier [159] suggested the idea of environmental key generation in which a cryptographic key is not placed dynamically in a binary, but constructed from data collected from the computing environment. The program is only able to generate the key and execute the code if a specific environmental condition is met, *i.e.* the activation environment. Otherwise, no sensitive data are revealed. Therefore, similar concepts are extended for code obfuscation. Sharif *et al.* [174] proposed a malware obfuscation scheme that makes the code conditionally dependent on an external trigger value. The activated action is shielded from dynamic analysis without the awareness of this specific value. Similar techniques can be widely observed in malicious software.

Hardware-assisted code obfuscation. In order to enhance the resiliency of the overall applied obfuscation transformations, hardware tokens are a solution [22, 73, 212]. The main idea is to bind both hardware and software by making the execution of the software dependent on some hardware token. Therefore, analyses of the software will fail without the token due to the lack of important information (*e.g.* targets of indirect jumps).

```
68  int function(int arg){
69  int a = arg;
70  int b = 2;
71  int c = a * b;
72  return c;
73  }
74
75  /*
76  ** byte-codes equivalence:
77  ** 31 ff 00 09:    mov r0, r9
78  ** 31 01 02 00:    mov r1, 2
79  ** 44 00 00 01:    mul r0, r0, r1
80  ** 60:         ret
81  */
```

Listing 2.12: A simple function written in C code with its associate byte-code

Code virtualization. Virtualization consists in the translation of a program functionality into byte code for a specific and customized virtual machine (VM) interpreter that is integrated in the program [77, 106]. Code virtualization can also be combined with other transformations such as *code polymorphism* [6]. Other work [190] proposes a combination of fine-grained encryption and virtualization to hide the virtual machine code from analysis. Collberg, Thomborson and Low [45] described a variant of this concept under the term *table interpretation*. A similar concept by Monden, Monsifrot and Thomborson [131] uses a finite state machine-based interpreter to dynamically map between instructions and their semantics.

Example 2.1.15. Listing 2.12 illustrates a function written in `C` code, from [167], that is to be virtualized. The disassembly of the function byte-code is listed as comments. Once the code is compiled into the virtual machine byte-code, it must be interpreted by the virtual machine itself.

```
82 void virtualisation(ulong vpc, struct vmgpr* gpr){
83   while(1){
84     // Fetch and decode
85     struct opcode* i = decode(fetch(vpc));
86     // Dispatch
87     switch (i->getType()){
88       // Handlers
89       case ADD: // byte-code 0x21
90         gpr->r[i->dst] = i->op1 + i->op2;
91         vpc += 4;
92         break;
93       case MOV: // byte-code 0x31
94         gpr->r[i->dst] = i->op1;
95         vpc += 4;
96         break;
97       case MUL: // byte-code 0x44
98         gpr->r[i->dst] = i->op1 * i->op2;
99         vpc += 4;
100        break;
101      case RET: // byte-code 0x60
102        vpc += 1;
103        return;
104    }
105  }
106 }
```

Listing 2.13: Application of code virtualization on a function written in `C` code

The sample of code illustrated by Listing 2.13 shows the virtualization of the function which is called with an initial `vpc` pointing to the first opcode of the byte-code (*e.g.* the virtual address of instruction `mov r0, r9`). Once the opcode has been fetched and decoded by the VM, the dispatcher points to the appropriate handler to virtually execute the instruction and then, the handler increments `vpc` to point to the next instruction to execute and so on until the virtualized program terminates. As we can see, the control flow of the original program is replaced by a dispatcher pointing on all handlers.

Anti-debugging and disassembly techniques. This class of transformations involves techniques that prevent analyses based on disassemblers or debuggers. As an example, the use of a debuggers can be detected using timing or latency analyses. Software breakpoints also induce specific code modifications that can be detected. The execution of undocumented or rarely used instructions is another approach used to confuse a tool or a human analyst [27]. M.V. Yason [204] gives an overview of existing anti-debugging and disassembly tricks present in malware.

Example 2.1.16. Listing 2.14 shows an example from [204] of a code made to identify if a debugger is present using the `IsDebuggerPresent()` API and the `PEB.BeingDebugged` flag.

```
107  // call kernel32!IsDebuggerPresent()
108  call    [IsDebuggerPresent]
109  test    eax,eax
110  jnz     .debugger_found
111
112  // check PEB.BeingDebugged directly
113  mov     eax,dword [fs:0x30]     // EAX = TEB.ProcessEnvironmentBlock
114  movzx   eax,byte [eax+0x02]     // AL  = PEB.BeingDebugged
115  test    eax,eax
116  jnz     .debugger_found
```

Listing 2.14: Examples of anti-debugging tricks written in assembly language

However, these checks are obvious thus many obfuscators, such as packers, add other layers of obfuscation upon them to enhance the stealth of the anti-debugging and disassembly tricks.

2.2 Deobfuscation

The term deobfuscation encompasses all techniques aiming at evading the protections introduced previously by obfuscation. Namely, the deobfuscation process can be seen as:

1. Reverting applied obfuscation transformations;
2. Simplifying the obfuscated code;
3. Collecting information about the obfuscated code.

There exist many approaches and methodologies to deobfuscate programs. In the following sections, we present each approach, namely static, dynamic, and symbolic deobfuscation techniques.

2.2.1 Static deobfuscation techniques

Static analysis is widely used for code optimizations, finding or proving the absence of bugs, and also for reverse-engineering. More generally, it refers to any program analysis that is performed by inspecting the executable code or a disassembled representation of a program, without executing it. Thus, static analysis is also used for software deobfuscation since it deduces information about a program by reasoning on the possible executions it has. We say that a static analysis method is considered to be *sound* if it is guaranteed that it includes all possible execution path of the program. However, because of undecidability questions, static analysis can only achieve soundness by *over-approximating*, *i.e.* by generalizing the concrete program behavior thus accepting execution paths that will not occur in real executions. Many static data-flow analyses have been proposed to address obfuscation transformations, *e.g.* data dependency, alias analysis, and abstract interpretation [138, 150]. However, they are prone to limitations, as static analysis that can be thwarted by obfuscation techniques such as control-flow flattening, which forces the analysis to lose precision.

2.2.2 Dynamic deobfuscation techniques

Dynamic analysis is an important part of todays forensic and malware analysis [62]. It studies real executions of a program, either *online* (*i.e.* during the execution) or *offline* (*i.e.* using a recorded trace). Dynamic analysis is conceptually dual to static analysis, as a sound dynamic analysis considers a subset of all execution paths of a program and is therefore an *under-approximation*. Thus, each observed behavior is guaranteed to also occur during at least one execution. Nevertheless, if the number of possible traces in a program is too large to be analyzed exhaustively, dynamic analyses can miss certain execution paths. Also known as *code coverage issue*, this limitation is usually inevitable due to undecidability results. Thus, while static analysis balances precision against cost, dynamic analysis compromises *coverage* against cost.

2.2.3 Symbolic deobfuscation techniques

State-of-the-art deobfuscation methodologies are nowadays based on symbolic execution techniques. While other existing approaches are essentially divided into static and dynamic methods, *symbolic approaches* offer a valuable balance between both. Given a path in a program, the main insight of *symbolic execution* techniques is the possibility in many cases to compute a formula (*i.e.* a path predicate) such that a solution to this formula is a test input exercising the considered path. Then, exploring all the bounded paths of the program allows intensive testing. First introduced by King [105], symbolic execution techniques re-emerged when it was mixed with concrete execution [173, 196] and with the satisfiability modulo theories (SMT) solvers. Thus, symbolic execution has successfully been applied in a wide range of security applications, such as vulnerability and malware analysis [10, 85]. More precisely, static symbolic execution techniques capture the semantics (*i.e.* logic) of a program by assigning a generic value to all inputs. An interpreter is then used to trace the program, which uses symbolic values for the calculations rather than obtaining concrete values as a normal execution would. A symbolic state $\mathbb{S}$ is built and consists in a set of symbolic expressions S for each variables (*e.g.* registers, memory, flags, etc.). Several techniques exist for symbolic execution [12], however, it suffers from the same limitations listed for static deobfuscation techniques. Thus, dynamic symbolic execution (DSE) [173, 196], also known as *concolic execution*, is widely used for deobfuscation purposes. It takes the advantages of a concrete execution path to perform the symbolic execution and provides the following conveniences:

- the sound execution of the program since the paths are sure to be feasible in practice;
- the next instruction executed is always known;
- all loops are unrolled.

Dynamic symbolic execution runs and interprets a program using its concrete state, as opposed to static symbolic execution which simulates the execution of the program with symbolic values. Then, SMT solvers are used to generate new concrete state and explore more paths to enhance the coverage. Recently, DSE methodologies are used to either extract the protected code [94] or to reduce the complexity of a control-flow graph of an obfuscated binary [201], contributing in this way to an improved analysis of obfuscated programs.

We discuss next existing evaluation methodologies for obfuscated programs. First, we introduce techniques that are specific to some transformations. Finally, we describe generic deobfuscation methodologies.

2.2.4 Existing evaluations techniques

In this section we provide an overview of existing state-of-the-art evaluation, *i.e.* deobfuscation, methodologies created for specific obfuscation transformations. We can see the evaluations techniques as two different approaches:

- the use of advanced analysis techniques (*e.g.* symbolic execution) against general transformations to simplify a program or recover key parts of the implemented algorithm,
- the use of specific analysis techniques to target a precise obfuscation transformation.

2.2.4.1 Evaluation of opaque predicates

The insertion of opaque predicates is a commonly used obfuscation transformations. Thus, they are the target of several published attacks, each of them having their strengths and limitations as described below.

Probabilistic check is a first deobfuscation methodology which consists in executing n times a program segment to see if a predicate is invariant [213]. This technique is however prone to high false positives and negatives results.

Pattern matching attack (otherwise called dictionary attack) [63], consists in taking obfuscated predicates from a program being attacked and pattern-matches the source code against known examples. However, the possibility to build variants of opaque predicates that cannot be matched using dictionary attacks implies a high false negative rate.

Abstract interpretation [150] employed by static analysis is another technique that provides correctness and efficiency in the deobfuscation of specific constructions of opaques predicates.

Another recently introduced technique [23] uses *program synthesis*. Originally made for the deobfuscation of virtualized code, this approach has been successful for the simplification of MBA expressions.

Current state-of-the-art deobfuscation approaches use *automated proving* [17, 129]. Udupa, Debray and Madou [187] use static path feasibility analysis based on the SMT solvers to determine the reachability of execution paths. However, their methodology is prone to the limitations of static analysis. Thus, more recent approaches are based on dynamic symbolic execution [17].

Yet, automated proving based analysis, either static or dynamic, may suffer from symbolic execution limitations as well as SMT solvers restraints. Indeed, it has been showed that SMT solvers fail against MBA opaque predicates [211], whereas symbolic execution can be slowed down effectively, or even misguided, as in the case of alias-based constructions or more recent opaque predicates constructions such as the bi-opaque ones [200].

Overall, automated proving is currently considered the most effective methodology against opaque predicates.

2.2.4.2 Evaluation of control-flow flattening

The control-flow flattening obfuscation transformation aims at obscuring the control-flow logic of a program by placing the basic block at the same nesting level. In that way, each basic blocks will have the same set of predecessors and successors. Thus, the actual control-flow is preserved using a selection of *dispatcher variables*. Many constructions exist for the selection methodology of the control-flow flattening. However, several works have presented deobfuscation and evaluation techniques against them.

Udupa, Debray and Madou [187] present in their work different analyses and program transformations that are, according to them, useful against control-flow flattening. They first propose to use *static path feasibility analysis*, which refers to a constraint-based static analysis that determines whether an execution path is reachable, *i.e.* feasible. Such analysis is used for different obfuscation techniques such as opaque predicates, however the subtility of the analysis resides in the construction of the path constraints for which several variants exists. They use a conservative approximation, taking into account the effects of arithmetic operations on the values of variables used and propagating the information along a single execution path to verify its feasibility. However, their main approach consists in an hybrid methodology that combines the conservative static analysis with dynamic analysis. The latter is used to get an under-approximation of the set of control-flow edges taken during execution on which they can apply the static path feasibility analysis. They also propose an alternative in which they start with an over-approximation with the static path feasibility analysis and then use dynamic analysis to remove the paths that are not actually taken at runtime.

More recently, the works of Yadegari and Debray [202] and Yagedari *et al.* [203] propose to use a concolic approach combined with taint analysis at a bit-level of granularity to avoid the existing limitations. Since analyses based on symbolic execution are important when dealing with obfuscated programs, their precision is essential as well. Identifying too many execution paths can lead to path explosions, while missing some execution paths can leads to code coverage issues. In their methodology, they first start by identifying input and output values, namely, any value that is obtained from the command line or defined by a routine (*i.e.* an input) and any value that is defined by an instruction or read by a routine (*i.e.* an output). They also use a combination of taint propagation and control-dependence analysis to identify instructions within execution traces, which are related the input and output values. Afterwards, semantics-preserving code simplifications are applied, in order to use the new simplified trace for the generation of a reduced control-flow graph. Simplifications on the control-flow graph are also applied in order to eliminate spurious execution paths. Figure 2.7 illustrates the application of their methodology on a binary

search algorithm obfuscated using `ExeCryptor` [180].

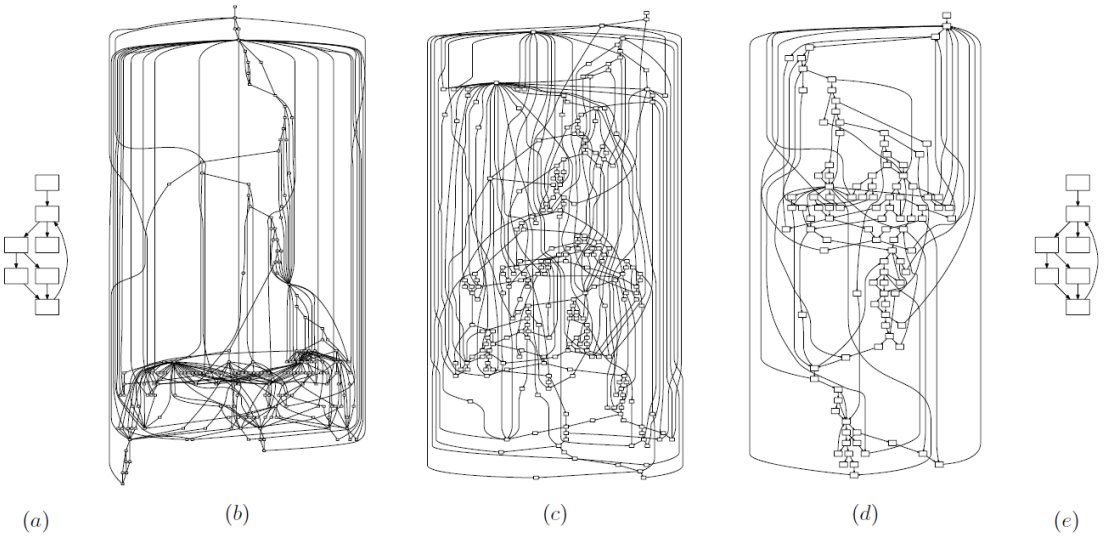

Figure 2.7: Control-flow graph reduction results from [203]

2.2.4.3 Evaluation of code virtualization

Code virtualization protects a program from reverse engineering by compiling it into byte-code, for a randomized virtual architecture, and with a corresponding interpreter. Thus, most static analyses are inefficient against this kind of obfuscation transformations, where only the code of the interpreter is directly visible. Rolles [162] introduced how to analyze and de-obfuscate programs obfuscated with code virtualization by means of manual attacks. Guillot and Gazet [79] present how to use an intermediate representation to convert the virtual machine byte-code into its original assembly instructions. Moreover, they illustrate that the reverse engineering of the complete interpreter is not required when using symbolic execution. However, the approaches of both Rolles as well as Guillot and Gazet are dependent to minor modification of the obfuscation transformation scheme, and may be time consuming due to the lack of automation.

Another static analysis methodology is given in [103, 104]. These works lift a *location-sensitive* analysis to be used in the presence of virtualization-based obfuscation schemes.

Dynamic analyses also exist for the evaluation of code virtualization. Coogan, Lu and Debray [50] show how to use execution traces with symbolic execution and taint analysis on predefined values, to reason about the inner workings of a protected binary. For that, they focus in the interaction of

the program with its environment in the form of system calls. Therefore, they use taint tracking of the system call values and trace them back throughout the execution. Since the effect of code virtualization is similar to control-flow flattening regarding the dispatching (*i.e.* selection) loop, the work of Yadegari *et al.* [203] is also used to evaluate both. Their approach is similar to the Coogan, Lu and Debray work as it is based on dynamic analysis, concolic execution and taint tracking. Note that, they taint the input and output values as described in the previous paragraph (*c.f.* Section 2.2.4.2). Other works [97] are based either on hybrid analysis (*i.e.* combining both existing static and dynamic approaches) or propose a methodology to de-obfuscate and recompile the simplified code [167] using symbolic path exploration and taint analysis. However, these approaches suffer from the dynamic analysis limitations, such as code coverage.

Therefore, recent work uses *program synthesis* [23] to obtain the semantics of the virtualized code. This work demonstrate how Monte Carlo Tree Search [31] can be used to compute a simplified expression that represents a deobfuscated version of the input.

2.2.4.4 Generic evaluation techniques

Since obfuscation-specific evaluations techniques may have the limitation to be only effective against previously-seen obfuscation transformations, generic deobfuscation methodologies have been introduced.

The first step of most generic deobfuscation methods consists in generating execution traces of a protected binary. Using forward and backward taint analysis [172], 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 combined with a constraint solver.

A first generic approach has been established by Yadegari *et al.* [203]. We presented their methodology as an evaluation methodology against control-flow flattening transformation. However, since they make only few assumptions about the analyzed code, their semantic approach is considered generic in order to face previously unseen obfuscation techniques. Yadegari and Debray [202], in their methodology, 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.

More recent approaches for a generic deobfuscation also emerged (*e.g.* [167]). They are based on the same principal, namely, that deobfuscation is a problem of identifying and simplifying the code that affects the input to output transformation. However, as presented before, this methodology differs in the manner that they capture and map the semantics of the program. Salwan *et al.* [94], [167] use transformations at the `LLVM` intermediate representation [114], which allow them to build a deobfuscated binary directly from the collected traces. Their approach succeeds against most of the Tigress challenges [43]. A similar approach from Garba and Favaro [75] exploits the `LLVM`

intermediate representation with existing optimizations passes to provides a generic deobfuscation framework which has been shown efficient against existing licensed obfuscators.

Yet, the main drawback is that generic deobfuscation techniques based on DSE often needs execution traces, which requires inputs generation. This may be time consuming and make code coverage and scalability the main issue 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) that makes the deobfuscation more difficult in terms of scalability.

Finally, recent works propose to counter DSE-based deobfuscation approaches [13, 140]. For example, Banescu *et al.* [13] propose novel obfuscation transformations (*e.g.* range dividers) that deliberately explodes the number of paths through a function, thus increasing the search space for dynamic symbolic execution engines. They also propose new ways of improving existing obfuscation transformations against these attacks by altering the functionality property of an obfuscator definition (*c.f.* Section 2.1.3). More generally, their methods exploit existing issues of symbolic execution, *e.g.* path explosion, path divergence, and complex constraints [5].

2.3 State-of-the-art tools

In this section we present some state-of-the-art obfuscation and deobfuscation tools. We start by presenting publicly available and well-known obfuscators used in this thesis, along with the obfuscation transformations that they can generate.

2.3.1 Obfuscators

The role of an obfuscator is to add layers of transformation in a code. To this end, several possibilities exists:

1. The user can apply obfuscation transformations on source code for portability;
2. The user can apply obfuscation transformations during compilation;
3. The user can apply obfuscation transformations on the binary.

Most of the existing obfuscators use the first or the second solution. In the followings, we first present the `Obfuscator-LLVM` that applies the transformations during compilation using the `LLVM` intermediate language. Afterwards, we introduce `Tigress` that works directly on the source code, generating a new obfuscated version of it.

2.3.1.1 Obfuscator-LLVM

`Obfuscator-LLVM` [96] (`OLLVM`) is a project initiated in June 2010 by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains (HEIG-VD). It provides an open-source fork of the `LLVM` compilation suite able to provide increased software security through code obfuscation and tamper-proofing. They work at the intermediate representation level. `OLLVM` is compatible with several programming languages and target multiple platforms.

`Obfuscator-LLVM` supports three obfuscation transformations, as described in the following paragraphs.

Bogus control-flow. The bogus control-flow obfuscation transformation of `OLLVM` modified a function call graph by adding a new basic block before the current original one. The generate basic block contains an invariant and thus bloc-centric opaque predicate.

Instruction substitution. In order to bring diversity into a program, instruction substitution aims at replacing standard binary operators (*e.g.* addition, subtraction, boolean operators) by functionally equivalent ones. The substitution is generated as a more complicated sequences of instructions.

Control-flow flattening. As presented in Section 2.1.4, control-flow flattening aims at obscuring links between basic-blocks, by leveling-out each basic blocks. `OLLVM` algorithms fully flattens the control-flow and uses an if-nest based construction.

2.3.1.2 Tigress

`Tigress` is a diversifying virtualizer and obfuscator for the C language. It supports many novel defenses against both static and dynamic reverse engineering and de-virtualization attacks. `Tigress` is freely available, with a large collection of obfuscation transformations. The obfuscator operates on the `C` language, at the source code level. In particular, `Tigress` protects against static de-virtualization by generating virtual instruction sets of arbitrary complexity and diversity, by producing interpreters with multiple types of instruction dispatch, and by inserting code for anti alias analysis. `Tigress` protects against dynamic de-virtualization by merging the real code with bogus functions, by inserting implicit flow, and by creating slowly-executing re-entrant interpreters. `Tigress` implements its own version of code packing through the use of runtime code generation. Finally, `Tigress` dynamic transformation provides a generalized form of continuous runtime code modification.

The following paragraphs presents the obfuscation transformations that can be generated with `Tigress`.

Major transformations. `Tigress` can generate several major transformations, mainly to prevent static analysis of the code. These transformations are the followings:

Virtualize: Code virtualization, or *Virtualize*, turns a functions into an interpreter. The byte-code language of the interpreter is specialized. `Tigress` generate this transformation by first constructing type-annotated abstract syntax tree (AST) from the input `C` code. Based on the AST, it generates control-flow graphs of instruction trees. Then, `Tigress` selects a random instruction set architecture (ISA) and generates a byte-code program specialized for the input function. The dispatch methodology is also randomly selected, before generated the obfuscated output program.

`Tigress` supports different mechanisms in order to generate an ISA with a high degree of diversity and stealth, either statically or dynamically. Moreover, the dispatch method selection can be made with different constructions, such as switch-based, direct, indirect or call-based dispatch.

Jit: The Jit transformation is an example of *runtime code generation*, as presented in Section 2.1.4.3. It translates a function F into a new function F' which consists in a sequence of intermediate code instructions. With such, when F' is executed, F will be dynamically compiled to machine code.

Jit Dynamic: Based on the previous *Jit* transformation, `Tigress` propose a dynamic variant which continuously modifies and updates the jitted code during execution.

Supporting transformations. In order to reinforce major transformations, `Tigress` also provides supporting techniques as describe in the followings.

Flatten: The Flatten transformation apply control-flow flattening on a given function. As opposed to `OLLVM`, `Tigress` allows more diversity. First, it propose several constructions for the dispatch method, such as switch, goto, indirect or call-based techniques. Second, flattened basic-blocks can be split-up, and their order randomized. `Tigress` also supports encodings and the insertion of opaque predicates in order to calculate the next basic-block to reach, or the conditional branches generated.

Merge: The Merge transformation combines multiple functions into one. The transformation also merges the argument list and the local variables of targeted functions. By doing so, an additional argument is added in order to select the desired function to be called and executed.

Split: As opposed to the Merge transformation, `Tigress` allows to split pieces of function into their own functions. Such transformation can be useful to bring more stealth on large obfuscated

function. By dividing them into smaller pieces, they become less conspicuous. `Tigress` supports different splitting methods for this transformation.

RndArgs: The RndArgs transformation randomize the order of arguments to a function and can also add bogus ones.

AddOpaque: `Tigress` can generate opaque predicates using the AddOpaque transformations. It supports many invariant constructions, *e.g.* structured-based, input-based or environment-based, as well as two-ways opaque predicate types. Moreover, when combined with encodings, `Tigress` opaque predicates can be formed as mixed-boolean and arithmetic constructions.

Encodings: `Tigress` implements many types of encoding transformation, namely *EncodeData*, *EncodeLiterals* and *EncodeArithmetic*. The *EncodeData* transformation replace integer variables with non-standard representations. The goal is that a variable real value is never revealed until it is printed or it escapes the program. The *EncodeLiterals* transformation obfuscate integers and string literals. It can also replace them with opaque expressions. Finally, the *EncodeArithmetic* transformation replace integer with more complex expressions, often combining arithmetic and boolean operators.

Anti-analysis transformations. In order to strengthen the obfuscated code against existing deobfuscation analyses, `Tigress` proposes several transformations. First, the *AntiBranchAnalysis* makes harder for automated static tools such as disassemblers to determine the target of branches in the code. Transformations such as *AntiAliasAnalysis* and *AntiTaintAnalysis* disrupt static or static analysis tools that make use of, respectively, inter-procedural alias analysis or taint analysis. Finally, `Tigress` allows the used API calls to be hidden using the *EncodeExternal* transformation.

2.3.2 Deobfuscation tools

In this section we discuss some of the tools that can be of use when reverse engineering or evaluating obfuscated binaries. The following list is not exhaustive, but based on tools studied or used in this thesis.

2.3.2.1 Interactive DisAssembler

Interactive DisAssembler[3], mainly known as `IDA`, is the state-of-the-art reverse engineering tool [61]. It is widely used for software reverse engineering, with built-in command languages. `IDA` supports a number of executables formats for variety of processors and operating systems. Moreover, `IDA` is used by a wide community of analysts that contribute to the tool by creating plug-ins that extend the disassembler functionality even further. Among these plug-ins, we can find:

- *Optimice*[4]: This plug-in enables you to remove some common obfuscations (*e.g.* dead code, `jmp` removal) and rewrite code to a new segment.
- *VMAttack*[5]: Static and dynamic virtualization-based packed analysis and deobfuscation.
- *HexRaysDeob*[6]: A Hex-Rays microcode API plugin breaking an obfuscating compiler used to create an in-the-wild malware family.
- *BinCAT*[7]: BinCAT is a static Binary Code Analysis Toolkit, designed to help reverse engineers, directly from `IDA`.

Many other plug-ins exists for various uses and applications. `IDA` also possesses a decompiler[8] for several architectures (*e.g.* `x86-x64`, `ARM` and `PowerPC`). However, when dealing with obfuscated code `IDA` may sometimes face some issues. In this thesis, our contributions are mainly implemented as plug-ins for `IDA`. However, there exist other tool for the same applications such as `Hopper`[9], `Binary Ninja`[10], `OllyDbg`[11], `radare2`[12], `CFGRecovery` (`Insight Framework`)[13] or `Ghidra` as introduced in the following paragraph.

[3] `https://www.hex-rays.com/products/ida/`
[4] `https://code.google.com/archive/p/optimice/`
[5] `https://github.com/anatolikalysch/VMAttack`
[6] `https://www.hex-rays.com/contests/2018/index.shtml`
[7] `https://github.com/airbus-seclab/bincat`
[8] `https://www.hex-rays.com/products/decompiler/`
[9] `https://www.hopperapp.com/`
[10] `https://binary.ninja/`
[11] `http://www.ollydbg.de/`
[12] `https://rada.re/r/`
[13] `https://insight.labri.fr/`

2.3.2.2 Insight Framework

The `Insight Framework` is made for binary analysis, with several purposes such as binary verification, reverse-engineering, binary test-cases extractions and decompilation [18, 39]. It also contains the `CFGRecovery` tool which aims at recovering a program control-flow and any information that can be rebuilt in under-approximation.

2.3.2.3 Ghidra

`Ghidra`[14] is a newly release reverse enginering tool made by the US National Security Agency. `Ghidra` software reverse engineering framework is a full-featured, high-end software analysis tools that enable users to analyze compiled code on a variety of platforms including `Windows`, `Mac OS`, and `Linux`. Capabilities include disassembly, assembly, decompilation, graphing, and scripting, along with hundreds of other features. `Ghidra` supports a wide variety of processor instruction sets and executable formats and can be run in both user-interactive and automated modes. Users may also develop their own `Ghidra` plug-in components and scripts using `Java` or `Python`

2.3.2.4 Metasm

`Metasm`[15] is an open source framework developed by Yoann Guillot. It is an open source software for manipulating binary executable files and covering a wide range of actions, allowing the compilation of source files as well as the disassembly of binaries, through process debugging and shell-code analysis.

2.3.2.5 Miasm

`Miasm`[16] is a reverse engineering framework developed by Fabrice Declaux. It offers PE and ELF manipulation, assembling and disassembling. The framework possess its own intermediate language called MiasmIR, thus most common instructions have their semantics encoded as a list of MiasmIR expressions. One of the main motivations behind the design and implementation options of this framework is to circumvent current limitations of existing malware and binary programs analysis solutions. All the contributions of this thesis are based on MiasmIR and `Miasm` symbolic execution engine.

[14] `https://ghidra-sre.org/`
[15] `https://github.com/jjyg/metasm`
[16] `https://github.com/cea-sec/miasm`

2.3.2.6 Angr

Angr[17] is a platform-agnostic binary analysis framework developed by the Computer Security Lab at UC Santa Barbara and their associated CTF team, Shellphish. Angr is a suite of python libraries that allows user to load and binary and process several analyses on it such as:

- Disassembly and intermediate-representation lifting;
- Program instrumentation;
- Symbolic execution;
- Control-flow analysis;
- Data-dependency analysis;
- Value-set analysis (VSA).

Angr was also one of the underpinnings of Shellphish's Cyber Reasoning System for the DARPA Cyber Grand Challenge, enabling them to win third place in the final round.

2.3.2.7 Triton

Triton[18] is a dynamic binary analysis (DBA) framework. It provides internal components like a Dynamic Symbolic Execution (DSE) engine, a dynamic taint engine, AST representations of the x86, x86-64 and AArch64 Instructions Set Architecture (ISA), SMT simplification passes, an SMT solver interface. As previously discussed, Triton succeeds against most of the Tigress challenges [43].

2.3.2.8 BINSEC

BINSEC[19] is a developed tool by the CEA List in collaboration with Verimag and LORIA. The general objective of the open-source platform is to encourage the next generation of binary-level analysis tools. The open-source framework aim at filing bringing together formal methods over executable code and binary-level security analyses. Thus, it is based on a binary-level semantic approaches. The BINSEC framework targets domains such as vulnerability analyses, malware analyses, code protection and binary-level verification.

In the following chapters, each contributions of this thesis are introduced. We start by presenting DoSE, our deobfuscation methodology based on semantic equivalence.

[17] https://angr.io/
[18] https://triton.quarkslab.com/
[19] https://binsec.github.io/

Chapter 3

Deobfuscation based on Semantic Equivalence

Contents

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 in this chapter our first contribution which is 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

3.1 Introduction

Recent binary deobfuscation techniques [17, 50] based on dynamic symbolic execution emerged in order to face obfuscation techniques such as code virtualization [141, 142, 189] or control-flow flattening [115, 191]. Generic deobfuscation methods have appeared in order to deobfuscate protected binaries, as introduced in Chapter 2. Such techniques can either extract protected code [94] or reduce the complexity of a control-flow graph of an obfuscated binary [201], contributing in this way to an improved analysis.

3.1.1 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 applicability. Besides, novel obfuscation techniques exploit these limitations to further hinder the analyses. Their goal is to divide the number of paths, forcing dynamic symbolic execution engines to slow down when trying to cover all the code.

3.1.2 Contributions

We propose a novel deobfuscation method based on semantic equivalence, called DoSE. The novelty of our contribution is built on the application of diffing methods 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. 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 techniques, threats also novel obfuscation techniques 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 to show how it reduces significantly the amount of code to cover. We also discuss how it can be used to combine and complete existing generic deobfuscation techniques.

We present our first contribution as follows:

- First we present background information about obfuscation techniques based on code-reuse such as range dividers, 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 3.2).
- Second, we propose our methodology of Deobfuscation based on Semantic Equivalence (*i.e.* DoSE). Formal definitions of our core methodology are given (Section 3.3) along with some improvements which make `DoSE` more efficient and more precise (Section 3.3.3).
- Third, we present concrete applications of `DoSE`, namely control-flow graph reduction (Section 3.4.1), two-way opaque predicate removal (Section 3.4.2) and cloned sub-functions detection (Section 3.4.3). Each application contains a detailed explanation of our approach and an evaluation on real-world malwares.
- Fourth, we introduce our implementation of `DoSE` as an `IDA` plug-in with with illustration of its usage (Section 3.5).
- Finally, we present a discussion on our perspectives and conclusions regarding `DoSE` (Sections 3.6.1 and 3.6.2).

3.2 Background

In this section, we will study code cloning and its combination with obfuscation techniques such as range dividers [13]. We will explain why these transformations cannot be detected by existing techniques and how `DoSE` can contribute. Then, we will present the benefits of such detection to the simplification of control-flow graphs and the removal of bogus branch functions. In the next sections, we will also focus on opaque predicates [47, 144] and more precisely on *two-way constructs* since most recent opaque predicate detection analyses and tools [17, 128] do not handle such type of constructions.

3.2.1 Range dividers

Range dividers is a novel obfuscation transformation, introduced by Banescu *et al.* [13], which exploits the limitations of generic deobfuscation techniques, such as path explosion, code coverage and complex constraints. Range dividers are input-based condition 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 3.1. 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 3.1: S. Banescu *et al.* illustration of range dividers [13]

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

3.2.2 Two-way opaque predicates

Opaque predicates [47, 128, 144] 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, Thomborson and Low 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 [211]) cannot

be analyzed.

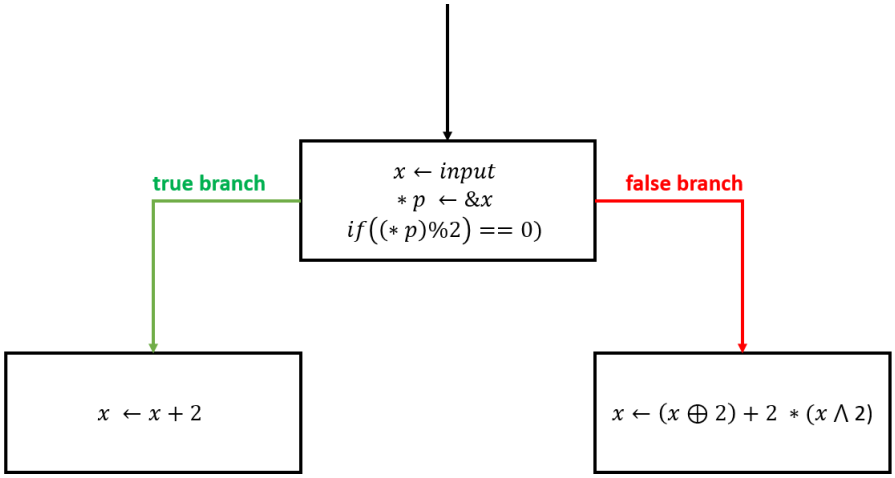

Figure 3.1: Example of a two-way opaque predicate.

Figure 3.1 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 [17, 128].

3.2.3 Binary diffing techniques

Detecting clones between binaries has a wide variety of applications such as software development [146, 194], software plagiarism detection [118, 193], vulnerabilities exploration [32, 137, 139] and malware variant detection [3, 65, 89]. Different comparison approaches have been published, either syntax-based (*i.e.* text-based) for example by measuring instruction sequences [134] or using byte n-grams [132], [99], metrics-based [65, 165] or structure-based [110, 214].

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.

3.2.3.1 Semantic-based comparison

Semantic-based comparisons 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[1] to express their inputs to outputs behavior using symbolic execution [105]. 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, Reiter and Song [74] as a static analysis in order to detect plagiarism between a set of binaries. It has since been modified, optimized [113] and extended to dynamic analysis combined with taint techniques, either to accept more noise [118], or to be more efficient [127, 130].

Our work is built on these approaches for the purpose of deobfuscating binaries. The novelty of `DoSE` comes from the transposition and a combinations of binary diffing techniques, used statically and optimized for the purpose of deobfuscation. The following section presents our methodology.

3.3 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 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 dynamic symbolic execution based deobfuscation techniques more scalable and efficient.

3.3.1 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 3.3.1. *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).

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

Definition 3.3.2. *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 3.3.3. *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. 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.

3.3.2 Semantic-based basic blocks comparison

As opposed to the syntactic approach, comparisons based on semantic equivalence rely on an intermediate representation of a basic block. It uses symbolic execution combined with a constraint solver in order to verify the equivalence between the computed expressions. 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 3.3.4. *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 $\forall x_i \in X_B, \exists! y_j \in Y_{B'}$ 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 an SMT solver.*

Definition 3.3.5. *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 3.3.6. *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.

3.3.3 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 using optimizations and conditional equivalence. The latter is presented in the followings.

3.3.3.1 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. However, the semantic approach is made 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 3.2, where the two blocks compute the same operations using different memory areas.

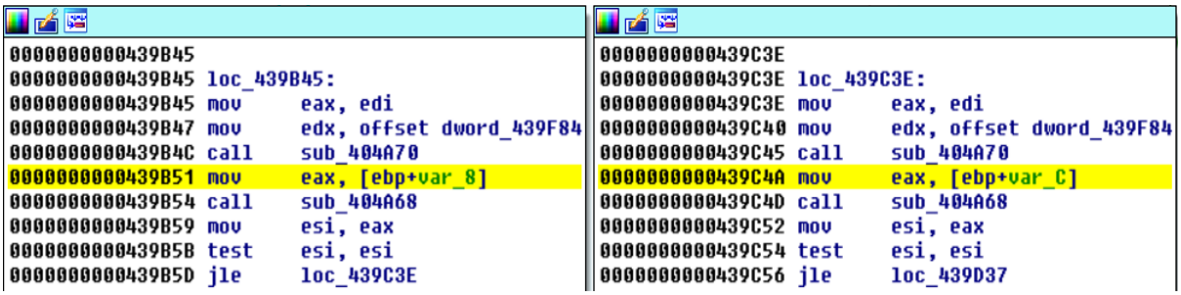

Figure 3.2: Example of two functionally equivalent basic block using different memory areas, from `Vipasana` ransomware.

We choose to treat such type of code as false positives. Statically, it is 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 [130]. Definitions of the conditional inclusion score and equivalence are similar to the semantic definitions, as presented below.

Definition 3.3.7. *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 $\mathscr{C}$ 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 $\mathscr{C}(y_j) =_{\text{cond}} \mathscr{C}(x_i)$ and set $B \subset_{\text{cond}} B'$.*

Definition 3.3.8. *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 3.3.9. *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.*

The 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 us 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 mark the blocks that are semantically but not conditionally equivalent, as undecidable. The user can further verify the equivalence when using dynamic analysis.

3.3.3.2 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. However, its efficiency may be questioned as some clones may not be detected. In order to prevent this issue, we add a normalization step 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 need to query the SMT solver. An illustration of the syntactic normalization is given in Figure 3.3.

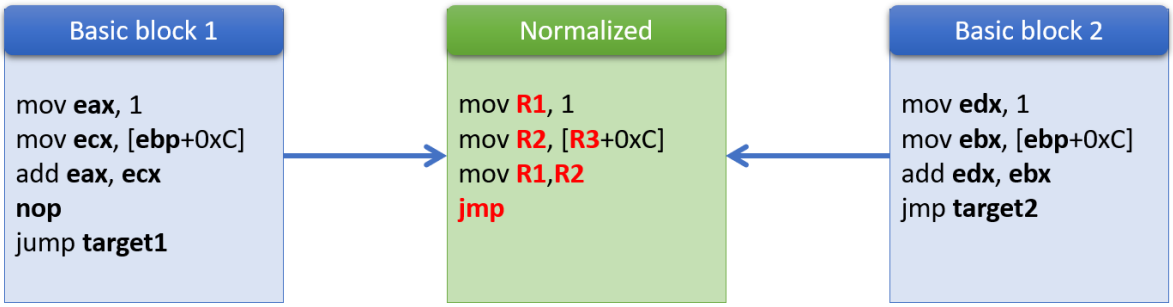

Figure 3.3: Normalization of the syntax of two blocks which are cloned.

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 3.1 illustrates the differences in execution time and false negatives and positives results of our method, before and after our improvements.

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 3.1: Differences of false positives, false negatives results and execution time before and after our improvements, based on control-flow graph reduction of several malware functions.

As we can see, our improvements eliminate most of the false positives and false negatives results. Moreover, it allows DoSE to run significantly faster, *e.g.* from 1851 seconds to 243 seconds on function 0x1000be35 of the Asprox malware. Thus, 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.

3.4 Applications

In this chapter 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 and limitations. DoSE is implemented as an IDA Pro [86] plug-in, based on the reverse-engineering framework Miasm [57] 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.

The next section start by introducing the main application of DoSE for the reduction of control-flow graphs.

3.4.1 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 techniques 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.

3.4.1.1 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 (and which has been previously simplified in order to merge basic blocks, cf. Section 3.3.1) 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).

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
```

Basic blocks comparisons. After the collection step, 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 verifications, we proceed to the syntactic comparison. If the syntactic inclusion score is 1 for B

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 σ_syn(B, B') = σ_syn(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_sem(B, B') = σ_sem(B', B) = 1 then
10:                    if σ_cond(B, B') = σ_cond(B', B) = 1 then
11:                        C[B] ← B'
12:                        C[B'] ← 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
```

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 3.3.

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.

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 3.2: Evaluation of static control-flow graph reduction using DoSE

3.4.1.2 Evaluations

To illustrate the efficiency of our analysis, we used several malware samples[2] among `Flame` [24] and `Cryptowall` [206] as shown in Table 3.2. We analyzed some functions of these samples with their entry-points listed in column "Function EP". These functions have been selected for their large sizes in order to measure the scalability of DoSE. Column "# Nodes" indicates the numbers of nodes 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 false negative results and also the execution time of the analysis. For each application of `DoSE`,

[2]Samples are available at `https://github.com/lamaram/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 3.2,

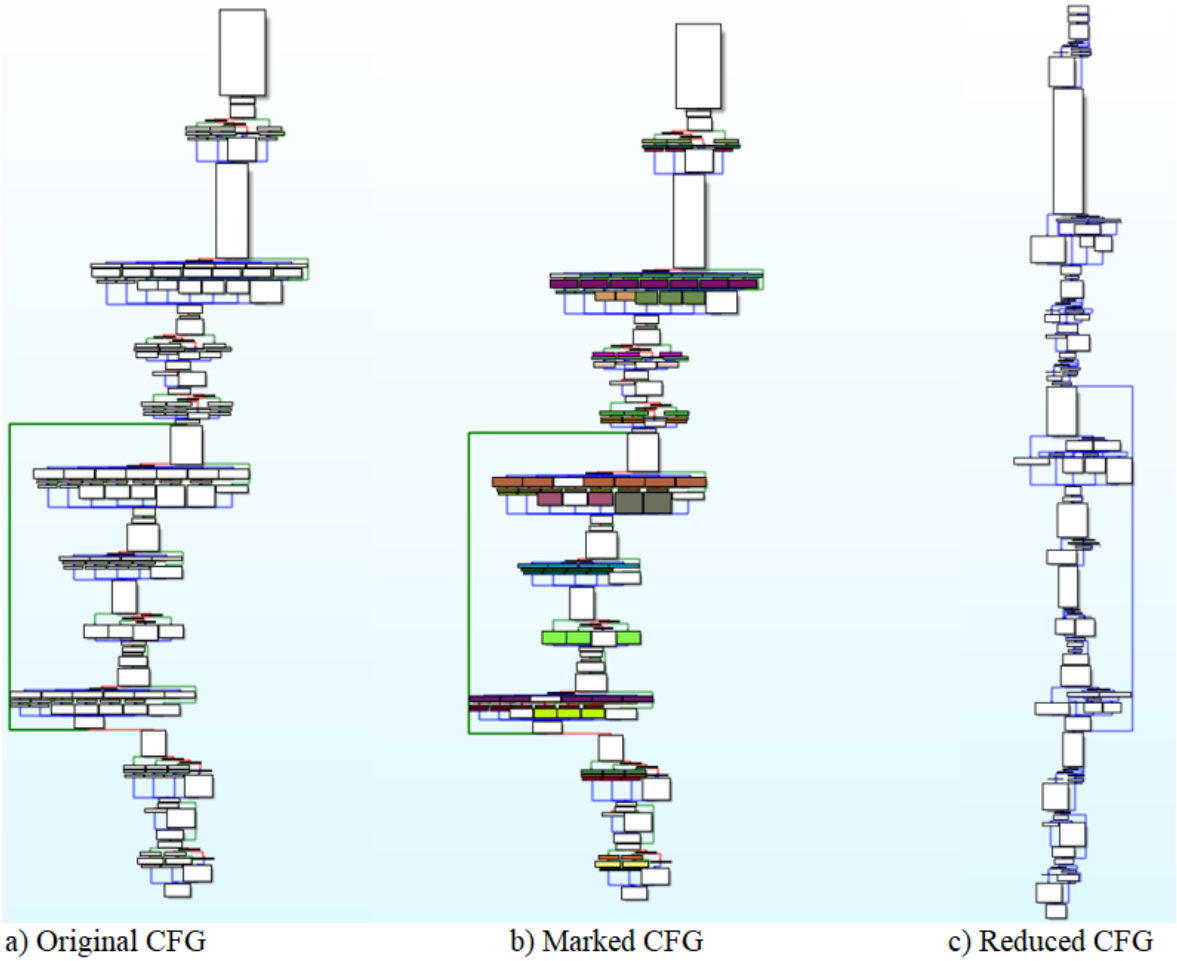

Figure 3.4: Example of `CryptoWall` main function control-flow graph reduction.

`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 62.6% of a function's control-flow graph. Figure 3.4 illustrates the application of `DoSE` on `Cryptowall` main function. The tagged control-flow graph (*i.e.* 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. Another example of DoSE applications to the Vipasana ransomware is given in Figure 3.5.

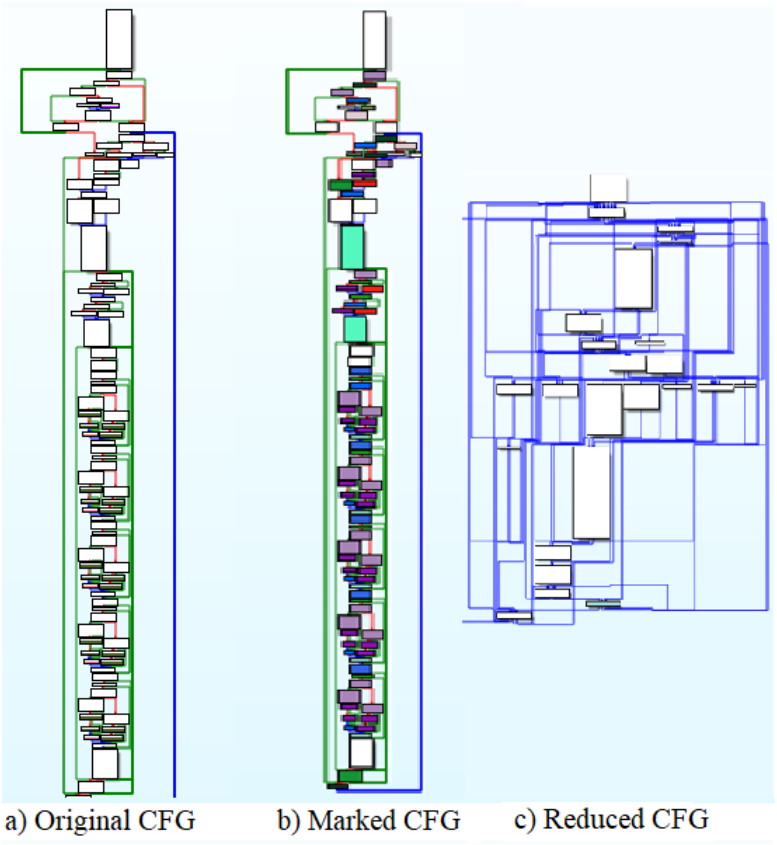

Figure 3.5: Example of a Vipasana function control-flow graph reduction.

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

3.4.2 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.).

3.4.2.1 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 $\mathscr{B}$, such that $\phi_n \in \mathscr{B}$, $n \in \mathbb{N}$. 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 $\mathscr{B}$. 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[3]. 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,

[3] A returning block refers to a basic block that exits a function.

but also for inclusions (cf. Section 3.3). For these purposes, let us note $\mathscr{S}_{\text{eq}}$ the equivalence score between two given ω_n^T and ω_n^F, and let us note $\mathscr{S}_{\text{inc}}$ their inclusion score. $\mathscr{S}_{\text{eq}}$ and $\mathscr{S}_{\text{inc}}$ represent the amount of coupled basic blocks that are equivalent and included respectively. Moreover, we define a total score $\mathscr{S}_{\text{tot}}$ such that:

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

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

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

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 $m \in [1, \texttt{min}(\#\omega_n^T, \#\omega_n^F)]$.

Then, three cases could occur:

- B_m and B'_m are syntactically equivalent; then we increment the score $\mathscr{S}_{\text{eq}}$.
- B_m and B'_m are semantically and conditionally equivalent; then we increment the score $\mathscr{S}_{\text{eq}}$.
- B_m is semantically and conditionally included (but not equivalent) to B'_m; then we increment the score $\mathscr{S}_{\text{inc}}$ (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 $\mathscr{S}_{\text{eq}} \geqslant \mathscr{S}_{\text{inc}}$ and $= \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ then we mark the predicate as a **two-way** opaque construct.
- if $\mathscr{S}_{\text{eq}} < \mathscr{S}_{\text{inc}}$ and $= \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ 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 $< \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ and > 0 then we mark the predicate as **normal** and we propose to group equivalent basic blocks to reduce the control-flow graph.

3.4.2.2 Evaluations

For the evaluation, we used the `Tigress` obfuscator [43] which implements these opaque predicates[4]. 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

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

Algorithm 4 Paths equivalence checking

procedure PATHS EQUIVALENCE CHECKING(ω_n^T: true path, ω_n^F: false path)
  $\mathscr{S}_{\text{eq}}, \mathscr{S}_{\text{inc}}, \mathscr{S}_{\text{tot}} = 0, 0, 0$
  for each basic blocks B, B' in ω_n^T, ω_n^F **do**
    if $\sigma_{\text{syn}}(B, B') = \sigma_{\text{syn}}(B', B) = 1$ **then**
      $\mathscr{S}_{\text{eq}} + +$
    else if $\sigma_{\text{sem}}(B, B') = \sigma_{\text{sem}}(B', B) = 1$ **then**
      if $\sigma_{\text{cond}}(B, B') = \sigma_{\text{cond}}(B', B) = 1$ **then**
        $\mathscr{S}_{\text{eq}} + +$
      end if
    else if $B \subset_{\text{sem}} B'$ or $B' \subset_{\text{sem}} B$ **then**
      if $\sigma_{\text{cond}}(B, B') = 1$ or $\sigma_{\text{cond}}(B', B) = 1$ **then**
        $\mathscr{S}_{\text{inc}} + +$
      end if
    end if
  end for
  $\mathscr{S}_{\text{tot}} = \mathscr{S}_{\text{eq}} + \mathscr{S}_{\text{inc}}$
  if $\mathscr{S}_{\text{eq}} \geqslant \mathscr{S}_{\text{inc}}$ and $= \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ **then**
    return **two-way**
  else if $\mathscr{S}_{\text{eq}} < \mathscr{S}_{\text{inc}}$ and $= \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ **then**
    return **probable**
  else if $< \texttt{max}(\#\omega_n^T, \#\omega_n^F)$ and > 0 **then**
    propose `Control-flow graph reduction()` // see Algorithm 1
  end if
  return **normal**
end procedure

opaque predicates constructs. We combined them with other obfuscation techniques implemented in `Tigress`, such as control-flow flattening (*i.e. Flat*), encodings of respectively data (*i.e. EncD*), arithmetics (*i.e. EncA*) and literals (*i.e. EncL*) and finally code virtualization (*i.e. 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

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 3.3: Evaluation on the generated use cases with `Tigress`.

these limitations in the following paragraph.

We also evaluated our implementation against real world malwares. Table 3.4 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 3 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.

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

3.4.2.3 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. 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.

3.4.3 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.

3.4.3.1 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 3.4.1. *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.

3.4.3.2 Evaluations

We evaluated the detection of statically equivalent sub-functions against known malwares as illustrated in Table 3.5. Column 3 represents the number of functions before our analysis whereas column 4 illustrates the number of detected cloned sub-functions.

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 3.5: Evaluation of sub-functions detection

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 or 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.

3.4.3.3 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.

3.5 Implementation

In this section we introduce our implementation of `DoSE` as an `IDA` plug-in. We start by presenting our tool for each applications previously introduced (see Section 3.4), namely reducing control-flow graphs, detecting two-way opaque predicates and detecting cloned sub-functions. Finally, we will describe the different libraries used to develop our plug-in.

3.5.1 DoSE plug-in for control-flow graph reduction

DoSE plug-in permits users to select different applications, directly within IDA. Once the plug-in is running, the application menu pops-up as illustrated in Figure 3.6. In this section we present the

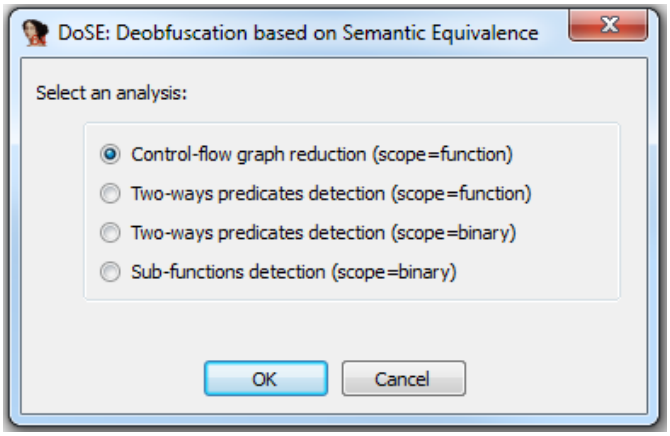

Figure 3.6: Start-up menu for application selection of DoSE IDA plug-in.

first application for control-flow graph reduction. Such application can be tweaked by the user in order to fit his requirements. The next pop-up window asks these information in order to run the desired analyses. As we can see in Figure 3.7, the user can select several options:

- **The scope of the analysis**: the plug-in propose either to work on the complete function in order to detect all cloned basic blocks, or to select two basic-blocks to compare.
- **The analyses**: these options are mostly made for debugging. The user can then compare DoSE methodology with only syntax-based, semantic-based or conditional-based equivalence.
- **The conditional equivalence bound**: this options sets the amount of time two conditionally-equivalent expressions have to match before ruling them as equivalent.
- **The optional display features**: the user can select either to rebuild a new control-flow graph without the detected clones, and also to color each equivalent basic-blocks with the same color.
- **The debugging options**: these options are mainly present to set up the verbosity of the plug-in, as well as comparing the methodologies with and without memory randomization during the equivalence checks.

Once the application is running, it will detect each cloned basic-blocks as described in DoSE methodology (see Section 3.3). An example of such application's output is illustrated in Figure 3.8.

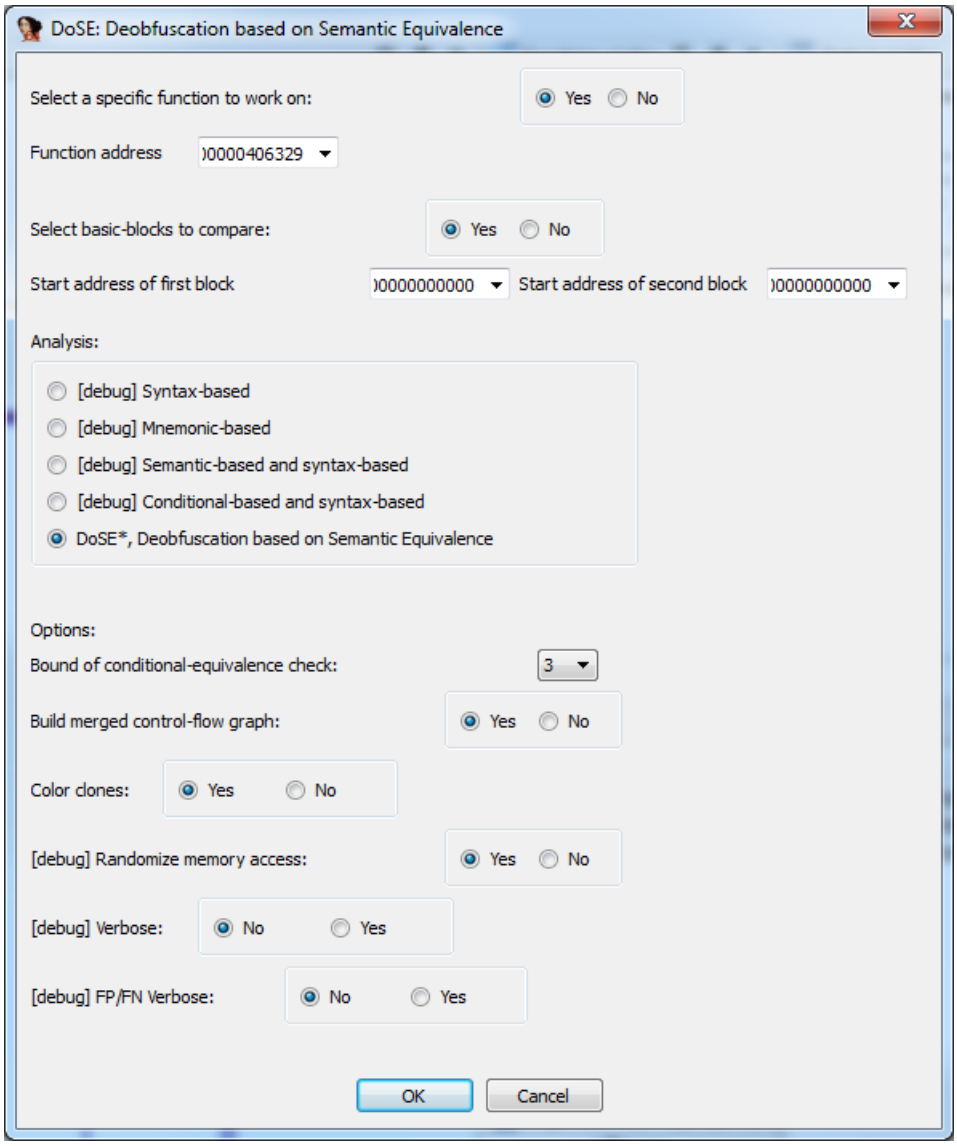

Figure 3.7: Options menu for DoSE plug-in applications to control-flow graph reduction.

We can see that DoSE displays on the original control-flow graph each equivalent basic-blocks with the same color. Moreover, it recreates a new control-flow graph without any clones. An optional pop-up window is displayed to inform the user about the amount of basic-blocks before and after the analysis.

3.5.2 DoSE plug-in for two-way opaque predicate detection

In order to detect two-way opaque predicates, the plug-in first start by collect each predicates. In this case, the user can choose either to specify a function, or to work on the whole binary. Each collected predicate is then displayed in an informative window, as illustrated in Figure 3.9. The window will

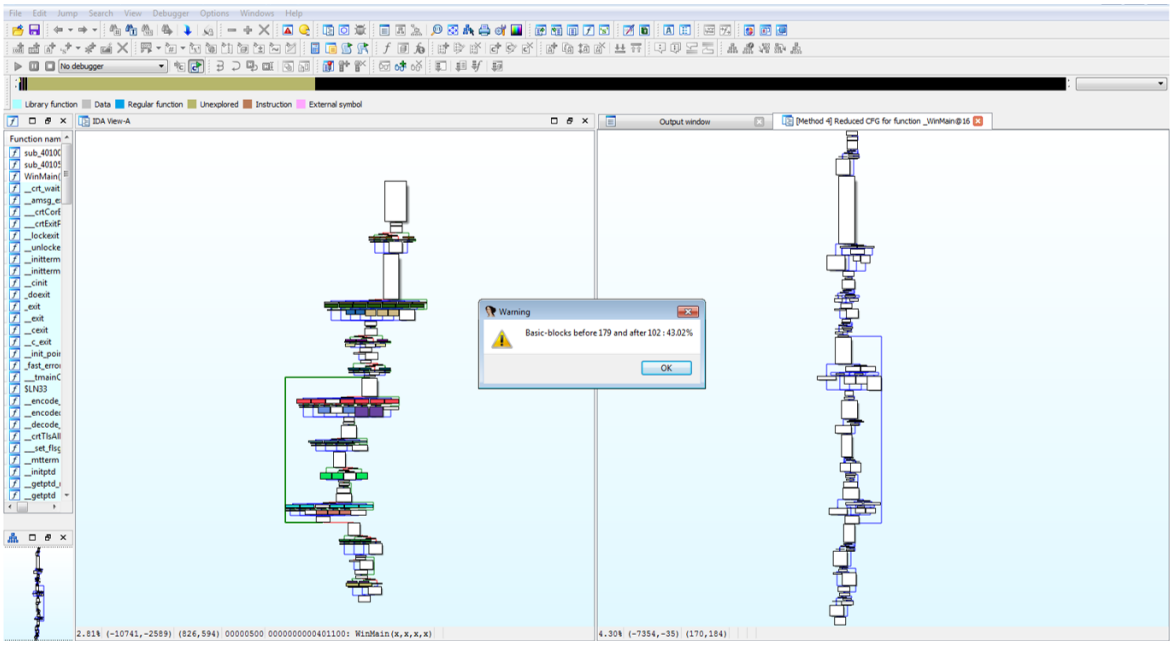

Figure 3.8: Example of DoSE output for control-flow graph reduction on the CryptoWall ransomware.

also display, after analysis, each equivalence score (namely syntactic, semantic, conditional, and all combined) in order to rule the predicate as either two-way *opaque* or *normal*. Afterwards, the user can right-click on any predicate in order to select from the following options:

- *Jump to predicate*: the user can jump to the disassembled code of the predicate.
- *Color successors*: the user can color all paths following the predicate.
- *Check selected predicate*: the user can check if the selected predicate is *opaque* or not.
- *Check all predicates*: the user can check all collected predicates.

Note that other options are from IDA. Finally, once all predicates are analyzed by the plug-in, all results are directly displayed within the predicates window as shown in Figure 3.11. As we can see, all scores are added for each analyzed predicates. Moreover, the user can visualize the disassembled code of a selected predicates, in our example an opaque one. This helps the user better understand the underlying construction of the detected obfuscation transformation.

3.5.3 DoSE plug-in for cloned sub-functions detection

The final application of DoSE plug-in consists in detecting cloned sub-functions within the whole targeted binary. To that end, the plug-in simply analyses each functions, and renames any cloned one so that the user can see them directly in the functions windows of IDA. Figure 3.12 illustrates the output of such application within IDA.

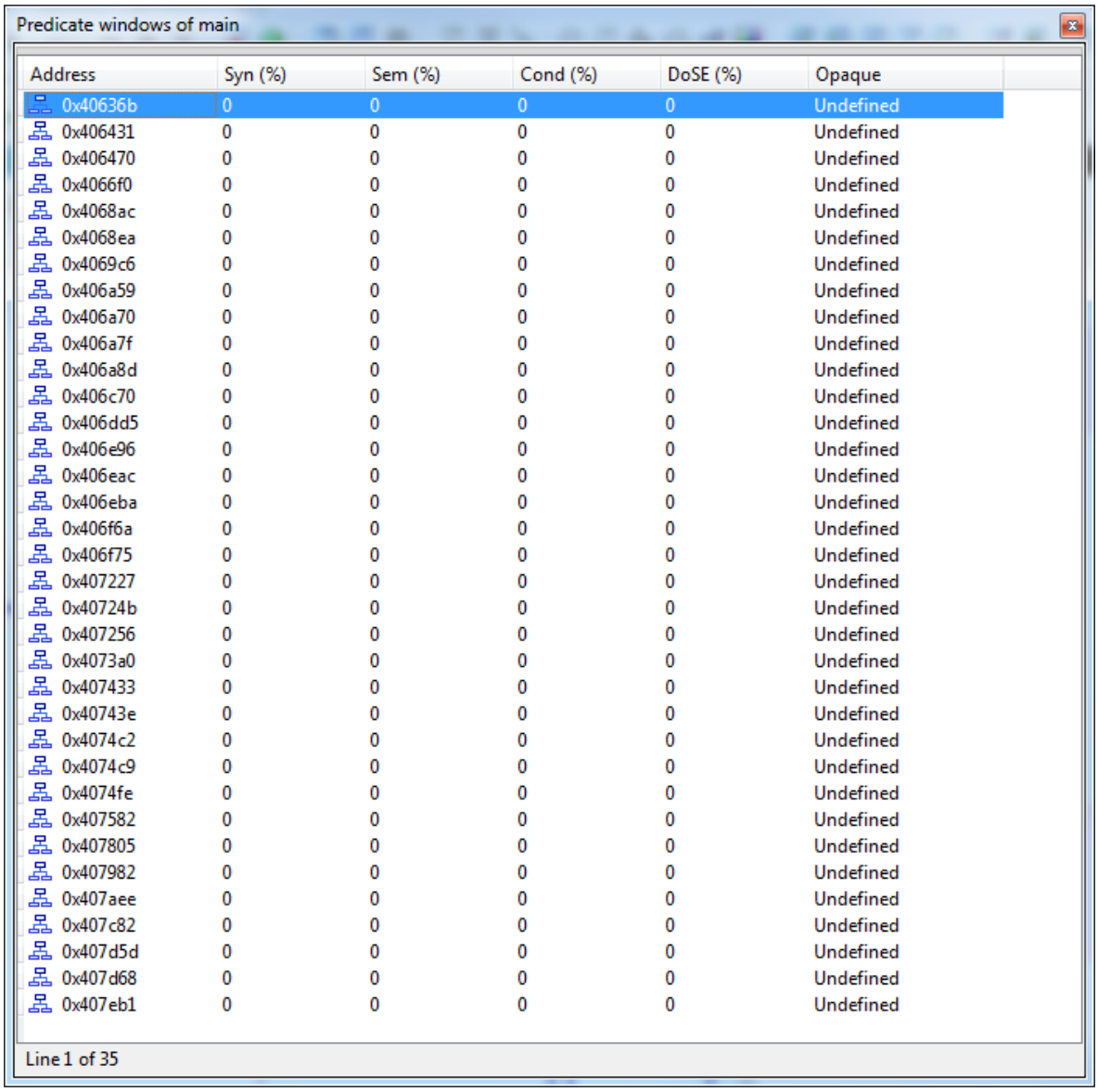

Figure 3.9: DoSE plug-in window of collected predicates.

3.5.4 Development

In order to develop our methodology as a plug-in for IDA, we choose the following API:

- Miasm2: the symbolic execution engine as well as Miasm intermediate representation are used to work directly on the semantics of the analyzed code.
- IDA Python: the IDA Python API is required to build our tool as an integrated plug-in for IDA.

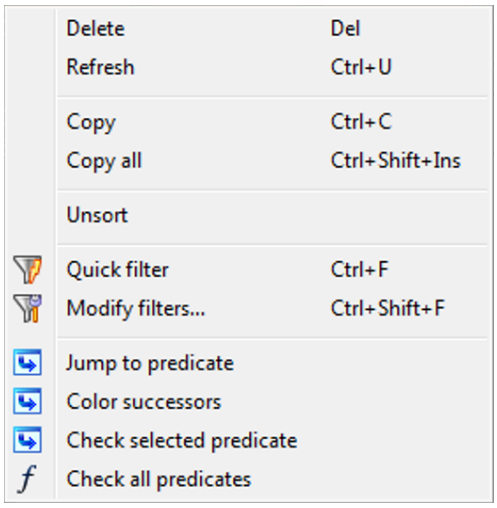

Figure 3.10: DoSE plug-in options on collected predicates.

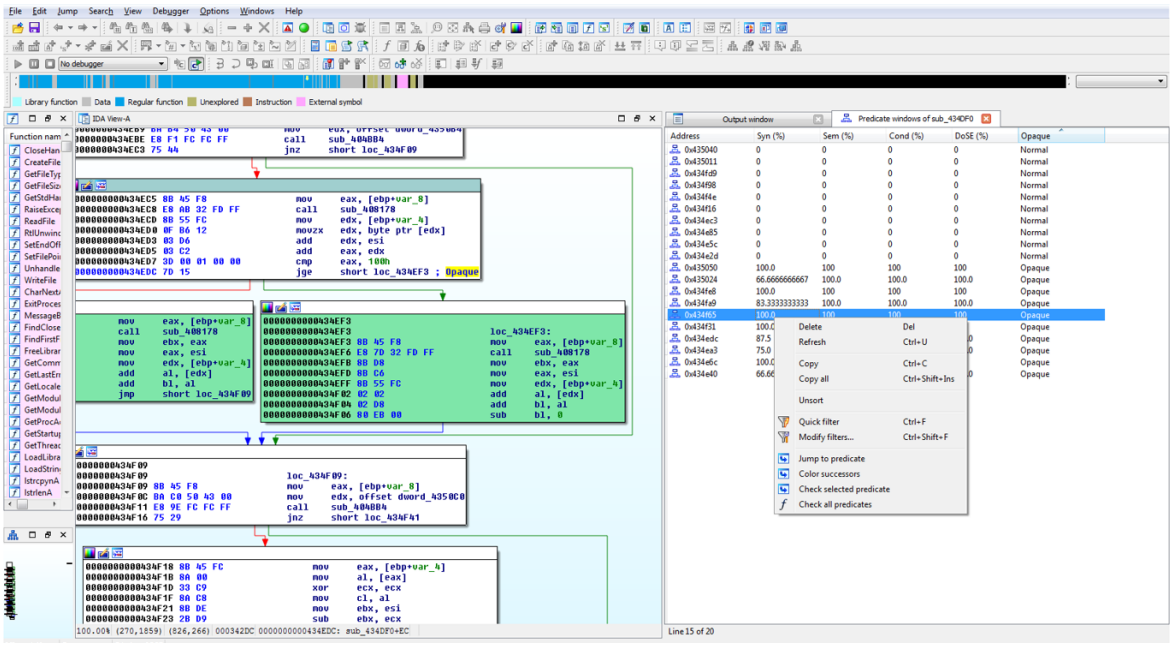

Figure 3.11: Example of DoSE output for two-way opaque predicate detection on the Vipasana ransomware.

3.6 Conclusions

In this section we present our concluding remarks and perspective about our methodology, and tool, for deobfuscation based on semantic equivalence.

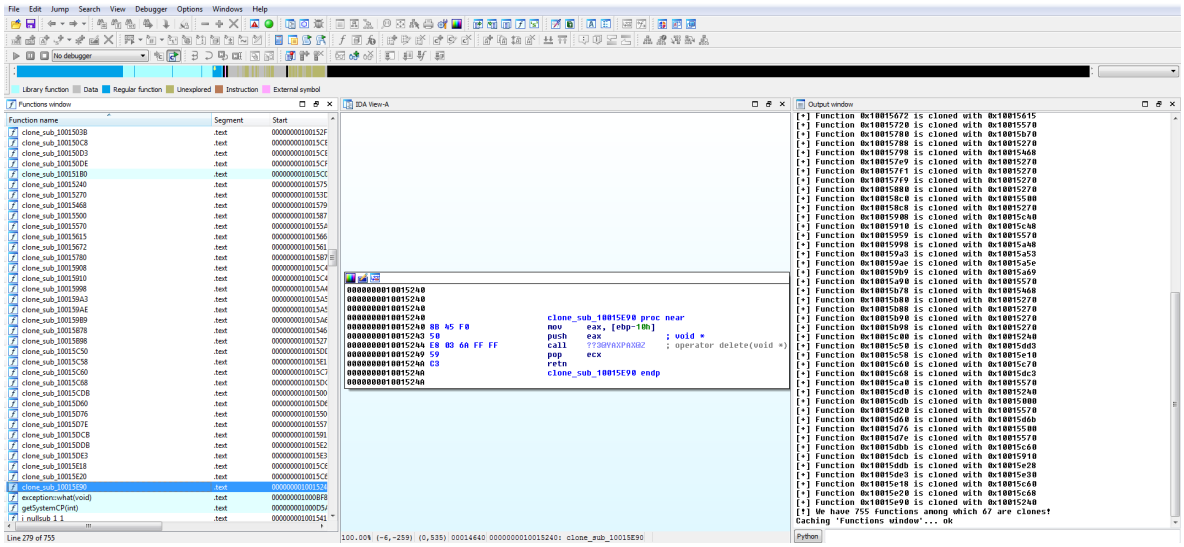

Figure 3.12: Example of DoSE output for sub-functions detection on the OnionDuke malware.

3.6.1 Perspectives

The following paragraphs present our perspectives as future work for DoSE.

3.6.1.1 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.

3.6.1.2 Hybrid analysis

Even if our current approach is evaluated statically, it is straightforward to use it dynamically through DSE. Using our approach dynamically would prevent limitations due to emulation of memory access since their concrete values are available at runtime. 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, and preserving DoSE scalability to real-world use-cases.

3.6.2 Conclusion

Obfuscated software 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 62%, 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.

Chapter 4

Defeating Opaque Predicate using Binary Analysis and Machine Learning

Contents

In this chapter, we present our second contribution consisting in a new approach that bridges binary analysis techniques with machine learning classification. Our goal is to provide a static and generic evaluation technique for opaque predicates, regardless of their constructions. We use this technique as a *static automated deobfuscation tool* to remove the opaque predicates introduced by obfuscation mechanisms. According to our experimental results, our models have up to 98% accuracy at detecting and deobfuscating state-of-the-art opaque predicates patterns. By contrast, the leading edge deobfuscation methods based on symbolic execution show less accuracy mostly due to the SMT solvers constraints and the lack of scalability of dynamic symbolic analyses. Our approach underlines the efficiency of hybrid symbolic analysis and machine learning techniques for a static and generic deobfuscation methodology.

4.1 Introduction

As introduced in Chapter 2, opaque predicates [47] are widely used as technique for various security applications, *e.g.* metamorphic malware mutation [33], Android applications [109] or white-box cryptographic implementations. As a consequence, several works focus on the deobfuscation of opaque predicates (*e.g.* [17, 21, 23, 63, 129, 150, 185]) in order to evaluate the quality of the obfuscated code rendered by this transformation. However, these techniques are often based on dynamic analysis and are therefore limited or not scalable.

4.1.1 Problem setting

Existing state-of-the-art opaque predicates deobfuscation techniques and tools suffer from the following limitations:

1. *Specificity:* Techniques that evaluate opaque predicates are often focused on specific constructions, hence lacking of generality towards all existing patterns of such obfuscation transformation.

2. *Code coverage:* Most recent deobfuscation techniques are based on dynamic symbolic execution which require the generation of instruction traces. The ability to cover all paths of the program is an issue that prevents, in some cases, the complete code deobfuscation.

3. *Scalability:* Dynamic symbolic execution techniques may also lack of scalability on some types of code such as malwares that use specific triggers (*e.g.* an input from a Command and Control server) to execute the non-benign part of the code. Thus, dynamic analysis may not scale and cover the non-triggered part of the code.

4. *Satisfiability modulo theories solvers:* SMT solvers used in path-reachability analyses suffer from several limitations depending on the constructions of the opaques predicates. Some constructions that are based on aliases or mixed boolean and arithmetic expressions usually cause SMT solvers to fail at predicting the feasibility of a path.

Our work has the goal to re-introduce static analysis for obfuscated software evaluation and deobfuscation. To this end, we propose a new approach that bridges static symbolic execution and machine learning models to provide a generic evaluation of opaque predicates.

We present several studies and experiments towards the construction of machine learning models that can either detect an opaque predicate or predict its invariant value without executing the code. We also extend our design to the deobfuscation of such obfuscation transforms, regardless

of their constructions, by creating a static analysis plug-in within a widely used reverse engineering tool called `IDA` [86]. To further evaluate our methodology, we compare it against available static and dynamic symbolic-based tools for the deobfuscation of opaque predicates. We conduct further evaluations against obfuscators such as `Tigress` [43] and `OLLVM` [96].

The aftermath of this contribution shows that combining machine learning techniques with static symbolic analysis provides a generic, automatic, and accurate methodology towards the evaluation of opaque predicates. Our work shows that machine learning enables a better efficiency and genericity for this application, while allowing us to work without SMT solvers to predict reachable paths.

4.1.2 Contributions

In order to face the above listed limitations, we provide the following contributions:

1. We present our novel methodology that binds binary analysis and machine learning techniques to evaluate and deobfuscate opaque predicates statically. A presentation of several studies towards an efficient and accurate creation of machine learning models is also given.
2. The evaluation of our methodology against state-of-the-art obfuscators such as `Tigress` and `OLLVM`, as well as novel opaque predicate constructions such as *bi-opaque* predicates.
3. The illustration of the efficiency of our methodology, used as a static analysis deobfuscation tool, on several datasets by comparing it to existing state-of-the-art deobfuscation tools based on symbolic execution and SMT solvers.

Our contribution is organized as follows: in Section 4.2 we recall background information on opaque predicates types, constructions, and deobfuscation methods. Then we introduce some notions of supervised machine learning. In Section 4.3, we present our methodology which combines binary analysis and machine learning to achieve an efficient evaluation and deobfuscation of opaque predicates. Section 4.4 presents our experiments towards an accurate model construction, whereas Section 4.5 illustrates our evaluations on state-of-the-art publicly available obfuscators. A comparison to existing symbolic-based deobfuscation techniques against our methodology is also provided in Section 4.5.3. We then describe our design limitations and perspectives in Section 4.6, along with related work in Section 4.7.

4.2 Background

Several types and constructions of opaque predicates exist [128]. The following paragraphs give an overview of them. Collberg, Thomborson and Low defined these predicates by, respectively,

P^T, P^F and $P^?$ opaque predicates. Several works use these two-ways opaque predicates constructs, either referred to as range-dividers [13], or as correlated opaque predicates [129, 199]. Moreover, regardless of their output, *e.g.* their type, there exists many different kinds of construction in order to render these predicates opaque.

4.2.1 Opaque predicate constructions

Apart from their types, opaque predicates are defined by their constructions. Several proposals exist in the literature about how to construct a resilient and stealthy opaque predicate [9, 47, 133, 199, 200], as presented in [55]. Each of these constructions have for purposes to thwart specific deobfuscation analyses, *e.g.* static or symbolic, as they will be summarized in Section 4.5.3.

4.2.1.1 Arithmetic-based

Opaque predicates can be constructed using mathematical formulas which are hard to solve. They aim at encoding invariants into arithmetic properties on numbers. Listing 1 shows an opaque predicate generate by `OLLVM` [96]. This opaque predicate is the encoding of the $7y^2 - 1 \neq x^2$ predicate, in `x86` language.

```
1  mov   eax , ds:x
2  mov   ecx , ds:y
3  imul  ecx , ecx
4  imul  ecx , 7
5  sub   ecx , 1
6  imul  eax , eax
7  cmp   ecx , eax
8  jz    <bogus_addr>
```

Listing 4.1: Example of an arithmetic-based opaque predicate generated by `OLLVM`.

The purpose of this construction is to hide the invariant property of such predicates, however, they are not resilient to static symbolic attacks based on abstract interpretation [150] or SMT solvers [17, 129].

4.2.1.2 Mixed-boolean and arithmetic based

Introduced by Zhou *et al.* [211], Mixed Boolean-Arithmetic (*i.e.* MBA) consists in a data encoding technique based on linear identities involving boolean and arithmetic operations, together with invertible polynomial functions. The resulting encoding is made dependent on external inputs such that it cannot be deobfuscated using compiler optimization techniques. Listing 2 shows an example

of the `Tigress` obfuscator [43] encoding option on the simple expression $x+y+z$. `Tigress` can generate linear MBA expressions of several layers, increasing their complexity, which makes these expressions hard to simplify symbolically or using SMT solvers.

```
# Normal expression
x + y + z

# MBA-based obfuscated expression
(((x ^ y) + ((x & y) << 1)) | z) + (((x ^ y) + ((x & y) << 1)) & z)
```

Listing 4.2: Example of an MBA-based opaque predicate generated by `Tigress`.

4.2.1.3 Alias-based

Aliasing is represented by a state of a program where certain memory location is referenced to by multiple symbols, *e.g.* variables, in the program. Pointer alias analysis is undecidable, thus using them for opaque predicate constructs in a pertinent choice. Collberg, Thomborson and Low [47] first choose to use this undecidability result to build opaque predicates using pointers in linked lists or arrays. Listing 3 shows an example of a pointer aliasing invariant opaque predicate.

```
void foo () {
  item * pointer_1 = create_circular_list(rand());
  pointer_1 = move (pointer_1, rand());
  item * pointer_2 = insert(pointer_1);
  int random = rand();
  pointer_1 = move(pointer_1, random);
  pointer_2 = move(pointer_2, random);
  if (pointer_2 == pointer_1) {
    // never taken
  }
}
```

Listing 4.3: Pointer aliasing invariant opaque predicate

In this example, the function `create_circular_list` creates a list of a random size and returns a pointer to it. Then the `move` function shifts the current pointer. Finally, the `insert` function adds a new item in the list and returns a pointer to it. Thus, variable `pointer_1` cannot alias with variable `pointer_2`, which makes this an alias-based invariant opaque predicate.

4.2.1.4 Environment-based

Environment-based opaque predicate consists in using invariant from the system, or libraries, to construct opaque predicates. Listing 4 illustrate an example of an environment-based opaque predicates using the `strcpy` library function's output.

```
1  char s1 [4] = "Foo";
2  char s2 [11] = "Helloworld";
3
4  if(strcpy(s2 ,s1) == s2) {
5    // always taken as strcpy returns s2
6  }
```

Listing 4.4: Environment-based invariant opaque predicate

4.2.1.5 Concurrence-based

Concurrence-based opaque predicates are encoded using race-condition properties. Thus, both static and dynamic analyses are known to be difficult and unreliable for proving properties on such concurrent code, which makes this construction of opaque predicates efficient although difficult to make reliably.

4.2.1.6 Bi-opaque

Recent work introduces opaque predicates constructs that aim at employing the weaknesses of symbolic execution to compose them such that they can evade detection from symbolic execution-based adversaries [200]. Based on the observation that deobfuscation techniques using symbolic execution are prone to some challenges and limitations, Xu *et al.* introduced bi-opaque predicates which intend to either introduce false negatives or false positives results. Therefore, bi-opaque predicates are based on techniques such as symbolic memory or floating point instructions in order to exploit current deobfuscation methodologies and tools. Such construction has been shown effective against state-of-the-art deobfuscation tools based on dynamic symbolic execution, such as `Triton` [1] or `Angr` [175].

4.2.2 Deobfuscation

Because of their wide utilization and popularity, opaque predicates are targets of several published attacks. Each of these deobfuscation methodologies have their strengths and limitations, as it will be synthesized in the following paragraphs.

4.2.2.1 Probabilistic check

Since using brute-force to check all possible inputs $x \in X$ such that a predicate $\mathcal{O}(\phi)(x) = \{0,1\}$ is time consuming (depending on the size of x), a methodology to deobfuscate opaque predicate proposes to randomly choose x and execute the program segment to compute $\mathcal{O}(\phi)(x)$ for all these values. If the output is always the same, then one might suspect that $\mathcal{O}(\phi)$ is an invariant predicate. Such technique can be practiced with fuzzing on the inputs. However, this approach is prone to high false positives/negatives results when the opaque predicates are not input-dependent (*e.g.*, environment-based) or correlated (*i.e.* two-ways type).

4.2.2.2 Pattern matching

Due to the overhead introduced by most complex opaque predicates constructs, it has been showed in the literature that there are surprisingly relatively few predicates that are used over and over again. This leads to a possible pattern matching attack (otherwise called dictionary attack) [63], where one takes obfuscated predicates from a program being attacked and pattern-matches the source code against known examples. However, it is possible to build variants of opaque predicates that cannot be matched using dictionary attacks, which implies a high false negative rate.

4.2.2.3 Abstract interpretation

First proposed by Dalla Preda *et al.* [150], abstract interpretation for opaque predicate deobfuscation is static and semantic-based attack. Since dynamic or hybrid attacks may be either not precise or time consuming, their work provides correctness and efficiency in the deobfuscation of certain constructions of opaques predicates. Indeed, this technique can only be efficient against some classes of invariant arithmetic-based opaque predicates, and do not focuses on other types and constructions.

4.2.2.4 Automated proving

Current state-of-the-art deobfuscation approach use SMT-solvers to compute if a predicate is constant [17, 128], *i.e.* opaque. Udupa, Debray and Madou [187] use static path feasibility analysis based on these SMT solvers to determine whether an execution path is feasible. However, their methodology is prone to the limitations of static analysis, that is why recent automated proving techniques are based on dynamic analysis, *e.g.* instructions traces, to check path feasibility and thus to detect and deobfuscate opaque predicates. Sometimes combined with taint analysis, it allows to capture only instructions or semantics related to the targeted predicate. However, since these techniques target only the question of feasibility, Bardin, David and Marion [17] uses bounded-backward analysis to target infeasibility questions. Yet, automated proving based analyses,

either static or dynamic, may suffer from symbolic execution limitations as well as SMT solvers restraints. Indeed, it has been showed that SMT solvers fail against MBA opaque predicates, whereas symbolic execution can be slowed down effectively, or even misguided. For example, alias-based constructions or more recent opaque predicates constructions such as the bi-opaque ones.

4.2.2.5 Program synthesis

Recently introduced by T.Blazytko *et al.* [23], program synthesis aims at synthesizing code of arbitrary complexity. Their approach is based on execution traces which are simplified, and from which the semantics are extracted and then 'learned' by a synthesis module. Originally made for the deobfuscation of virtualized code, their approach has been successful for the simplification of Mixed Boolean and Arithmetic expressions. However, their work is not focused on opaque predicates and thus do not cover all types and constructs as presented in the previous paragraphs.

Table 4.1 summarizes the strengths and targets of existing researches regarding the evaluation and deobfuscation of opaque predicates, in terms of their constructions. As we can see, automated proving, often based on dynamic symbolic execution (abbreviated *DSE*), is the most effective methodology against opaque predicates. However, the evaluation of such techniques has been shown effective mainly against arithmetic or environment based opaque predicates, hence the importance of a generic methodology that can help evaluate both their stealth and resiliency, while covering all existing constructions.

Overall, dynamic symbolic execution is currently considered the most effective methodology against opaque predicates, but the evaluation of such technique has been shown effective mainly against arithmetic or environment based opaque predicates. This demonstrates the importance of a generic and scalable methodology that can evaluate both stealth and resilience of opaque predicates for all existing constructions.

4.2.3 Supervised Machine Learning

The decision of labeling a predicate as opaque, and even more as invariant P^T or P^F opaque predicate, can be considered as classifications problems. Our target is to find algorithms that work from external supplied instances (*e.g.*, binaries, instructions traces) in order to produce general hypotheses. From these hypotheses, we want to make predictions about future instances. *Supervised machine learning* provides a dedicated methodology that achieves this goal. The aim of supervised machine learning is to build a *classification model* which will be used to assign *labels* to testing

Constructions	Probabilistic check	Pattern matching	Abstract interpretation	Automated proving	Program synthesis
Arithmetic-based	✓[187] High FN/FP	✗ High FN	✓[150]	✓[17, 129]	✗
MBA-based	✗ High FN/FP	✓[63]	✗	✗ (limitations of SMT solver)	✓[21, 23]
Alias-based	✗ High FN/FP	✗	✗	✗ (limitations of symbolic execution)	✗
Concurrence-based	✗ High FN/FP	✗	✗	✗ (limitations of symbolic execution)	✗
Environment-based	✗ High FN/FP	✗	✗	✓[17, 129]	✗
Bi-opaque	✗ High FN/FP	✗	✗	✗ High FN/FP	✗

Table 4.1: Illustrations of opaque predicates deobfuscation strengths and targets against known constructions and types.

or unknown instances. In other words, let X be our inputs (*i.e.* instances) and Y the outputs (*i.e.* predicted labels). A supervised machine learning algorithm will be used to learn the mapping function f such that $Y = f(X)$. The goal is to approximate f such that for any new instance X we can predict its label Y.

Figure 4.1 provides a generic overview of a supervised machine learning classification scheme. In our case the inputs are represented by n-dimensional vectors of numerical features that represent these features, *i.e. features vectors*, for which the extraction is described in the following paragraph.

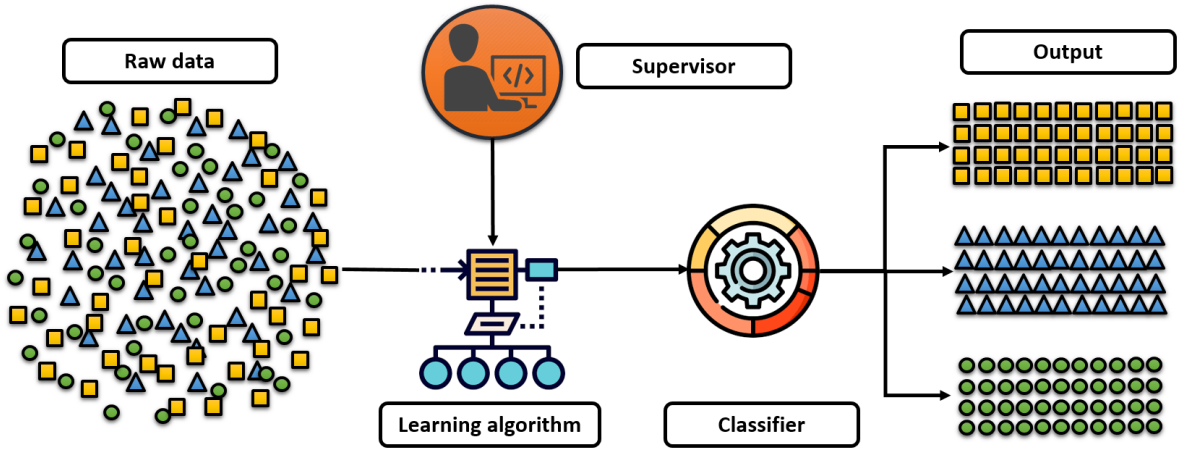

Figure 4.1: Generic overview of a supervised machine learning classification scheme.

4.2.3.1 Feature extraction.

In the machine learning terminology, the inputs of a model are usually derived from what is called *raw data*, *i.e.* the data samples we want to classify or predict. These data samples cannot be directly

given to a classification model and need to be processed beforehand. This processing step is called *feature extraction* and consists in combining the raw data variables into numerical features. It allows to effectively reduce the amount of data that must be processed, while accurately describing the original dataset of raw data. In our case, since raw data are text documents (*e.g.* disassembly code, symbolic execution state, etc.), one practical use of feature extraction consists in extracting the *words* (*i.e.* the features) from the raw data and classify them by frequency of use (*i.e.* weights). Different approaches exist for understanding what a word is and to compute its weight. In this chapter we use the *bag of words* approach which identifies terms with words. As for the weights, we studied *term frequency* (*i.e.* how frequently a word occurs in a document) with and without *inverse document frequency* [95] used in Section 4.4 in order to select the best possible extraction technique. In other words, the approach we use consists in several steps. First, a vocabulary of known words is created. Second, we measure the presence (*i.e.* occurrence) of each known words. With such, any information about the order or the structure of words in our dataset is discarded. Our model will only focus on whether known words occur or not in a given dataset.

4.2.3.2 Classification algorithm

The choice of which specific learning algorithm to use is a critical step. Many classification algorithms exist [91], each of them having different mapping functions. Our methodology is based on the Decision tree algorithm which predicts the output by learning simple decision rules deduced from the training dataset. One downside of Decision tree models is *over-fitting* [58] which may cause the creation of over-complex trees that do not generalize the data well. To that end, ensemble methods such as Adaptative Boosting classifiers [71] and Random Forest [29] have been introduced. They consist of several decision trees called *weak learners* in which the output are computed through aggregations of the predications of the individual decision trees. Several other classification exists (*e.g.* Support Vector Machine [84], Bayesian Network Classifiers [72], etc.), each of them having different mapping functions. Our choice of using decision tree based models is made because of their transparency (*i.e.* the easy interpretation of the results) and their efficiency for opaque predicates evaluations. The choice of such model is illustrated in Section 4.4 presenting our experiments. Since classification is a common application of machine learning, there are many metrics that can be used for evaluation as describe in the following paragraph.

4.2.3.3 Classification evaluations

Classification problems are a common type of machine learning problem and, as such, there are many metrics that can be used to evaluate the efficiency of a model. In our different experimentations and evaluations, our goal is to measure the accuracy as well as the efficiency in terms of

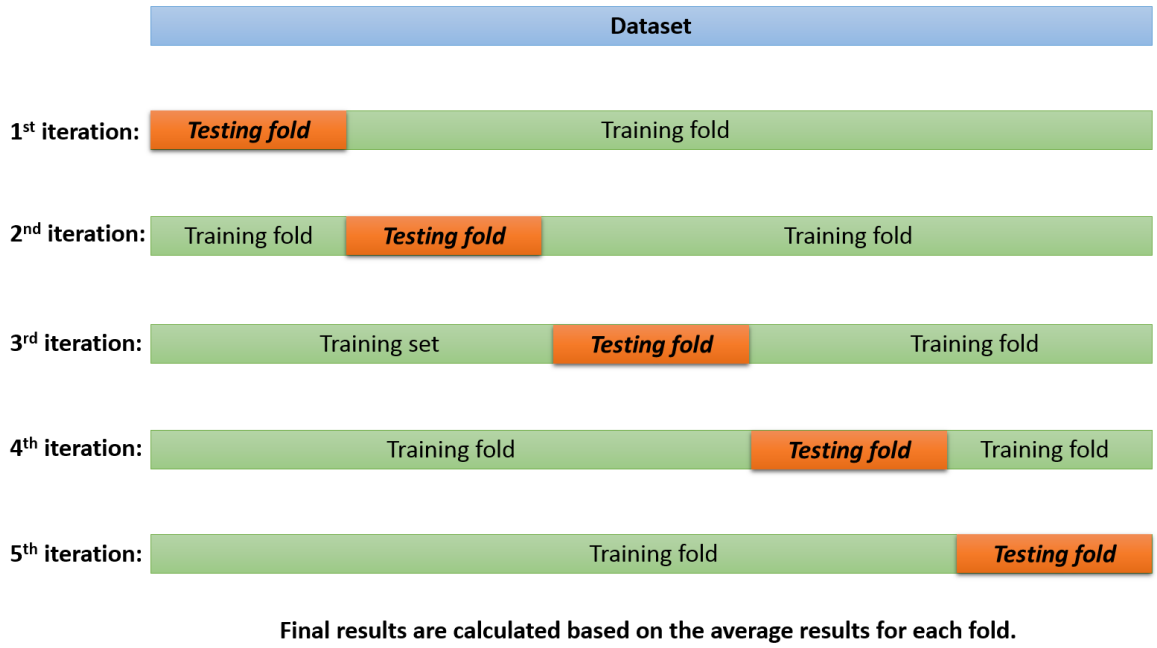

Figure 4.2: Illustration of a 5-folds cross-validation evaluation.

execution time of our models. In order to compute these metrics, *k-Fold Cross-Validation* is a commonly used technique. Cross-validation [107] consists in reserving a particular set of samples on which the model doesn't train. It is a commonly used evaluation methodology in applied machine learning to properly estimate the efficiency of a model on unknown data. Thus, it allows to use a limited set of samples in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training phase. The parameter k refers to the number of folds that a given dataset of samples is split into. Figure 4.2 illustrates the division of the dataset into 5-folds for a cross-validation evaluation. This allows us to calculate the mean of our models *accuracy* as well as the *F1-score* based on the value of k. While the accuracy of the model represents the ratio of correctly predicted labels to the the total of labels, F1-score takes both false positives and negatives into account. We can also estimate the variance of each metrics based on the different portions of the initial datasets on which the predictions are done for the testing phase. Thus, during our evaluations and experiments, accuracy and F1-scores are calculated using *k-fold cross-validation*, with $k = 20$ for a better generalization of our model to unknown instances.

4.3 Our Methodology

Our methodology design is built in two parts. The first part consists in creating a machine learning model for the evaluation and deobfuscation of opaque predicates. The second part uses the

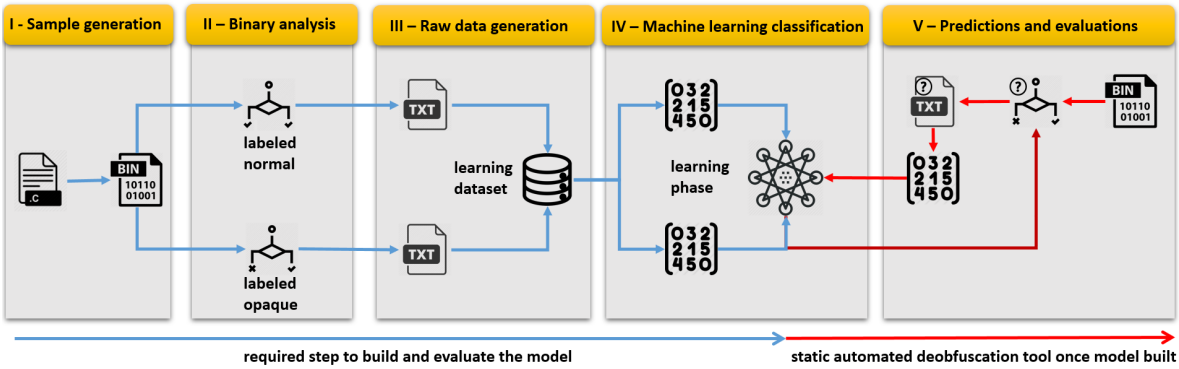

Figure 4.3: Evaluation scheme for the detection and deobfuscation of opaque predicates using both binary analysis and machine learning techniques.

validated model in order to remove such obfuscation transformation statically. Figure 5.2 illustrates our methodology. The first step consists in generating a set of obfuscated binaries. Our datasets of `C` code samples are presented in Section 5.4.1. In the second step, the binary is disassembled and we collect and labelize each predicate, *e.g.* defining if the predicate is opaque or normal, as described in Section 4.3.1. The third step consists in a depth-first search algorithm to collect each path leading to a predicate. We use a thresholded static symbolic execution to collect our raw data for the machine learning model. These data are normalized, processed and used to train and validate our model in a fourth step, as presented in Section 4.3.2. Finally, the fifth and final step shows that our model can be used and integrated in a static deobfuscation tool to predict and remove opaque predicates as presented in Section 5.5.

4.3.1 Binary analysis

Our methodology relies on static symbolic execution to retrieve *the semantics of the predicate constructions* before the machine learning classification models evaluates them. Thus, a first step in our design is the generation of *raw data*. This refers to a representation of data samples that contain noisy features and need to be processed in order to extract informative characteristics from the data samples, before training a model. Since our goal is to evaluate the opaque predicates, we choose to generate our raw data from the disassembled binary code control-flow graph.

Moreover, in order to have a *scalable methodology*, we work statically in order to prevent the need of executing the code. This approach also permits a *better code coverage* compared to existing dynamic approaches. However, our approach can be extended to instruction traces in cases where the analyzed code is encrypted or packed. The raw data used contains the symbolic expressions $\mathscr{S}$ of collected predicates ϕ denoted by $\mathscr{S}_\phi$.

We studied different formats and contents of such raw data as well as their impact on the

efficiency of the trained model (see Section 4.4). In the following sections we present the binary analysis part of our design, namely *thresholded static symbolic execution*, which we employ to generate the raw data from predicates.

4.3.1.1 Thresholded Static Symbolic Execution

Static symbolic execution is a binary analysis technique that captures the semantics (*i.e.* logic) of a program. An interpreter is used to trace the program, while assuming symbolic values for inputs rather than obtaining concrete values as a normal execution would. A symbolic state $\mathbb{S}$ is built and consists in a set of symbolic expressions S for each variables (*i.e.* registers, memory, flags, etc.). Several techniques exist for symbolic execution [12].

In our work we use disassembled functions to collect the symbolic expressions of a predicate $\mathscr{S}_\phi$. We start by generating all possible paths from a function entry point to a predicate ϕ using a depth-first search algorithm. The latter prevents us from using SMT solvers to generate all feasible paths since they are prone to limitations and errors depending on the protections applied. In order to avoid path explosion, we use a *thresholded* static symbolic execution technique that bounds the number of paths generated for one predicate and the amount of time the analysis has to iterate on a loop. Note that our methodology is intra-procedural since publicly available obfuscators, *e.g.* `Tigress` and `OLLVM`, generate intra-procedural opaque predicates.

Path generation. We denote by ϕ_n the n-th predicate within a disassembled function F in a binary B. When a predicate is identified, we generate all paths from F entry point to the collected ϕ_n using a depth-first search (*i.e.* DFS) algorithm. DFS expands a path as much as possible before backtracking to the deepest unexplored branch. This algorithm is often used when memory usage is at a premium, however it remains hampered by paths containing loops. Thus, we use two distinct thresholds, one for loop iterations denoted by α_{loop}, and one for the number of paths to be generated denoted α_{paths}.

Symbolic state generation. In order to have a symbolic state, we use all collected paths of a predicate. We denote by $\mathscr{P}$ the set of all collected paths σ of a predicate ϕ. Let S be the symbolic execution interpreter function such that $\mathrm{S}(\sigma_i) = \mathbb{S}_{\phi^{\sigma_i}}$. In other words, the symbolic execution interpreter S returns a symbolic state $\mathbb{S}_{\phi^{\sigma_i}}$ for a path σ_i, $i \in [O, |\mathscr{P}|]$ of a predicate ϕ. The generated symbolic states for all predicates will be used as raw data and then be processed for the classification models. Algorithm 5 illustrate our methodology for the generation of the raw data. The next section introduces our machine learning part. We will further describe the content of our raw data as well as the feature extractions algorithm used and the different models we want to create.

Algorithm 5 Predicate symbolic state collection

procedure RAW DATA GENERATION(F: disassembled function)
    Initialize a list $\mathscr{P}$ to store the paths
    Initialize a dictionary $\mathbb{S}$ to store symbolic states
    for all predicate ϕ in F **do**
        $\mathscr{P}$ = `DFS_generate_paths`(ϕ, α_{loop}, α_{paths})
        for each path σ in $\mathscr{P}$ **do**
            $\mathbb{S}$ = S(σ)
        end for
        `Generate_raw_data`($\mathbb{S}$)
    end for
end procedure

4.3.2 Machine learning

We experiment different instances for our classification models to study the impact on their accuracy. Since symbolic execution is often based on an intermediate representation that captures all the semantics as well as side effects of the assembly instructions, several intermediate representations exist and are widely used, *e.g.* `LLVM-IR` or MiasmIR [57]. We implemented our methodology using `Miasm2` reverse engineering framework, which integrates translators from MiasmIR to other languages (*e.g.* `SMT-LIBv2` [19], `Python` [163], or C [101]). This gives us the ability to study the impact of the language used to express the symbolic expressions, within our raw data, on our classification models.

4.3.2.1 Raw data

Intermediate representations use concrete values within their generated expressions. This causes raw data to depend on addresses that are specific to some binaries and prevents our models to scale on unknown data. Listing 4.5 illustrates this issue with one predicate symbolic expression in the MiasmIR language. Moreover, some intermediate representations, *e.g.* MiasmIR, use identifiers in order to express modified registers name or memory locations. This may further affect the scalability of our trained models.

```
# MiasmIR predicate expression of an P^T opaque predicate
ExprId('IRDst', size=64) = ExprInt(0x401e87, 64)

# MiasmIR predicate expression of an P^F opaque predicate
ExprId('IRDst', size=64) = ExprInt(0x4028f8, 64)
```

Listing 4.5: MiasmIR predicate expressions with identifiers and concrete values

For the purpose of having a model that can scale to unknown data, we use a normalization phase that replaces identifiers and concrete values by symbols, and non-alphanumerical characters by alphanumerical words. This is a necessary step for a complete feature extraction phase that sometimes excludes non-alphanumerical characters when working on text-based raw data. In Listing 4.6 we provide examples of the normalization step, based on Listing 4.5 predicates.

```
# MiasmIR predicate expression of an P^T opaque predicate
ExprId(id1, size=64) = ExprInt(v1, 64)

# MiasmIR predicate expression of an P^F opaque predicate
ExprId(id1, size=64) = ExprInt(v1, 64)
```

Listing 4.6: MiasmIR predicate expressions after our normalization phase

Since our methodology computes a full symbolic state from any function entry-point to a targeted predicate, there is a need to know if all information within the collected symbolic state is relevant for our models. The goal is to have many features for an accurate classification without adding too much noise. Another issue to be avoided is having raw data samples that do not contain enough information to distinguish between samples that have different labels, as illustrated also in Listing 4.6. In other words, we may have two expressions that are identical but have different labels, *e.g.* the first being the expression of a P^T and the second an expression of a P^F.

```
# MiasmIR predicate expression of an P^T opaque predicate
ExprId('af', size=1) = ExprSlice(ExprOp('^', ExprOp('+', ExprId('RSP_init',
     size=64), ExprInt(0xfffffffffffffff8, 64)), ExprOp('+', ExprId('
    RSP_init', size=64), ExprInt(0xfffffffffffffff8, 64)), ExprInt(0x40,
    64)), 4, 5)
ExprId('RBP', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xffffffffffffffff8, 64))
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffc0
    , 64)), size=64) = ExprId('RDI_init', size=64)
ExprId('pf', size=1) = ExprInt(0x1, 1)
ExprId('RAX', size=64) = ExprOp('call_func_ret', ExprInt(0x400510, 64),
    ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8,
    64)), ExprId('RCX_init', size=64), ExprId('RDX_init', size=64), ExprId(
    'R8_init', size=64), ExprId('R9_init', size=64))
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff0
    , 64)), size=64) = ExprMem(ExprOp('segm', ExprId('FS_init', size=16),
    ExprInt(0x28, 64)), size=64)
ExprId('IRDst', size=64) = ExprInt(0x401e87, 64)
ExprId('zf', size=1) = ExprInt(0x1, 1)
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8
    , 64)), size=64) = ExprId('RBP_init', size=64)
ExprId('of', size=1) = ExprInt(0x0, 1)
ExprId('nf', size=1) = ExprInt(0x0, 1)
ExprId('cf', size=1) = ExprInt(0x0, 1)
ExprId('RSP', size=64) = ExprOp('call_func_stack', ExprInt(0x400510, 64),
    ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8,
    64)))
ExprId('RDI', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xfffffffffffffffe0, 64))
```

Listing 4.7: MiasmIR always true invariant opaque predicate expressions after our normalization phase

To avoid this matter we use the thresholded symbolic execution, which generates expressions for each path leading to a predicate. Listings 4.7 and 4.8 illustrate respectively the P^T and P^F predicates expressions from Listing 4.6 along with others memory and registers expressions from their symbolic state. We can see that now we have more informations that allows us to distinguish between both predicates.

```
# MiasmIR predicate expression of an P^F opaque predicate
ExprId('af', size=1) = ExprInt(0x0, 1)
ExprId('RBP', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xfffffffffffffff8, 64))
ExprId('RCX', size=64) = ExprOp('^', ExprMem(ExprInt(0x604078, 64), size
    =64), ExprInt(0x1, 64))
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffc0
    , 64)), size=64) = ExprId('RDI_init', size=64)
ExprId('pf', size=1) = ExprOp('parity', ExprOp('&', ExprCompose(ExprOp('&',
     ExprMem(ExprInt(0x604078, 64), size=32), ExprInt(0x1, 32)), ExprInt(0
    x0, 32)), ExprInt(0xff, 64)))
ExprId('RAX', size=64) = ExprInt(0x2, 64)
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff0
    , 64)), size=64) = ExprMem(ExprOp('segm', ExprId('FS_init', size=16),
    ExprInt(0x28, 64)), size=64)
ExprId('IRDst', size=64) = ExprInt(0x4028f8, 64)
ExprId('zf', size=1) = ExprCond(ExprCompose(ExprOp('&', ExprMem(ExprInt(0
    x604078, 64), size=32), ExprInt(0x1, 32)), ExprInt(0x0, 32)), ExprInt(0
    x0, 1), ExprInt(0x1, 1))
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8
    , 64)), size=64) = ExprId('RBP_init', size=64)
ExprId('RDX', size=64) = ExprOp('^', ExprMem(ExprInt(0x604078, 64), size
    =64), ExprInt(0x1, 64))
ExprId('nf', size=1) = ExprInt(0x0, 1)
ExprId('cf', size=1) = ExprInt(0x0, 1)
ExprId('RSP', size=64) = ExprOp('call_func_stack', ExprInt(0x400510, 64),
    ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8,
    64)))
ExprId('RIP', size=64) = ExprInt(0x4028f8, 64)
ExprId('of', size=1) = ExprInt(0x0, 1)
ExprId('RDI', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xfffffffffffffff0, 64))
```

Listing 4.8: MiasmIR always false invariant opaque predicate expressions after our normalization phase

We study the use of several expressions in our raw data to distinguish between samples that have different labels. To this end, we divide our instances into three sets:

- **Set 1**: with samples containing only the expression of the predicate in a static single assignment form (*i.e.* SSA) as illustrated in Listing 4.6.
- **Set 2**: with samples containing only the expressions of the predicate and its corresponding flags in an SSA form.
- **Set 3**: with samples containing the full symbolic state of a path, from an entry-point to a targeted predicate, *i.e.* all memory, flags, and registers modified in a SSA form as illustrated in Listing 4.8 or 4.7.

In Section 4.4.2.2, each set is studied in order to find the best possible raw data content. We start by calculating for each set the similarity percentages based on 5000 samples of predicates, either *normal* or *opaque* predicates generated by the `Tigress` obfuscator on a dataset of `C` code samples (see Section 5.4.1). In other words, we search for raw data with different labels (*e.g.* P^F and P^T) but with the same content. As we can see in Table 4.4, only the Set 3 has a low rate of similarities between opaque or legit raw data content (3.5%) and between P^T and P^F raw data (6%). This indicates that Set 3 is more suited for our raw data representation.

Raw data	Detection similarities	Deobfuscation similarities
Set 1	24.94%	31.92%
Set 2	17.38%	26.62%
Set 3	**3.5%**	**6%**

Figure 4.4: Number of raw data content similarities experiments in different datasets of various contents.

4.3.2.2 Decision tree based models

Decision trees [161] predict the output by learning simple decision rules deduced from the training dataset. The internal nodes of a decision tree contain binary conditions based on input features vectors, whereas the leaves are associated with the class labels we want to predict. Decision trees are built recursively. The root node contains all the training instances and each internal node splits its training instances into two subsets according to a condition based on the input. Leaf nodes however represent a classification or decision on these training instances. Different approaches exist for the splitting conditions of internal nodes [83]. However, one downside of decision tree models is *over-fitting* [58] which may cause the creation of over-complex trees that do not generalize the data

well. In our case, the decision tree model is capable of identifying and deobfuscating an opaque predicate $\mathcal{O}(\phi)$. We choose to create two distinct models: a first one that evaluates the stealth of an opaque predicate and a second one to evaluate its resiliency, as presented in the following paragraphs.

Model for stealth (detection). The construction of a classifier consists in the definition of a mapping function $C_f : \mathscr{D} \rightarrow [0,1]$ that, given a document d (*i.e.* an input), returns a *class label*, which is represented by a number (here 0 or 1) that defines the category of d. Applied to the evaluation of opaque predicates stealthiness, the function can be seen as:

$$C_f : \mathscr{D} \rightarrow [\text{NORMAL, OPAQUE}].$$

In other words, given the term-frequency vector of a symbolic execution state D, from a function entry point to a predicate, our model mapping function C_f will return two values: NORMAL or OPAQUE. If a model is capable of detecting a predicate as opaque, we can assume that the transformation is not stealthy.

Model for resiliency (deobfuscation). In order to evaluate the resiliency of an opaque predicate, we construct a model with a different function as presented for the evaluation of stealthiness. Indeed, our goal is to predict if an opaque predicate is of type P^T or P^F, thus, the function $C_f : \mathscr{D} \rightarrow [0,1]$ in that context can be expressed as:

$$C_f : \mathscr{D} \rightarrow [\text{TRUE, FALSE}].$$

The choice of the best suited classification algorithm is often made on accuracy but in our work we choose our model based on its *transparency* to easily interpret our results. Since many learning algorithms exist, the next section will present our experiments to select the best classification model for both detection and deobfuscation of opaque predicates.

4.4 Experiments

In this section we present our study of efficient and accurate creation of classification models. We start by introducing the datasets variety used in our work.

4.4.1 Datasets

Our experiments are made on several `C` code samples. We use the `scikit-learn` API [145] for the implementation of the models. The datasets contain various types of code, each of them having different functionalities in order to have a model that does not fit to a specific type of program, as listed below:

- `GNU` core utilities (*i.e.* `core-utils`) binaries [151] for normal predicate samples;
- Cryptographic binaries for obfuscated and non-obfuscated predicates [49];
- Samples from [13] containing basic algorithms (*e.g.* factorial, sorting, etc.), non-cryptographic hash functions, small programs generated by `Tigress`;
- Samples involving the uses of structures and aliases [4, 88].

Our choice is motivated by their low ratio of dependencies and their straightforward compilation. This makes their obfuscation possible using tools such as `Tigress` and `OLLVM` without errors during compilation. A list of all different combinations of obfuscation transformations and options related to Tigress is given in Appendix A.1 and Listing 22.

4.4.1.1 Dataset size determination

One important point is to determine the amount of samples required since this can significantly impact the cost of our studies and evaluations, as well as the reliability of our results. If too many samples are collected, we face a longer evaluation time but if there are not enough samples in our dataset, our results may be irrelevant. Several propositions based on statistical tests allow to determine the size of our datasets depending on the area of research [66]. Based on this work, we estimated the required samples based on several parameters:

- The confidence level, *i.e.* how confident to we need to be that the classification of our model did not occur by chance. We set this parameter at 99%, leaving only 1% to chance.
- The percentage of difference that we want to detect. The lower the percentage, the more sample is required, thus we set this parameter at 1%.

- The distribution of our samples which is supposed to be balanced, *e.g.* 50% of opaque predicate and 50% of normal ones.

Figure 4.5 illustrates the amount of samples required for two confidence levels, 95% and 99%, depending on the probability of difference we want classify. As we can see, the best possible size of datasets for our use-cases should contain 27.000 samples. However, the generation of that amount of samples for each opaque predicates constructions provided by each obfuscators is costly. To that end, we use in the followings datasets with 5000 to 15.000 samples in order to have a high probability of detection and of confidence level. Each of our datasets are *balanced*, *i.e.* with an equal number of samples of each classes. Next, we present our studies using these datasets.

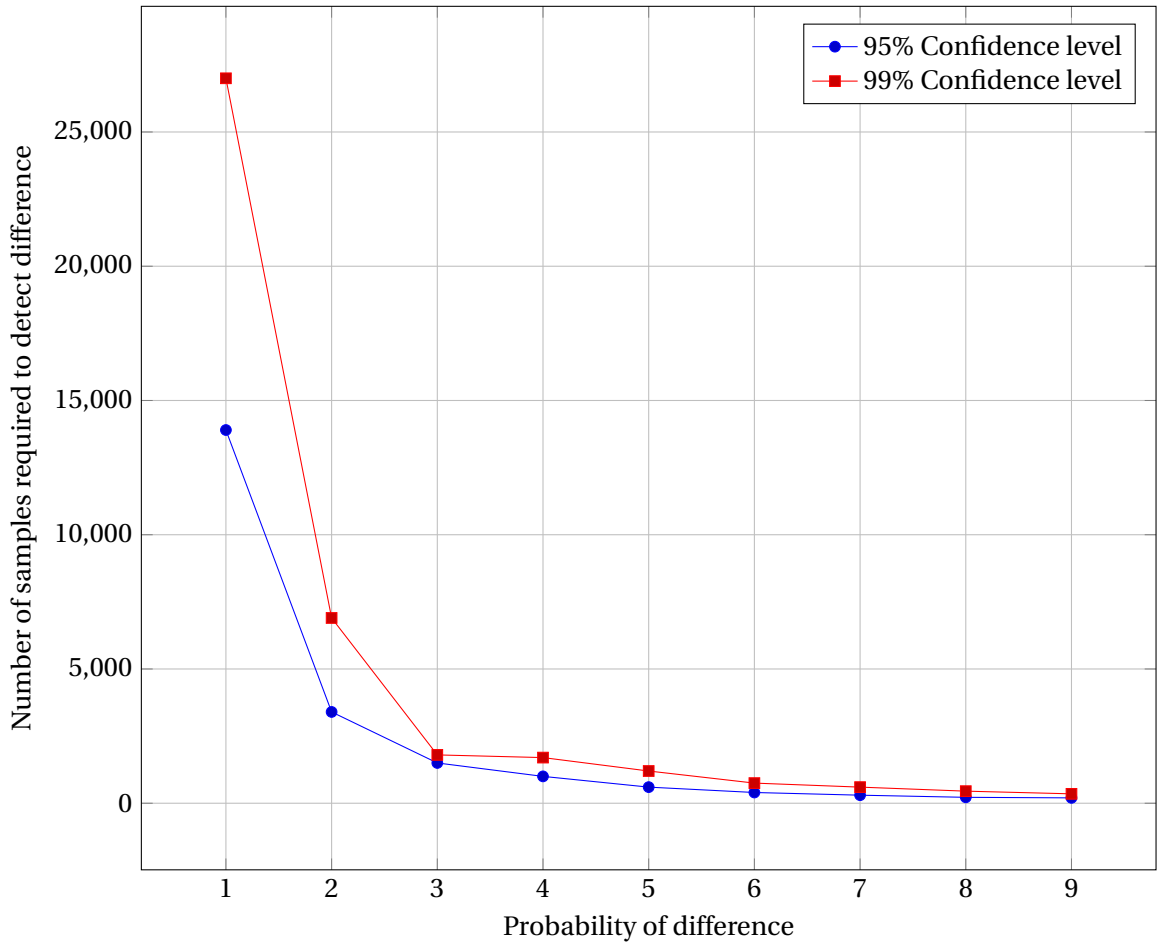

Figure 4.5: Number of sample required to detect between 1% to 9% of difference between in our use-cases.

4.4.2 Preliminary studies

The goal of our experiments is to investigate and answer the following questions:

- **Study 1:** Which raw data language is the most efficient (in terms of time and space) and also the most accurate?
- **Study 2:** Which raw data content best expresses the normal and opaque predicates?
- **Study 3 and 4:** Which classification model is more accurate and which feature extraction algorithm is best suited?

The following paragraphs present our experiments for each question. For this section and for our evaluations (see Section 5.5) we used a laptop running `Windows` 7 with 16 GB of RAM and a Intel Core i7-6820HQ vPro processor.

4.4.2.1 Study 1: Raw data language selection

Our goal is to select the most appropriate language for the symbolic execution engine. We use MiasmIR, which we compare with the translators it implements in `SMT-LIBv2` language, `C`, and `Python`. After normalizing these languages, as presented in Section 4.3.2.1, we use our dataset of normal predicates from core-utils binaries along with structured-based opaque predicates from `Tigress` to study several points:

1. Which set of samples is more efficient in terms of disk space?
2. Which set of samples is more efficient in terms of computation time?
3. Which language is more accurate for our models when representing our raw data?

Raw data language	Miasm2	SMT-LIBv2	C	Python
Detection accuracy (%)	94%	90%	87%	87%
Deobfuscation accuracy (%)	88%	80%	78%	78%
Execution time for detection (s)	15s	114s	21s	20s
Execution time for deobfuscation (s)	12s	50s	15s	13s
Size of dataset (GB)	1.91GB	37.4GB	2.11GB	1.98GB

Figure 4.6: Study of the raw data language accuracy and efficiency

Table 4.6 illustrates our experiments using 20-fold cross-validation on decision-tree based models. For each language, we used a dataset of 10000 balanced samples. We observe that `Miasm2` intermediate representation gives higher accuracy rates for both the detection and deobfuscation

model. Moreover, it is more efficient in terms of disk space used (as opposed to the `SMT-LIBv2` dataset), which leads to a faster time of execution. This is mainly due to the fact that `Miasm2` intermediate language has a small set of terms expressing the semantics of the code as compared to other languages in our study. According to these results, we choose `Miasm2` for all of our raw data samples for the remaining of the chapter.

4.4.2.2 Study 2: Raw data content selection

It remains to single out the most suitable content that will express the construction of normal and invariant opaque predicates. Table 4.4 in Section 4.3.2.1 shows that the use of full symbolic state representation prevents having similarities between samples of different classes (*i.e.* labels). Thus,

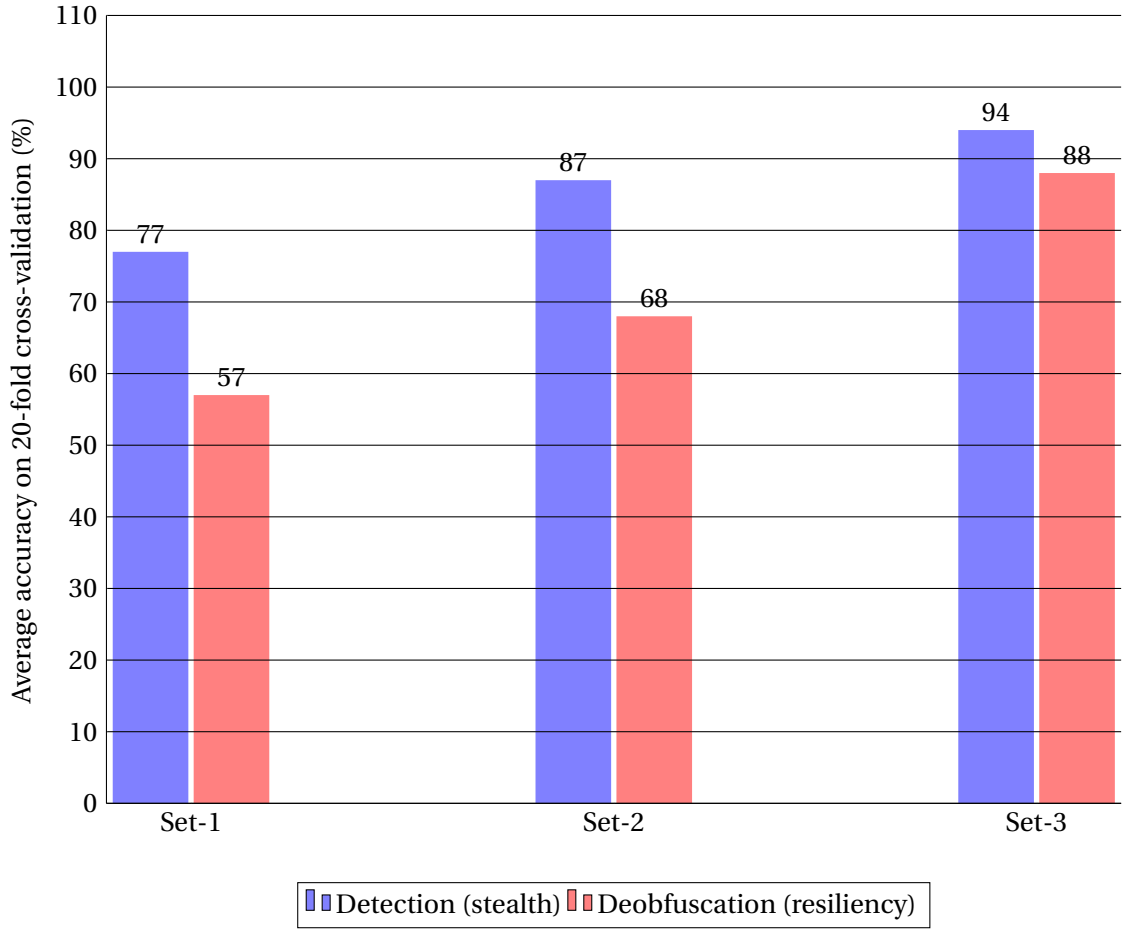

Figure 4.7: Predictions accuracy on the different raw data sets

based on the same dataset of core-utils and structured-based opaque predicates generated with `Tigress`, we measure the average of our models accuracies for both detection and deobfuscation, evaluated with a 20-fold cross-validation. Figure 4.7 confirms that the Set 3, *i.e.* the full symbolic

state, presents a better accuracy for both detection (at 94%) and deobfuscation (at 88%) when using the decision tree algorithm on balanced datasets of 10000 samples.

4.4.2.3 Study 3 and 4: Classification algorithm and feature extraction selection

In order to properly evaluate our methodology, we need to select the appropriate features extraction techniques combined with an accurate classification algorithm.

We have done experiments with the most common classifications models [108], namely decision trees, k-nearest neighbors [53], support vector machines, neural network [176, 207], naive Bayes [93] and random forest [30]. The use-case of our experiments is to evaluate the stealth of structured-based opaque predicates generated with `Tigress` on our datasets. The features are expressed using *term-frequency* (*i.e.* tf) vectors as well as *td-idf* vectors in order to compare both extraction techniques. Default parameters are applied for each classification algorithms used in our study.

Classification algorithm	Term-frequency vectors	TD-IDF vectors
Decision-tree	94%	93%
k-Nearest Neighbors	91%	92%
Support Vector Machine	87%	71%
Linear Support Vector Machine	77%	83%
Multi-layer Perceptron	84%	92%
Multinomial Naive-Bayes	58%	75%

Figure 4.8: Accuracy results of different classification models using term-frequency and td-idf vectors.

The evaluation is made with a 20-fold cross-validation, each time increasing the number of samples in order to have an *accuracy curve* for our models. Features are expressed using term-frequency vectors as well as inverse document frequency vectors in order to compare both extraction techniques. Figures 4.9 illustrate the learning curves of each studied algorithms on datasets with different sizes. Table 4.8 illustrate our results. We can observe that the decision tree model stands out from others when term-frequency vectors are used. It averages 94% of detection accuracy whereas k-Nearest Neighbors averages 91%. As for the use of *td-idf* vectors, the decision tree model has a better accuracy at 93%.

According to this experiment, we choose the **Decision-tree** classification algorithm with **term-frequency** as features extraction technique in our methodology.

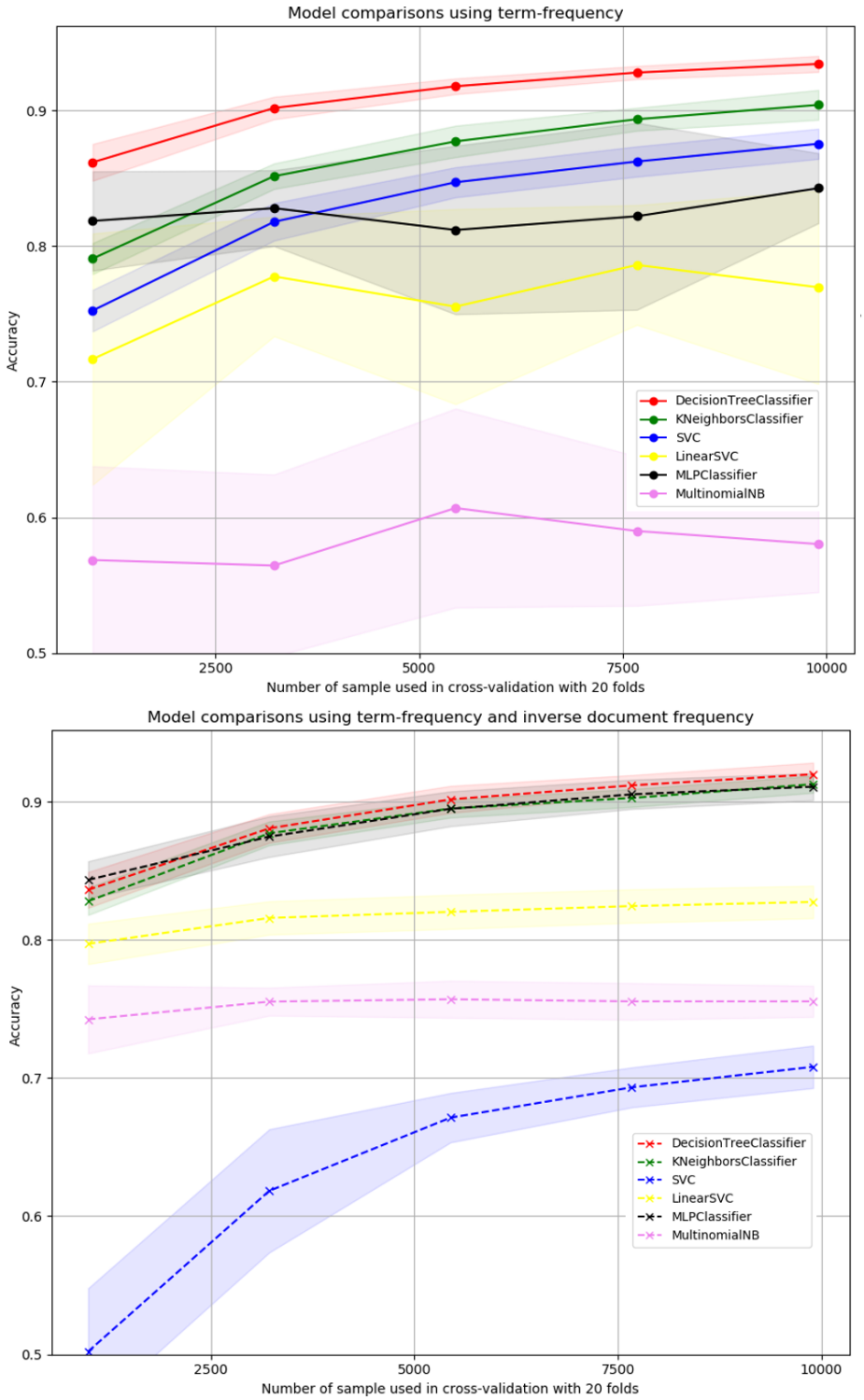

Figure 4.9: Learning curves of several models using tf and tf-idf for the detection of `Tigress` opaque predicates

4.5 Evaluations

Our goal in this section is to evaluate opaque predicates stealth and resiliency using a model based on decision trees. We divide our evaluation into two parts:

1. *Stealth:* can the model differentiate an opaque predicate from a normal predicate, ***i.e.* is the opaque predicate stealthy?**
2. *Resilience:* can the model differentiate a P^T opaque predicate from a P^F opaque predicate, ***i.e.* is the opaque predicate resilient?**

4.5.1 Measuring stealth

In this section we focus on the evaluation of stealthiness of opaque predicates. Namely, we want to see if our model is able to distinguish opaque predicates from normal predicates. Based on our datasets, our goal is to measure the efficiency of our model for the detection of opaque predicates based on different constructions. *Note that each dataset is balanced and contains 10000 samples.*

	Dataset 1	**Dataset 2**	**Dataset 3**
Number of samples	10000	10000	10000
% Opaque samples	46.03%	49.99%	50.03%
% Normal samples	53.97%	50.01%	49.97%
Types of opaque predicates	Arithmetic, Environment-based	Arithmetic, Structure-based	Arithmetic, MBA, Structure-based
Other transformations	None	None	EncA, EncL, EncD, Flat, Virt
Analysis time (s)	1.13 s	1.74 s	1 s
Accuracy (%)	93 %	95 %	99 %
F1-score (%)	93 %	95 %	98 %
Variance (%)	1 %	2 %	1 %

Figure 4.10: Evaluations of stealth (detection) using `Tigress`

4.5.1.1 Tigress

The `Tigress` obfuscator can generate a variety of complex obfuscation transformations, *e.g.* MBA-based, structured-based or environment-based. To this end, we use several datasets of different opaque predicates constructions, balanced with normal predicates, to evaluate our model for detection. *Dataset 1* contains arithmetic, MBA and environment-based opaque predicates whereas *Dataset 2* contains structured-based (*i.e.* alias-based) opaque predicates. Moreover, we used a third dataset (*Dataset 3*) that combines these opaque predicates with other obfuscation transformations

such as arithmetic, literal, and data encodings (*i.e. EncA*, *EncL*, and *EncD*, respectively) joined with control-flow flattening (*Flat*) and code virtualization (*Virt*).

Our results are illustrated in Table 4.10. Regardless of their types and of the implication of other obfuscation transformations, our detection model is able to efficiently predict if a predicate is *opaque* or *normal*. Indeed, the detection of arithmetic and environment-based opaque predicates scores an accuracy and F1-score of 93%, whereas arithmetic and structured-based opaque predicates are less stealthy for our model with scores up to 95%. However, as more obfuscation techniques are combined with opaque predicates, our predictions accuracy and F1-score rises to respectively 99% and 98%. This is due to the fact that opaque predicates patterns, once combined with other combination of transforms, become more specific thus lower their stealthiness. In our case however, code virtualization (*i.e.* Virt) is applied before opaque predicates, as illustrated in Appendix A.1. The opposite, namely applying code virtualization after other transformations, is a limitation to our methodology since the generated opaque predicates will be virtualized, thus transformed into byte-code.

4.5.1.2 OLLVM

In order to evaluate our model against opaque predicates generated by `OLLVM`, we split our evaluations in two sets. The first set uses samples obfuscated only with opaque predicates (*i.e.* the bogus control-flow transformations *bcf*). The second set uses samples obfuscated with opaque predicates combined with control-flow flattening and instructions substitutions (*i.e. fla* and *sub*, respectively) to see if we can evaluate opaque predicates stealthiness when they are combined with others transformations. Table 4.11 illustrates our results. In the second dataset, our model is

	Dataset 1	**Dataset 2**
Number of samples	10000	10000
% Opaque samples	50.02%	49.97%
% Normal samples	49.98%	50.03%
Types of opaque predicates	Arithmetic-based	Arithmetic-based
Other transformations	None	fla, sub
Analysis time (s)	2 s	1 s
Accuracy (%)	89 %	95 %
F1-score (%)	89 %	94 %
Variance (%)	3 %	2 %

Figure 4.11: Evaluations of stealth (detection) using `OLLVM`

able to efficiently detect the labels of most predicates. However, when opaque predicates are not combined with other obfuscation transformation, we observe a loss of efficiency, from 95% to 89%

accuracy. This indicates that OLLVM opaque constructions are stealthier than other constructs, thus our model cannot properly distinguish them from normal predicates. At best, it will requires more training samples for our model in order to have a better accuracy. One reason for their stealthiness in regard to our model is the fact that OLLVM arithmetic opaque predicates are bloc-centric, with basic encodings, which may have similar patterns to normal predicates from hash functions or cryptographic codes in our datasets. However, when they are combined to the other transformations, their patterns become more specific and our model has better prediction results.

4.5.1.3 Bi-opaque

Several constructions exist for bi-opaque predicates, among which float-based (*i.e.* using floating instructions) or symbolic-memory based. We use their obfuscator based on the OLLVM framework to evaluate our detection model. As we can see in Table 4.12, our model is efficient at detecting

	Dataset 1	**Dataset 2**
Number of samples	10000	10000
% Opaque samples	49.98%	50.02%
% Normal samples	50.02%	49.98%
Types of opaque predicates	float-based	symbolic-memory based
Other transformations	None	None
Analysis time (s)	0.6 s	0.9 s
Accuracy (%)	93 %	98 %
F1-score (%)	93 %	98 %
Variance (%)	2 %	4 %

Figure 4.12: Evaluations of stealth (detection) using Bi-opaque predicates from [200]

bi-opaque predicates with 93% accuracy for float-based constructs. Bi-opaque predicates are constructed based on the same patterns as OLLVM opaque predicates but using floating-point instructions and registers instead. However, symbolic-memory based constructs rely on more specific patterns, thus allowing a better detection rate at 98% accuracy and F1-score.

4.5.2 Measuring resiliency

Once a predicate is detected as being *opaque*, our goal is to measure its resiliency. In other words, we want to know if our model is able to deobfuscate, *i.e.* predict the output of the opaque predicate. Our evaluations are based on invariant opaque predicates, P^T and P^F, generated using different constructions.

4.5.2.1 Tigress

The patterns between P^T and P^F are more difficult to predict since both predicates are opaque and generated using the same construction. However, the underlying invariant properties render our models efficient towards their deobfuscation. Table 4.13 shows our results. We can observe that

	Dataset 1	**Dataset 2**	**Dataset 3**
Number of samples	5000	5000	5000
% Opaque samples	51.50%	50.02%	50.02%
% Normal samples	48.50%	49.98%	49.98%
Types of opaque predicates	Arithmetic, Environment-based	Arithmetic, Structure-based	Arithmetic, MBA, Structure-based
Other transformations	None	None	EncA, EncL, EncD, Flat, Virt
Analysis time (s)	0.3 s	1 s	3 s
Accuracy (%)	90 %	88 %	92 %
F1-score (%)	91 %	87 %	92 %
Variance (%)	3 %	3 %	2 %

Figure 4.13: Evaluations of resiliency (deobfuscation) using `Tigress`

our model is able to detect environment-based invariants with scores of 90% accuracy and 91% of F1-score on balanced datasets of 5000 samples. For structure-based invariants, we get slightly lower results, with 88% and 87% of accuracy and F1-score. This is due to the fact that structured-based invariants use aliasing, producing patterns which are less dissimilar than for environment-based opaque predicates. However, our model has a better accuracy and F1-score (92% for both) when other transformations are used. Thus, we are able to efficiently and accurately predict the invariant value of opaque predicates generated with `Tigress`, regardless of their constructions, and of the combination of obfuscation transformations used..

4.5.2.2 Bi-opaque, OLLVM and Tigress

Since OLLVM only produces P^T opaque predicates, we choose to combine all available samples generated from our three evaluated obfuscators. A first dataset is used to evaluate our deobfuscation models against normal predicates and opaque predicates generated without any other transformations. A second dataset is used to combine opaque predicates with others existing transformations from these obfuscators. *Note that all datasets are balanced and contain 15000 samples.* Our results in Table 4.14 show that our methodology is efficient against all patterns of opaque predicates from available obfuscators. Our model is able to detect the invariant patterns of all the opaque predicate constructs with 92% accuracy and 91% F1-score. Moreover, when these opaque predicates are

	Dataset 1	Dataset 2
Number of samples	15000	15000
% Opaque samples	50.02%	50.00%
% Normal samples	49.98%	50.00%
Types of opaque predicates	Arithmetic, MBA, Environment, Structure, Symbolic-memory, float-based	Arithmetic, MBA, Environment, Structure, Symbolic-memory, float-based
Other transformations	None	fla, bcf, EncA, EncL, EncD, Flat, Virt
Analysis time (s)	1 s	0.5 s
Accuracy (%)	92 %	95 %
F1-score (%)	91 %	95 %
Variance (%)	4 %	5 %

Figure 4.14: Evaluations of resiliency (deobfuscation) using Bi-opaque, `OLLVM` and `Tigress`

combined with other obfuscation transformations, the scores rise up to 95%.

4.5.3 Deobfuscation methodology

Our methodology can be used as an efficient deobfuscation technique, if it is based on an adequate dataset of training samples. We developed our methodology as an experimental `IDA` [86] plug-in that detects directly on the disassembled binary any opaque predicates and deobfuscates them, if needed. We will compare our results with existing opaque predicates deobfuscation tools based on SMT solvers and symbolic execution, such as DROP [158]. The latter is an `IDA` Pro plug-in based on `Angr`, which uses static symbolic execution for the removal of invariant and contextual opaque predicates. Meanwhile, for the dynamic symbolic execution, we use `Miasm2` dynamic symbolic execution engine. We employ several datasets of opaque predicates obfuscated with various constructions and transformations. Moreover, we remove all samples used in our evaluations datasets from our learning samples used to built our model.

Our invariant opaque predicates are generated mainly from [13] and Table 4.15 shows the results. For each deobfuscation tool we use several samples obfuscated by different obfuscators (*c.f.* column Obfuscator) and obfuscation transformations (*c.f.* Obfuscation). Column "OP detection rate" indicates the percentage of removed opaque predicates, whereas column "#FP, #FN" shows the number of false positive and false negative results respectively. Finally column "Errors" indicates if an error occurred during the analysis, *e.g.* lack of memory or a timeout.

We observe that, for a static analysis, our experimental plug-in performs better at removing opaque

predicates with complex constructs such as the one generated by `Tigress`, or the bi-opaque constructs. We obtain better results than the experimental plug-in `DROP`, as well as a better rate than DSE-based techniques for most constructions of opaque predicates.

Tool	Obfuscator	Obfuscation	OP detection rate %	#FP, #FN	Error
DROP	OLLVM	bcf	100%	1,0	0
	OLLVM	bcf, sub	100%	0,0	1
	Bi-opaque	float	100%	4,0	0
	Bi-opaque	symbolic-memory	75%	1,5	2
	Tigress	Environment-based	60%	1,8	0
	Tigress	Structure-based	25%	2,12	1
	Tigress	MBA, struct	10%	0,10	8
Our methodology	OLLVM	bcf	100%	0,0	0
	OLLVM	bcf, sub	100%	0,0	0
	Bi-opaque	float	92%	0,0	0
	Bi-opaque	symbolic-memory	100%	1,0	0
	Tigress	Environment-based	88%	2,3	0
	Tigress	Structure-based	82%	1,4	0
	Tigress	MBA, struct	85%	2,2	0
Miasm DSE	OLLVM	bcf	100%	0,0	0
	OLLVM	bcf, sub	100%	0,0	0
	Bi-opaque	float	100%	0,0	0
	Bi-opaque	symbolic-memory	85%	0,3	0
	Tigress	Environment-based	88%	1,2	0
	Tigress	Structure-based	65%	1,7	0
	Tigress	MBA, struct	52%	2,10	6

Figure 4.15: Comparisons of opaque predicates deobfuscation using machine learning vs. SMT-solver based analyses.

4.6 Limitations and perspectives

Our experiments and evaluations underline the efficiency of decision tree models to detect and deobfuscate opaque predicates. The most important achievement of our technique is that it allows a generalization to most invariant opaque predicates constructions. Next we enumerate the limitations of our method.

A first limitation is due to decision tree models and the switch between obfuscators. Namely, we can observe that a model that learns from samples generated using one obfuscator, cannot efficiently fit to transformations of another obfuscator if they use different kinds of constructions. This also hinders our ability to detect new constructions of opaque predicates.

A second limitation comes from the use of static symbolic execution to generate the symbolic state as a raw data. Such process is part of the deobfuscation application of our methodology, and, as any static analysis, may be time consuming. This explains the use of our thresholded static symbolic execution in order to prevent as much as possible issues such as path explosion.

Our work proposes a new application of machine learning techniques for the purpose of evaluating obfuscation transformations, and also for removing them in a static automated manner. Our experimentations and evaluations, indicate that our design can be extended to other complex constructions of opaque predicates such as thread-based and hash-based constructs. Future work includes also a more in-depth study of obfuscation transforms combinations and options as well as the generation of deobfuscated program to report any good or bad behaviors (*e.g.* crashes).

4.7 Related work

Many binary analysis techniques are often based on pattern matching for either detecting plagiarism, or malicious behaviors. Recent studies show the efficiency of machine learning and deep learning techniques for the detection and classification of malwares, *e.g.* [157], which also implicates the detection of similar codes within the malwares samples. More closely related to the obfuscation area, the work in [166] aims at recovering meta-data information using machine learning techniques. Their goal is to detect the obfuscation transformation used in several protected binaries generated by `Tigress`. Their evaluations show that naive Bayes and decision tree models can be efficient at detecting obfuscation transformations using filtered instruction traces. However, their work focuses on the recovery of informations about the obfuscation techniques used, but it does not aim at deobfuscating.

Another work, [13], aims at predicting the resiliency of obfuscated code against symbolic execution attacks. They use machine learning to measure the ability of several different symbolic execution engines to run against various layers and combinations of obfuscation techniques. Nevertheless, machine learning is not primarily used to remove any obfuscation transforms.

To summarize, existing work shows that machine learning techniques are pertinent with respect to the classification or the detection of features, within binary samples. However, to the best of our knowledge, no deobfuscation study and methodology exists regarding these techniques. For this reason, in this chapter, we proposed an efficient way to evaluate both the stealth and the resilience of opaque predicates through several studies and experiments combining binary analysis technique and machine learning.

4.8 Conclusion

In this contribution we applied machine learning techniques to the evaluation of opaque predicates. By introducing the different constructions of opaque predicates and the limitations from dynamic symbolic execution techniques and SMT solvers, we underlined the importance of studying other alternatives for generic evaluations of these transformations.

We proposed a new approach that bridges a thresholded static symbolic execution with machine learning classification to evaluate both the stealth and resilience of invariant opaque predicates constructions. The use of static symbolic execution allows us to have a better code coverage and scalability, which combined with a machine learning model, permits a generic approach by discarding the use of SMT solvers. Our studies illustrate that our choices conduct towards the implementation of an efficient and accurate evaluation framework against state of the art obfuscators. We created two models for the evaluation of stealth and resiliency of state-of-the-art opaque predicates constructions, with results up to 99% for detection and 95% for deobfuscation. Moreover, we extended our work to a deobfuscation plug-in and compared our results to other tools, showing the efficiency of machine learning for the deobfuscation of most invariant opaque predicates constructions. As future work, we propose to extend machine learning techniques to the evaluation of other obfuscation transformations as well as a more in-depth study of deep learning techniques, which we envision to render promising results.

We believe that our work provides a new framework to evaluate opaque predicates transformations, as well as a new alternative towards their static and automated deobfuscation.

Chapter 5

Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and semantic reasoning

Contents

The third contribution of this thesis consists in a novel approach that combines semantic reasoning techniques with ensemble learning classification for the purpose of providing a static detection framework of obfuscation transformations. The ability to efficiently detect the software protections used is at a prime to facilitate the selection and application of adequate deobfuscation techniques. Thus, we provide several studies for the best practices of the use of machine learning techniques for a scalable and efficient model. Moreover, we extend our work to detect constructions of obfuscation transformations, thus providing a fine-grained methodology. According to our experimental results and evaluations on obfuscators such as `Tigress` and `OLLVM`, our models have up to 91% accuracy on state-of-the-art obfuscation transformations. Our overall accuracies for their constructions are up to 100%. By contrast to existing work, we provide solutions that exploit semantic reasoning, *i.e.* semantics, as opposed to disassembled code. Our approach underlines the efficiency of semantic reasoning combined with advanced machine learning techniques, such as ensemble learning and multi-label with multi-output classification models.

5.1 Introduction

In order to properly evaluate obfuscation transformations, or to efficiently analyze malwares, many deobfuscation techniques have emerged. Their goal is to remove the protection layers applied on the code. The deobfuscation process can be seen as different strategies such as reverting, simplifying, or gathering information about the obfuscated code. For this contribution, we mainly focus on information gathering, namely, the static detection of obfuscation transformations. We also study an extension to the transformations constructions. This approach is previously known as *metadata recovery attacks* [166], as it will be introduced next.

As discussed in Chapter 2, state-of-the-art deobfuscation techniques are often specific to obfuscation transformations. For example, the work of Udupa, Debray and Madou [187] targets control-flow transformations, whereas others [17, 129, 150, 185] aim at removing opaque predicates. Generic deobfuscation techniques, however, make no assumption about the applied protections [167, 203]. These techniques are based on dynamic symbolic execution and may lack in code coverage and scalability.

Though obfuscation transformations are semantic-preserving, they may introduce side effects to the code [45]. Each transformations has its own construction methodology, thus specific patterns. Recent works try to tackle the detection of software protections using machine learning or deep learning techniques. Ugarte-Pedrero *et al.* [188] propose a semi-supervised learning approach in order to classify packed and unpacked binaries. Sun *et al.* [182], and more recently Biondi *et al.* [20], aim at detecting and identifying packers using machine learning techniques. In the previous chapter [184], we propose a deobfuscation methodology for invariant opaque predicates based on machine learning techniques.

From the variety of obfuscation techniques, as well as deobfuscation methodologies, the ability to efficiently detect the software protections used is at a prime. To that end, the recent work of Salem and Banescu [166] focuses on the detection of obfuscation transformations. Their goal is to facilitate the selection and application of adequate deobfuscation techniques. To the best of our knowledge, their work is the first to tackle code obfuscation detection using machine learning. However, their methodology is also prone to some limitations as explained next.

5.1.1 Current limitations

Existing detection technique for code obfuscation [166] based on machine learning techniques comes with the following limitations:

1. *Syntax reasoning*: detecting obfuscation transformations often reasons about the syntax of the targeted code. They are either based on the use of metrics or disassembly code. However, the syntax may provide limitations such as architecture-dependency or implementation dependencies, thus lowering the accuracy of classification models.

2. *Code dependency*: machine learning and syntax-reasoning used for the detection of obfuscation transformations can lead to code dependency. Namely, the trained model becomes dependent to the analyzed code used in the training set, thus lowering its accuracy.

3. *Multi-class problem*: the methodology used relies on multi-class problems for classification. Namely, they consider that one binary cannot be obfuscated with more than one obfuscation transformation. However, transformations can be combined, thus the necessity to be able to detect the several applied layers.

4. *Granularity*: the detection technique has a high-level of granularity. They may detect an obfuscation transformation, but they do not focus on their constructions types. The latter is of importance in order to decide which analysis to apply on obfuscated code. Many transformations constructions are made to prevent existing deobfuscation techniques.

5. *Mono-models*: the detection technique mainly use a single trained model. More advanced classification techniques, such as ensemble-learning models, may provide a better accuracy for classification problems [119, 160]. We study this approach for the specific task of obfuscation transformation detection.

5.1.2 Motivation

When applying obfuscation transformations for software protections, stealth is sometimes not desired. Many applications aim for dissuasion in order to prevent reverse-engineering. In any case, the goal of our methodology is to provide a static and automated framework to help reverse-engineers. By detecting obfuscation transformations, and more specifically their constructions, an analyst will save an important amount of time. The selection of the deobfuscation process to apply requires such knowledge beforehand. A motivating example is illustrated in Figure 5.1. It represents the obfuscated control-flow graph of a quick-sort function. Based on the previously introduced problems, our goal is to answer the following questions:

- *Complexity:* can we detect all applied layers of obfuscation transformation?

- *Granularity:* can we detect the constructions of applied obfuscation transformations?
- *Efficiency:* can we create accurate and generic enough models for unknown data?

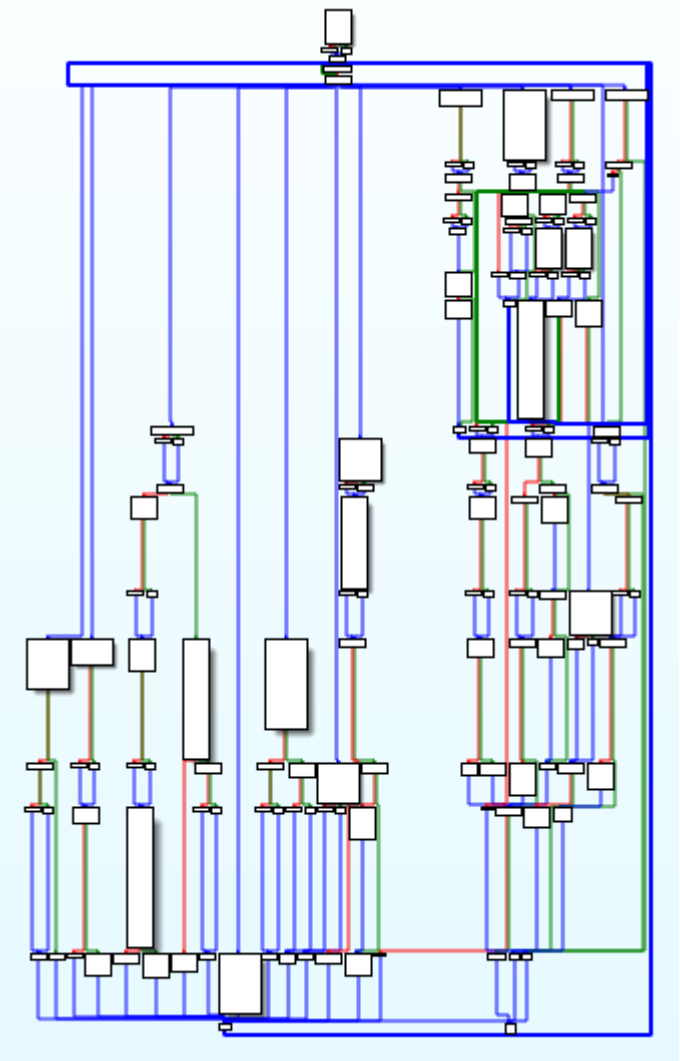

Figure 5.1: Control-flow graph of a quick-sort function obfuscated using several `Tigress` transformations.

As an example, several opaque predicates constructions prevent SMT-solver based deobfuscation techniques [200]. Other recent works prevent the application of dynamic symbolic execution techniques [14**?**]. Thus, knowing which transformations and constructions analysts are facing may prevent using unadapted techniques for the deobfuscation process.

5.1.3 Contributions

In order to face the above listed limitations and provide answer to our motivating questions, we bring the following contributions:

1. A novel methodology that combines semantic reasoning with ensemble learning techniques. We believe that semantic reasoning will prevent our model from code dependency limitations.
2. The application of a multi-label and multi-output ensemble model. This classification approach provides us with the ability to detect several combined layers of obfuscation transformations.
3. An extension of our methodology for a fine-grained detection. Based on our main approach, a second classification model is used for the detection of the transformations constructions, based on a multi-class classification model.
4. Several studies and experiments that justify the constructions of our methodology. We compare different machine learning approaches and techniques in order to build efficient and scalable models.
5. The evaluation of our methodology against state-of-the-art obfuscators such as `Tigress` [43] and `Obfuscator-LLVM` [96] (*i.e.* `OLLVM`).

This chapter is organized as follows: in Section 2 we present the background information about code obfuscation and targeted transformations. We also introduce related work, as well as notions of supervised machine learning. In Section 3, we describe our methodology which combines semantic reasoning with ensemble learning. Section 4 contains our studies and experiments towards an efficient implementation of our methodology. Section 5 illustrates our evaluations on state-of-the-art and publicly available obfuscators. Section 6 briefly discuss the application of our methodology to setup deobfuscation strategies. Then, we discuss our design limitations in Section 7, as well as our perspectives in Section 8. Finally, we conclude in Section 9.

5.2 Background

In this section we briefly introduce several notions related to supervised machine learning. We also present related work to our contribution, namely *metadata recovery attacks*.

5.2.1 Classification algorithms

As previously discussed in Chapter 4, the choice of which specific learning algorithm to use is a critical step. Many classification algorithms exist [91], each of them having different mapping functions. Classification is a common application of machine learning. As such, there are many

metrics that can be used to measure and evaluate our models. In order to compute these metrics, *k-Fold Cross-Validation* [107] is a frequently used technique.

The definition of k-fold cross-validation consists in reserving a particular set of samples on which the model does not train. The limited set of samples allows to estimate how the model is expected to perform on data not used during the training phase. The parameter k refers to the number of groups that a given dataset of samples is split into, in order to calculate the mean of our models *accuracy* as well as the *F1-score* based on the value of k. While the accuracy of the model represents the ratio of correctly predicted labels to the total of labels, F1-score takes both false positives and negatives into account. In our experimentations and evaluations, the accuracies and F1-scores are calculated using *k-fold cross-validation*, with $k = 10$ for a better generalization of our model to unknown instances. Another application of cross-validation, introduced in [166], consists in a functionality-based folding. In other words, the learning set and training set are divided based on the functionality of the samples from which the raw data is generated. The goal of such an evaluation methodology is to measure if the model is dependent from the underlying code functionality, independently of the obfuscation transformation applied. The next paragraph introduce furthermore the work of Salem and Banescu [166], known as *metadata recovery attack*.

5.2.2 Metadata recovery attack

Salem and Banescu [166] introduce the use of machine learning techniques to evaluate the stealth of obfuscation transformations throughout their detection (otherwise call *metadata recovery attack*). Their primary hypothesis is that machine learning techniques are capable of implementing these attacks by classifying obfuscated programs according to the transformations applied. Their experiments are based on two learning algorithms, namely Naive Bayes [72] and Decision trees [161]. Their raw data is based on static disassembly or dynamic instruction traces, either stripped or not. Thus, we refer to such raw data generation as *syntax-reasoning*. The evaluation of their models is made with two classification techniques. The first one is a traditional k-fold cross validation, with $k = 10$. The second one is more fine-tuned since it discriminates the training and test dataset on program functionality. In other words, the test dataset is excluded of any raw data that have been used in the training dataset, based on the functionality they implement. Such process is also repeated 10 times, to calculate the average accuracy for each fold. Their results are promising, showing up to 100% of accuracy for obfuscation transformations detection with decision trees, on dynamic traces. However, these results are obtained with the conventional cross-validation, whereas the second classification mode provides lower results (up to 61% of accuracy) with decision trees. This indicates that their model is dependent of the functionality implemented in their raw data. Moreover, their work is not implemented yet to cover several layers of obfuscation transformations, as it can be the case in most obfuscated programs.

Our goal for this contribution is to combine semantic reasoning and more advanced machine learning classification techniques. We want to have a static analysis tool, based on symbolic execution, in order to have a model that does not depend on the functionality of the program. The models are used to detect several layers of obfuscation transformations, thus having a multi-label and multi-output classification problem. Then, we extend our detection not only to the obfuscation transformations but also to their constructions. To this end, in the next section, we present our approach and methodology.

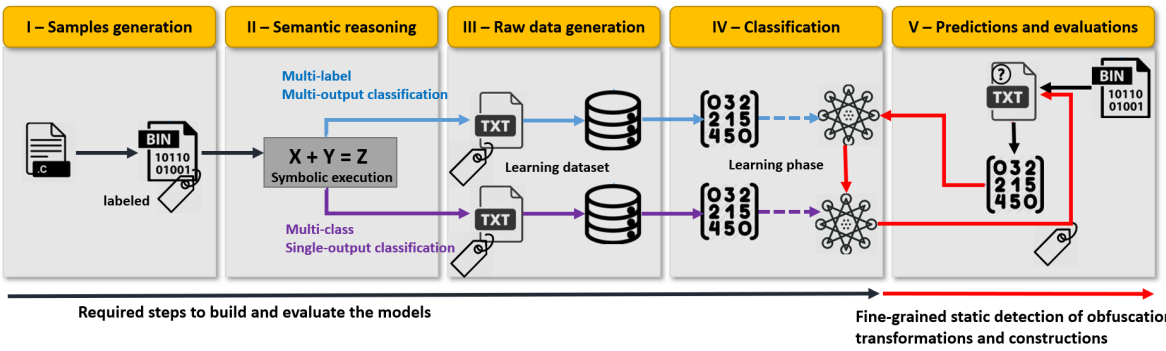

Figure 5.2: Design steps for fine-grained static detection of obfuscation transformations and constructions.

5.3 Methodology

In this section we present our methodology composed of several steps, as illustrated in Figure 5.2. **I.** In order to create our models, we need to generate obfuscated as well as clean samples. The generation of our obfuscated sample is done using publicly available obfuscators, namely `Tigress` and `OLLVM`. **II.** We then employ semantic reasoning via symbolic execution to extract our raw data, from the generated samples. This step is presented in Section 5.3.1. **III.** We create two different datasets for two different kinds of classifications. Using labeled raw data, we build our datasets for the detection of obfuscation transforms, including several combinations. Another dataset is made for the detection of specific constructions related to the transformations. These steps are introduced in Section 5.3.4. **IV.** The previous datasets are used to train our models. In order to select the most relevant approach and learning algorithms, several studies and experiments are provided in Section 5.4. **V.** The final step consists in their evaluation and their application on unknown instances, as presented in Section 5.5.

5.3.1 Semantic reasoning

Static symbolic execution is a binary analysis technique that captures the semantics (*i.e.* logic) of a program. An interpreter is used to trace the program, while assuming symbolic values for inputs rather than obtaining concrete values as a normal execution would. A symbolic state $\mathbb{S}$ is built and consists in a set of symbolic expressions S for each variables (*i.e.* registers, memory, flags, etc.). Several techniques exist for symbolic execution [12]. Since static symbolic execution is prone to limitations (*e.g.* path explosion), we use an intra-procedural and bloc-centric approach, as summarized next.

5.3.1.1 Bloc-centric intra-procedural symbolic execution

We use semantic reasoning for the generation of our *raw data*. The symbolic representation helps to efficiently detect obfuscation transformations and constructions. Raw data refers to the representation of data samples, containing noisy features, which need to be processed in order to extract the informative characteristics to train the models. For the detection of obfuscation transformations, we choose to work on disassembled functions of binary code. On these functions, we apply static symbolic execution to retrieve their semantic representation. In our work we use disassembled functions to collect the symbolic expressions from the code, as illustrated in Algorithm 6. First, the semantic reasoning part of our methodology is given a disassembled function F as input. For the learning phase of our methodology, F needs to be labeled. In other words, we need to know which transformations are applied in order to properly train our model. However, in order to use our methodology as a static and automated detection framework, F does not require to be labeled once the models are trained. Based on F, we iterate over each basic block B. We then collect the instructions of B, denoted by I_B, with the function `getInstructions()`. I_B is translated into an intermediate language, denoted by IR_B, using `getIntermediateLanguage()`. Finally, IR_B is being used for the bloc-centric symbolic execution function `symbolicExecution()`. The latter will return the symbolic state S_B, in other words, expressions of each modified variables in a static single assignment form, based on the intermediate representation IR_B previously used. The generated semantics S_B is then normalized using `normalizeSemantics()` function. Finally, the normalized semantics NS_B is added to the dictionary L containing all normalized semantics for each processed basic block B. The content of L will be used to generate our raw data as text file. Our normalization step has the crucial role of making the model scale to unknown data. Next, Section 5.3.2 describes this step, along with the content of our raw data.

Algorithm 6 semantic reasoning for raw-data generation

```
1: procedure SEMANTIC REASONING(F: a disassembled function)
2:     Initialize a dictionary L
3:     for each basic block B in F do
4:         I_B ← getInstructions(B)
5:         IR_B ←getIntermediateLanguage(I_B)
6:         S_B ←symbolicExecution(IR_B)
7:         NS_B ←normalizeSemantics(S_B)
8:         L[B] ← NS_B
9:     end for
10:    textFile = generateRawData(L, F)
11:    return textFile
12: end procedure
```

5.3.2 Semantic-based raw data

Intermediate representations often use concrete values within their generated expressions. This causes raw data to depend on addresses that are specific to some binaries and prevents our models to scale on unknown data. Some intermediate representations also use identifiers in order to express modified registers or memory areas. This notation may further affect the scalability of our trained models. For the purpose of having a model that can scale to unknown data we use a normalization phase. The normalization consists in replacing all identifiers and concrete values by symbols, and non-alphanumerical characters by alphanumerical words. This is a necessary step for a complete features extraction phase that sometimes excludes non-alphanumerical characters when working on text-based raw data. In our methodology, we generate the raw data using the `Miasm2` [57] intermediate language. This language is part of the symbolic execution engine that we use for the implementation of our methodology as `IDA Pro` plug-in. Additionally, `Miasm2` intermediate language has also been successful for the application of machine learning techniques in order to deobfuscate opaque predicates [184].

Listing 1 illustrates the symbolic state S of the first basic-block of the function quick-sort, which is illustrated in Figure 5.1. Note that the complete raw data will contain the symbolic states of each basic-blocks of the quick-sort function. We can see that `Miasm2` intermediate language uses several keywords to express the semantics of the basic blocks. For example, `ExprId` is used for registers and `ExprInt` for concrete values. The registers and concretes values prevent our model from scaling to unknown data, thus potentially lowering our model accuracy. This underlines the necessity to normalize the intermediate language for an efficient semantic reasoning. Listing 30 illustrates the same basic-block symbolic state, but normalized.

```
ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffd0
    , 64)), size=64) = ExprId('RDI_init', size=64)

ExprId('af', size=1) = ExprSlice(ExprOp('^', ExprOp('+', ExprId('RSP_init',
     size=64), ExprInt(0xfffffffffffffc8, 64)), ExprOp('+', ExprId('
    RSP_init', size=64), ExprInt(0xfffffffffffffff8, 64)), ExprInt(0x30,
    64)), 4, 5)

ExprId('RBP', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xffffffffffffffff8, 64))

ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffc8
    , 64)), size=32) = ExprSlice(ExprId('RDX_init', size=64), 0, 32)

ExprId('pf', size=1) = ExprOp('parity', ExprOp('&', ExprMem(ExprInt(0
    x606078, 64), size=64), ExprInt(0xff, 64)))

ExprId('RAX', size=64) = ExprMem(ExprInt(0x606078, 64), size=64)

ExprId('IRDst', size=64) = ExprCond(ExprMem(ExprInt(0x606078, 64), size=64)
    , ExprInt(0x40064b, 64), ExprInt(0x400644, 64))

ExprId('zf', size=1) = ExprCond(ExprMem(ExprInt(0x606078, 64), size=64),
    ExprInt(0x0, 1), ExprInt(0x1, 1))

ExprMem(ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0xfffffffffffffff8
    , 64)), size=64) = ExprId('RBP_init', size=64)

ExprId('of', size=1) = ExprInt(0x0, 1)

ExprId('nf', size=1) = ExprSlice(ExprMem(ExprInt(0x606078, 64), size=64),
    63, 64)

ExprId('cf', size=1) = ExprInt(0x0, 1)

ExprId('RSP', size=64) = ExprOp('+', ExprId('RSP_init', size=64), ExprInt(0
    xfffffffffffffc8, 64))

ExprId('RIP', size=64) = ExprInt(0x400650, 64)

ExprId('IRDst', size=64) = ExprInt(0x400650, 64)
```

Listing 5.1: Symbolic state using Miasm2 intermediate language

```
ExprMem(ExprOp(op+, REG0, v0), size=64) = REG1

REG2 = ExprSlice(ExprOp(op^, ExprOp(op+, REG0, v1), ExprOp(op+, REG0, v2),
    v3), 4, 5)

REG3 = ExprOp(op+, REG0, v2)

ExprMem(ExprOp(op+, REG0, v1), size=32) = ExprSlice(REG4, 0, 32)

REG5 = ExprOp(opparity, ExprOp(op&, ExprMem(v4 ,size=64), v5))

REG6 = ExprMem(v4, size=64)

IRDst = ExprCond(ExprMem(v4, size=64), v7, v8)

REG8 = ExprCond(ExprMem(v4, size=64), v9, v10)

ExprMem(ExprOp(op+, REG0, v2), size=64) = REG9

REG10 = v9

REG11 = ExprSlice(ExprMem(v4, size=64), 63, 64)

REG12 = v9

REG13 = ExprOp(op+, REG0, v1)

REG14 = v11

IRDst = v11
```

Listing 5.2: Symbolic state using our normalized Miasm2 intermediate language

Additionally, the normalization step also reduces the size of the raw data. This helps enhancing the efficiency of learning and testing phase in terms of execution time. The next sections will present the different machine learning techniques used in our methodology. The purpose is to create automated and efficient models for the detection of obfuscation transformations, as well as their constructions.

5.3.3 Ensemble learning

In machine learning, ensemble methods [59] use multiple learning algorithms. They are mostly used to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone [119, 160]. An *ensemble*, in this case, consists of a set of individually trained classifiers whose predictions are combined when processing novel instances. Different families of ensemble learning methods exists, e.g. Bagging [28], Boosting [69, 70] or Stacking [177]. Since every model has its strengths and weaknesses, ensemble models combine individual models to help cope with the weaknesses of each algorithms.

In order to select the best possible predictions from our ensemble, we use a *voting* [181] algorithm. A voting classifier simply aggregate the predictions of each classifier and predict the class that gets the most votes. Also nown as *hard voting*, this approach is usually used for classification problems. The bagging approach (*i.e. bootstrap aggregating* approach) consists in using the same training algorithm for each model, but the training is done on different random subsets of the training dataset. The booster approach consists in training the models sequentially, each trying to correct its predecessor. A generic overview of ensemble learning for supervised classification is given in Figure 5.3.

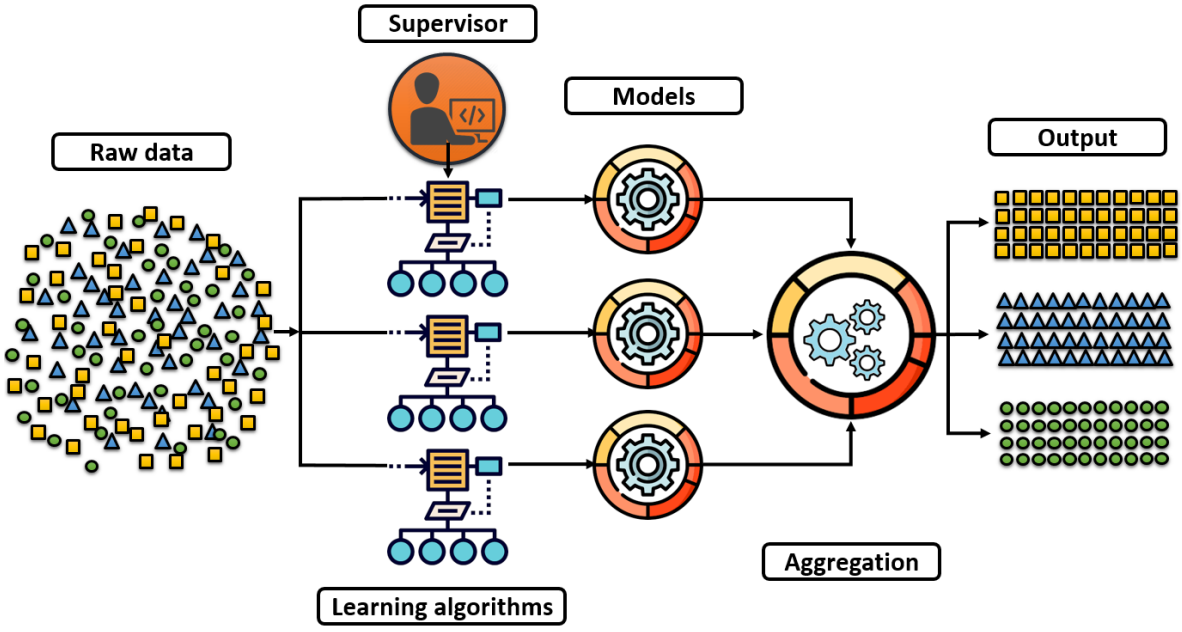

Figure 5.3: Generic overview of a supervised ensemble learning scheme.

Our work emphasizes the benefits of the ensemble learning approach, as opposed to individual models. Thus, **we based our approach on voting classifiers**. However, a more in-depth studies of

other approaches could provide better results in terms of accuracy or execution time.

5.3.4 Multi-label and multi-class classifications

As discussed previously, several classification problems exists. Multi-label classification methods for example, are increasingly required by modern applications [26, 116]. We use multi-label with multi-output classification, in order to return all the detected obfuscation layers, specially when combined. We also focus on multi-class classifications. These approaches are important in our methodology because:

1. the detection of all the applied obfuscation transformations is a **multi-label classification problem**. For example, if our set of labels are the applied transformations, namely control-flow flattening and code virtualization, then one binary can have both protections. In such case, our methodology needs to return all predicted labels. We then refer to such model as a **multi-output classification**.

2. the fine-grained detection of the constructions is a **multi-class classification problem**. For example, if we know that control-flow flattening is applied on a code, then its constructions can only be one unique label (e.g. switch-based, ifnest-based, indirect, call-based, etc.).

Multi-label classification methods differ from binary or multi-class approaches. Tsoumakas and Katakis [186] group multi-label classification methods into two categories: *problem transformation methods* that transform the multi-label classification problem either into one or more single-label classification problems, and *algorithm adaptation methods* that extend specific learning algorithms in order to handle multi-label data directly. **In our methodology we use classifier chains** [154], where each model is an ensemble of learning algorithm, as presented in Section 5.3.3. We also study the *binary relevance* methodology [78] in Section 5.4. These two methodologies are briefly introduced in the following paragraphs.

5.3.4.1 Problem transformation methods

Binary relevance method [208] is one problem transformation technique that transforms any multi-label problem into one binary problem for each label. Hence, it trains several classifiers, one for each class, *i.e.* one per obfuscation transformations. The union of all classes that are predicted is taken as the multi-label output. Binary relevance method is popular because of its ease of implementation. However, the main drawback is that it ignores the possible correlations between labels.

Classifier chains [155] however, as opposed to binary relevance method, take into account the labels correlations. With this methodology we have for n labels also n binary classifiers $f_0, f_1, ..., f_n$

constructed. The construction is made as a chain where a classifier f_i uses the predictions of all its previous classifiers f_j with $j < i$. The chain order is randomly selected in our design.

Another known problem transformation method is called *Label Powerset* (or *Label Combination*) which considers each combination of labels as a single label. In our work we do not study this approach because of its high computational complexity due to the possible combinatorial explosion. Table 5.1 provides an overview of the above listed problem transformation methods based on the work of Ceylan and Pekel [38].

Method	Computational complexity	Advantage(s)	Disadvantage(s)
Binary Relevance	$O(n)$	Computationally efficient. Simple and fast.	Does not consider the relationship among labels.
Classifier Chains	$O(n)$	High predictive performance. Works with any type of classifier.	Cannot use unlabeled data.
Label Powerset	$O(2^n)$	Considers relationship among labels	Computationally expensive and complex. May lead to over-fitting.

Table 5.1: Comparison of some problem transformation methods for multi-label classification

The core of our methodology is based on Classifier Chains because of its computational complexity and advantages.

5.3.4.2 Algorithm adaptation methods

Algorithm adaptation extends single label classification to the multi-label context. It is usually done by changing the decision functions. Some learning algorithms support multi-label and multi-output classification (*e.g.* [209, 210]), whereas other can be extended.

During our experiments, these two classifications approaches, and multi-label problems will be studied in Section 5.4. Our objective is to provide the best suited algorithms and techniques for an efficient and accurate model.

5.4 Experiments

In this section we present first the dataset used, common with previous related work [166, 184]. Our preliminary studies towards an efficient implementation of a fine-grained detection framework are also introduced. All our experiments and evaluations are done on a `Windows` 7 laptop, using 16GB of RAM, and an `Intel` processor.

5.4.1 Datasets

Our experiments are made on several `C` code samples. We use the `scikit-learn` API [145] for the implementation of the models. The datasets contain various types of code, each of them having different functionalities in order to have models that do not fit to a specific type of program. The used samples are listed below:

- GNU core utilities (*i.e.* core-utils) binaries [151] for normal predicate samples;
- Cryptographic binaries for obfuscated and non-obfuscated predicates [49];
- Samples from [13] containing basic algorithms (*e.g.* factorial, sorting, etc.), non-cryptographic hash functions, small programs generated by `Tigress`;
- Samples involving the uses of structures and aliases [4, 88].

Our choice is motivated by the samples low ratio of dependencies and their straightforward compilation. This makes their obfuscation possible using tools such as `Tigress` and `OLLVM` without errors during compilation. *Furthermore, all datasets used for the studies and evaluations are balanced and contain between 1000 to 5000 samples.* The obfuscation transformations applied are given in Appendix B.2 and B.1. The next section will present our studies based on these datasets.

5.4.2 Preliminary studies

Our goal in this section is to provide some answers to the following questions related to our methodology:

- **Study 1**: when only one obfuscation transformation is applied, is a single model more effective than ensemble models for the detection?
- **Study 2**: when several obfuscation transformations are applied, can the model from Study 1 be applied to the multi-label and multi-output classification problems?
- **Study 3**: when several obfuscation transformations are applied, is a multi-label and multi-output model more efficient than one binary model for each transformation, *i.e.* classifier chains?
- **Study 4**: for the fine-grained detection of obfuscation constructions, is a single model more efficient than ensemble models?

Our different studies and evaluations present two different types of results based on two different evaluations approaches. One is the traditional k-folds cross-validation with scores in black colored font. The other is made with the functionality-based cross-validation approach in red colored

font, used in Salem and Banescu related work [166]. Besides, we use as a traditional single-model random-forest algorithm throughout all our studies. As for the ensemble models, we combined extra-tree and random-forest learning algorithms. These algorithms were selected because they provided the best scores in terms of accuracy, precision and recall. For simplicity, a preliminary evaluation was made between several learning algorithms [108] (*e.g.* decision trees, k-nearest neighbors, support vector machines, neural network, naive Bayes, random forest, etc). In order to select the best ensemble models, we combined between 2 to 6 single models, and selected the combination that provided the best scores.

5.4.2.1 Study 1

In this study we experiment traditional models against ensemble learning for multi-class classification problems. Namely, each sample is assigned with *an unique* label. Thus our model returns only one label per sample. We experiment here if ensemble learning can be more efficient at detecting obfuscation transformation, when only one layer is applied. Therefore, we do not combine obfuscation transformations for this study. Table 5.2 illustrates our results where we see that ensemble-learning

Obfuscation transformation	**Mono-model**	**Ensemble-learning**
Tigress transformations	**Random-forest**	**Extra-tree & Random-forest**
EncA	93% / 98%	95% / 100%
EncL	100% / 97%	100% / 100%
EncD	95% / 98%	95% / 100%
AddO	100% / 100%	98% / 100%
Flat	97% / 100%	97% / 100%
Virt	100% / 100%	100% / 100%
Jit	100% / 100%	100% / 100%
clean	91% / 100%	91% / 100%
Overall Accuracy	97% / 99%	97% / 100%

Table 5.2: Multi-class accuracy and F1-scores per labels for the detection of `Tigress` obfuscation transformations (1 layer).

provides a similar accuracy to random-forest, up to 97%, with traditional cross-validation. The illustrated F1-scores per labels, namely the obfuscation transforms, also points out that most of them are predicated similarly with both approaches. An exception is made for arithmetic encoding, *i.e. EncA*, and opaque predicates, *i.e. AddO*. With the functionality-based cross-validation approach however, the results differs more as observed in red font. Ensemble-learning technique provides 100% accuracy and F1-score for each classes, whereas random-forest achieves slightly lower results, with an average accuracy at 99%. Due to the semantic reasoning of our methodology, the results are better with this approach when having one layer of obfuscation. Yet, these results are not sufficient

to select traditional mono-models over ensemble-learning, or the opposite way. Hence, the next study will experiment these two approaches for multi-label and multi-output classification.

5.4.2.2 Study 2

In the following study, we combine all obfuscation transformations. The goal of our model is to give n correct output for each n obfuscation transformation applied. Each sample can have one or more labels. We aim to compare the random-forest algorithm with the ensemble model based on random-forest and extra-trees for multi-label and multi-output classification.

Our results in Table 5.3 illustrate that traditional cross-validation provides a higher overall accuracy for ensemble learning classifier as opposed to random forest. Our ensemble of models scores 92% as opposed to 90% for random-forest, with F1-scores per labels above 91%. The functionality-based cross-validation provides lower results, with an overall accuracy at 83% and at 82% for respectively random forest and ensemble models. Still, our result indicates that both approaches can efficiently detect several layers of obfuscation transforms. However, we may improve our results using problem transformations methods such as classifier chains.

Obfuscation transformation	**Multi-label mono-model**	**Multi-label ensemble**
Tigress transformations	**Random-forest**	**Extra-tree & Random-forest**
EncA	95% / 93%	96% / 92%
EncL	90% / 78%	92% / 85%
EncD	95% / 93%	96% / 92%
AddO	96% / 88%	97% / 88%
Flat	98% / 97%	99% / 91%
Virt	99% / 98%	99% / 99%
Jit	100% / 95%	97% / 92%
clean	90% / 90%	91% / 87%
Overall Accuracy	90% / 83%	92% / 82%

Table 5.3: Multi-label accuracy and F1-scores per labels for the detection of `Tigress` obfuscation transformations (several layers).

The next study will experiment this hypothesis.

5.4.2.3 Study 3

As in the second study, we combine all obfuscation transformations but we use binary classification problem for multi-label and multi-output classification using classifier chains. Our results with standard cross-validation does not differ much from previous Study 2 as illustrated in Table 5.4. The functionality-based cross-validation provides improved overall accuracies and F1-scores per

Obfuscation transformation	Mono-model chain	Ensemble chain
Tigress transformations	**Random-forest**	**Extra-tree & Random-forest**
EncA	95% / 92%	97% / 90%
EncL	90% / 80%	93% / 87%
EncD	95% / 92%	97% / 96%
AddO	96% / 92%	97% / 88%
Flat	97% / 97%	99% / 91%
Virt	99% / 98%	99% / 99%
Jit	100% / 90%	100% / 92%
clean	88% / 90%	92% / 90%
Overall Accuracy	90% / 85%	92% / 90%

Table 5.4: Classifier chain accuracy and F1-scores per labels for the detection of `Tigress` obfuscation transformations (several layers).

labels. Ensemble models used in classifier chains provide 90% of overall accuracy, compared to random-forest used in classifier chains that score 85% of overall accuracy. This study led us to select ensemble-learning techniques with classifier chains in our methodology since classifier chains allow us to create an efficient and accurate model for the detection of obfuscation transformations with one or more layers.

5.4.2.4 Study 4

For this final study, our goal is to evaluate the models for the fine-grained detection of an obfuscation transformation construction.

Code virtualization	Mono-model	Ensemble model
Tigress constructions	**Random-forest**	**Extra-tree & Random-forest**
linear-based	100% / 99%	100% / 100%
switch-based	100% / 98%	100% / 100%
if-nest-based	100% / 100%	100% / 100%
Overall Accuracy	100% / 99%	100% / 100%

Table 5.5: Accuracy and F1-scores per labels for the detection of Virtualized constructions.

We use in our dataset several Virtualized samples with `Tigress` for our experiment. `Tigress` allows the user to select different kinds of constructions, such as *switch-based, ifnest-based, linear-based, interpolation-based* for example. This experiment is equivalent to Study 1 in the sense that it is a multi-class classification problem. Namely, each sample has a unique label and the selected model will return one unique label per instance.

Our results in Table 5.5 show that both random-forest and ensemble models provides the same F1-scores per labels. Their overall accuracies with standard cross-validation are also with 100% accuracy. With functionality-based cross-validation, ensemble models are more efficient with a 100% accuracy as opposed to 99% for mono-model based on random-forest. This led us to select ensemble models in our methodology also for the classification of constructions, as it allows a fined-grained detection capability.

5.5 Evaluations

In this section we evaluate our models with respect to the following classification problems:

1. *Multi-label and multi-output evaluation:* can our model, based on a classifier chain of ensemble models, efficiently and accurately detect all obfuscation transformations when one or more layers are applied?

2. *Multi-class evaluation:* once the obfuscation transformation detected, can our ensemble model efficiently and accurately detect the construction of the latter?

We use both cross-validation evaluation schemes as detailed in Section 5.2.1. Our evaluations are made with publicly available obfuscators, namely `Tigress` and `OLLVM`, in order to combined obfuscation transformations from different tools.

5.5.1 Transformations detection

First, our goal is to evaluate the stealth of obfuscation transformation, either applied as unique layer or combined. We use our multi-label and multi-output model based on ensemble-models and classifier chain to detect all the transformations applied. To measure the efficiency of our model, we used both traditional and functionality-based cross-validation as explained in Section 5.2.1. A list of all combinations of the applied transformations used in our evaluations can be found in Appendix B.1 and B.2. Additionally, command line options for `Tigress` and `OLLVM` are given in Appendix A.1 and B.2.

5.5.1.1 OLLVM

Our first evaluation uses `OLLVM`. It implements transformations such as opaque predicates (*i.e.* bogus control flow, *bcf*), instruction substitutions (*i.e. sub*) and control-flow flattening (*i.e. fla*). We built a dataset with several combinations of these transformations (*c.f.* Appendix B.2) in order to measure the efficiency of our model.

Obfuscation transformation	Classifier Chain
OLLVM	**Ensemble model**
bcf	98% / 98%
fla	92% / 95%
sub	82% / 80%
clean	94% / 93%
Overall Accuracy	86% / 89%

Table 5.6: Evaluated accuracy and F1-scores per labels for the detection combined `OLLVM` transformations.

Table 5.6 shows our results. Our model achieves an overall accuracy of 86% with traditional cross-validation and 89% with the functionality-based one. F1-scores for labels *bcf*, *fla* and non-obfuscated samples marked as *clean*, are over 92% and up to 98% for *bcf*. However, the efficiency of our model to detect `OLLVM` instructions substitutions transformations, labeled as *sub*, achieves a low F1-score at 80%. Further evaluations indicate that *sub* is often considered *clean* by our model. Thus, when combined with other transformations, *sub* transformation is often undetected.

5.5.1.2 Tigress

Our second evaluation is done with the `Tigress` obfuscator. `Tigress` can generate state-of-the-art transformations such as dynamic-code generation (*i.e. Jit*), code-virtualization (*i.e. Virt*), control-flow flattening (*i.e. Flat*), opaque predicates (*i.e. AddO*) and several encoding (*i.e. Arithmetics, Literals* and *Data*, respectively *EncA, EncL* and *EncD*), among others.

Obfuscation transformation	Classifier Chain
Tigress	**Ensemble model**
EncA	94% / 90%
EncL	90% / 86%
EncD	92% / 91%
AddO	95% / 96%
Flat	96% / 98%
Virt	99% / 100%
Jit	100% / 100%
clean	91% / 89%
Overall Accuracy	90% / 91%

Table 5.7: Evaluated accuracy and F1-scores per labels for the detection combined `Tigress` transformations.

As illustrated in Table 5.7, our model accuracy is up to 90% with standard cross-validation.

With functionality-based cross-validation, the overall accuracy is at 91%. F1-scores for heavy transformation such as *Virt* and *Jit* are up to 99% and 100%. The lowest F1-score is for *i.e. EncL* which is sometimes considered as a *clean* sample by our model. Regardless, our evaluation underlines the accuracy and efficiency of our methodology against `Tigress` transformations.

5.5.1.3 OLLVM and Tigress

Our final evaluation combines the datasets for both `OLLVM` and `Tigress`. We aim to see if our model is able to detect common obfuscation transformations. Table 5.8 shows our results. F1-scores

Obfuscation transformation	Classifier Chain
Tigress and OLLVM	**Ensemble model**
EncA and sub	93% / 90%
EncL	88% / 88%
EncD	90% / 88%
AddO and bcf	95% / 95%
Flat and fla	96% / 99%
Virt	99% / 100%
Jit	100% / 100%
clean	83% / 80%
Overall Accuracy	88% / 86%

Table 5.8: Evaluation accuracy and F1-scores per labels for the detection of both combined `Tigress` and `OLLVM` transformations.

for heavy transformations such as *Virt*, *Jit* and *Flat* are high, averaging up to 100% for *Jit* as an example. Combined test samples between obfuscators such *EncA-sub*, *AddO-bcf*, and *Flat-fla* have high F1-scores, even when combined with other transformations. These heavy transformations introduce important side-effects to the code, allowing an efficient and accurate detection of our model. The ability to efficiently detect non-obfuscated samples is still low compared to the ability to detect all layers of obfuscation transformations. In that case, our model F1-scores are up to 83% and 80% depending on the cross-validation approach used. Still, our model is averaging an accuracy up to 88% and 86%. These overall accuracies illustrate our model efficiency regarding the detection of obfuscation transformations, even when combined, and between the two different obfuscators. The execution time average approximately two minutes for a balanced dataset of 4000 raw data.

5.5.1.4 OLLVM vs. Tigress

Our final evaluation aims to compare the accuracies of our model depending on the learning dataset used. First, we use a learning dataset only based on `OLLVM` transforms. The model will be then

evaluated against some similar obfuscation transformations generated by `Tigress`. Second, we do the opposite, namely train our model on `Tigress` samples to evaluate it on `OLLVM` raw data. The results are displayed in Table 5.9. As we can see, our model efficiently detects `Tigress` *Flat*

Training dataset	Testing dataset	Overall accuracy
OLLVM	Tigress (Flat)	100% / 100%
OLLVM	Tigress (Flat, AddO)	68% / 61%
Tigress	OLLVM (fla)	95% / 92%
Tigress	OLLVM (all)	82% / 75%

Table 5.9: Overall accuracies of our model using either `OLLVM` or `Tigress` learning dataset.

transformation when training on 1000 samples of all `OLLVM` transforms, with 100% of accuracy. Results are lower when the training dataset is based on `Tigress` (4000 samples), against `OLLVM` *fla* transform, with an overall accuracy up to 95% with a standard cross-validation. Moreover, we can observe that our model cannot efficiently detect `Tigress` opaque predicates, *i.e. AddO*, when training only on `OLLVM` transforms. The results, in that case, indicates that our model efficiently detects the *Flat* transformation, but only few *AddO* ones. Finally, when our model is trained on `Tigress`, the overall accuracy is up to 82% against all OLLVM transforms (c.f. Appendix B.2). This result indicates that our methodology provides some genericity, while still having room for improvement.

5.5.2 Constructions detection

In this section we evaluate our model for the detection of specific obfuscation transformations constructions. We use our multi-class model, based on ensemble-models, to provide a fine-grained detection technique. As for previous evaluations, we use traditional and functionality-based cross-validation techniques. A first evaluation on code virtualization is already presented in Study 4, Section 5.4.2.4. In the followings we focus on control-flow flattening and opaque predicates constructions.

5.5.2.1 Control-flow flattening

As for code virtualization, control-flow flattening can also be constructed in several ways, as introduced in Section 2.1.4.

Facing the same limitations as for code virtualization constructions, we evaluated two constructions namely *switch-based* from the `Tigress` obfuscator, and *ifnest-based* from `OLLVM`. The evaluation results are in Table 5.10. Our model averages high F1-scores and accuracy, the latter being at 98% with standard cross-validation evaluation.

Control-flow flattening	Ensemble model
Tigress and OLLVM	**Extra-tree & Random-forest**
switch-based	98% / 95%
if-nest-based	98% / 100%
Overall Accuracy	98% / 97%

Table 5.10: Evaluation accuracy and F1-scores per class for the detection of control-flow flattening constructions.

5.5.2.2 Opaque predicates

Many opaque predicates constructions exists (cf. Chapter 2, Section 2.1.4). For the detection of their constructions, we used `Tigress`, `OLLVM` but also novel `bi-opaque` constructions [200]. Our results in Table 5.11 show that our model is accurately detecting opaque predicates constructions. F1-scores are up to 100% with standard cross-validation. Bi-opaque constructions are however often un-detected when combined with other transformations.

Opaque predicates	Ensemble model
Tigress and OLLVM	**Extra-tree & Random-forest**
Floats	85% / 89%
Symbolic-memory	87% / 87%
Arithmetic	100% / 100%
Aliasing	100% / 99%
Mixed-boolean and arithmetic	100% / 96%
Overall Accuracy	95% / 93%

Table 5.11: Evaluation accuracy and F1-scores per class for the detection of opaque predicates constructions.

Yet, the overall accuracy of our model is at 95% and 93% depending on the evaluation approach used. This illustrates the efficiency of our methodology towards the detection of obfuscation transformations constructions.

5.6 Application

As previously discussed in [166], metadata recovery attacks are usually manual tasks, therefore a potential bottleneck in the reverse engineering process. Our methodology, which could be plugged-in a disassembler framework, provides all applied transformation and construction and allows reverse-engineers to setup automated deobfuscation strategies. In particular, it may prevent the

use of analyses that are inefficient against specific transformations, such as mixed-boolean and arithmetic opaque predicates against SMT solvers, or dynamic-symbolic execution against range dividers.

Since code obfuscation may introduce some overhead to the execution time of a binary, the transformations are not applied throughout all the functions. Generally, heavy transformations are used to protect sensitive code portions. Thus, another application of an efficient metadata recovery framework consists in the detection of sensitive functions, which are detected as obfuscated with several transforms.

Finally, malware authors are usually known to implement specific protections schemes and patterns to hide their malicious intent. With a proper training dataset, we believe that our methodology can scale to the detection of such patterns, thus providing another application for the classification of malicious samples.

5.7 Limitations

One threat to the validity of our results is that we only use datasets of relatively small C programs, except for the core-utils binaries used for non-obfuscated samples. However, we consider that our dataset can be representative for a large number of programs. They use all common programming language constructions and different functionalities (*e.g.* hash functions, sorts, cryptographic algorithms, etc.). Nevertheless, our work shows that semantic reasoning combined with advanced machine learning present capabilities for a fine-grained detection of obfuscation transforms.

The capability of detecting *unknown* transformations or constructions represents another limitation of our methodology. If our model did not trained on one specific transformation or constructions, it will not predict properly the unknown sample. This can lead to a loss of accuracy when unknown transformations are combined with others.

Dynamic transformations cause limitations to our model for the *static* detection of obfuscation transforms. Despite from the fact that we are able to accurately detect some of these transformations (*i.e.* Jit, Code Virtualization), when other obfuscation transformations are applied before them, our model is less efficient. Moreover, other transformations such as packing, or anti-symbolic execution techniques may lower the accuracy of our model. However, as we introduce in the next section, our methodology can scale to dynamically collected traces which allows to thwart some of these limitations.

5.8 Perspectives and future work

First, more in-depth studies of aggregation approaches used in ensemble learning must be done in order to see if ensemble learning are more efficient for that task compared to mono-models. The hard voting scheme used is a simple approach, but may not achieve the real gain behind the use of the ensemble learning approach.

As already shown in other work related to deobfuscation [184], semantic reasoning and machine learning provides promising results. We believe that, compared to the results in Salem *et al.* work, our model does not depend on the code functionality as illustrated in our evaluations. However, a more accurate comparison must be made as future work.

In order to overcome the dynamic transformations limitations, our methodology is easily adaptable to dynamically collected instructions traces. With a given instructions trace, it is possible to reconstruct each basic-blocks and apply our semantic reasoning approach in order to generate raw data. This step can be done either for the learning or the evaluation phase. Our future work consists in extending the implementation of our framework for this and evaluating other combinations of obfuscation transformations based on dynamic traces.

Another future work we need to consider is the application of n layer of the same obfuscation transformations. For now, our evaluations is done by combining several transformations, but using one time each of them. We believe that extending our evaluations to the use of one transformation several times is an interesting study.

5.9 Conclusions

In this chapter we presented the efficiency of semantic reasoning combined with advanced machine learning techniques. This combination is motivated by the construction of a fine-grained detection framework of obfuscation transformations and constructions. By extending our approach to multi-label and multi-output classification, we enhanced metadata recovery attacks to the detection of multiple layers of obfuscation transformations. We proposed a new approach that combines a bloc-centric symbolic execution with machine learning ensemble model and classifier chains. We used our models to evaluate the stealth of both obfuscation transformations and constructions. Our results are promising, with overall accuracies up to 91% for the transformations and 100% for the constructions. The use of static symbolic execution allows us to be dependent on the underlying functionality of the code samples used for the learning phase. Our empirical studies illustrate that our choices conduct towards the implementation of an efficient and accurate evaluation framework against state of the art obfuscators. However, there is still place for improvements with a more in-depth study of learning algorithms used and their parameters. Yet, our work improves

metadata-recovery attacks, and paves the way towards the efficient use of advanced machine learning combined with semantic

Chapter 6

Conclusion

Contents

In this thesis we have studied different deobfuscation approaches toward a static evaluation of obfuscation transformations. We mainly focused on static semantic reasoning, combining it with well known techniques from other research areas such as binary diffing and machine learning. We studied and developed several deobfuscation frameworks, one for each of the followings approaches; simplifying the obfuscated code, removing the obfuscation transformations or gathering informations about the protections applied. Our methodologies and designs have been validated on well known malwares and state-of-the-art obfuscators implementing widely used obfuscation transformations. In this chapter, we start by giving a summary of all this thesis contributions. We conclude with our perspectives for future work.

6.1 Contributions summary

This thesis contributions are made to answer the following questions:

1. *How can we contribute to existing generic deobfuscation methodologies?*
2. *How can we use machine learning techniques for the purpose of removing widely used obfuscation transformations?*
3. *How can we help reverse-engineers select the adequate deobfuscation analyses?*

6.1.1 How can we contribute to existing generic deobfuscation methodologies?

In order to answer the first question, we transposed semantic-based binary diffing techniques for the purpose of statically simplifying obfuscated binaries. We developed our methodology, called `DoSE`, as an `IDA` Pro plug-in and for three major applications, namely:

- A bloc-centric and intra-procedural approach to statically simplify control-flow graph by detecting cloned basic-blocks;
- An path-oriented, bounded and intra-procedural approach to statically detect two-way opaque predicates. To the best of our knowledge, no other work tackles these types of opaque predicates.
- A function-oriented approach to detect cloned branching functions in order to reduce the amount of code to be analyzed.

We evaluated each applications against real-world and well-known malwares such as `Cryptowall` and `Flame`. Our evaluations underlines the efficiency of DoSE for each applications, with up to 63% of control-flow graph reduction or 1954 cloned functions detected on `Flame`. 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 contribution paves the way for combining semantic equivalence methodologies with existing generic deobfuscation techniques, in order to improve their efficiency and scalability.

6.1.2 How can we use machine learning techniques for the purpose of removing widely used obfuscation transformations?

Our second contribution is, to the best of our knowledge, the first deobfuscation technique based on machine learning for the purpose of removing obfuscation transformations. By introducing the

different constructions of opaque predicates and the limitations from dynamic symbolic execution techniques and SMT solvers, we underlined the importance of studying other alternatives for generic evaluations of these transformations.

We developed an `IDA` Pro plug-in, a new approach that bridges a thresholded static symbolic execution with machine learning classification to evaluate both the stealth and resilience of invariant opaque predicates constructions. The use of static symbolic execution allows us to have a better code coverage and scalability, which combined with a machine learning model, permits a generic approach by discarding the use of SMT solvers.

Our studies illustrate that our choices conduct towards the implementation of an efficient and accurate evaluation framework against state of the art obfuscators. We created two models for the evaluation of stealth and resiliency of state-of-the-art opaque predicates constructions, with results up to 99% for detection and 95% for deobfuscation. Moreover, we extended our work to a deobfuscation plug-in and compared our results to other tools, showing the efficiency of machine learning for the deobfuscation of most invariant opaque predicates constructions.

Thus, this contribution initiate the use of machine learning techniques in order to remove obfuscation transformations while preserving a genericity with respect to their constructions.

6.1.3 How can we help reverse-engineers select the adequate deobfuscation analyses?

The third contribution of this thesis aims at improving existing metadata recovery attack. Existing technique syntax-oriented and thus suffers from several limitations. We presented the efficiency of semantic reasoning combined with advanced machine learning techniques. This combination is motivated by the construction of a fine-grained detection framework of obfuscation transformations and constructions. By extending our approach to multi-label and multi-output classification, we enhanced metadata recovery attacks to the detection of multiple layers of obfuscation transformations, as well as their constructions.

Our results are promising, with overall accuracies up to 91% for the transformations and 100% for the constructions. This further illustrates that our choices conduct towards the implementation of an efficient and accurate evaluation framework against state of the art obfuscators.

6.2 Perspectives

Our different contributions give us several perspectives for future work. Since we developed two evaluation methodologies based on machine learning techniques, a first perspective is to provide a more in-depth study on deep learning techniques for deobfuscation purposes.

Second, all of our contributions are based on static analyses. The ability to combine them with existing dynamic techniques to provide more efficient generic deobfuscation approach is another idea for future work.

Third, from existing works and our contributions, a possible future work is to build a complete evaluation framework that gathers all its different approaches, namely, the collection of metadata informations, the simplification, and the removal of obfuscation transforms. With such framework, for example as an `IDA` plug-in, we can provide analysts and reverse-engineer a complete set of tools and techniques to efficiently and rapidly work on protected binaries.

Overall, we believe that our approach of combining semantic-reasoning of a program with machine learning techniques can be applied to further code obfuscation and deobfuscation research subjects, which is something that we envision to explore.

Appendix A

Defeating Opaque Predicate using Binary Analysis and Machine Learning

Contents

In the following we present appendixes for the second contribution of this thesis, namely the evaluation of our methodology for breaking opaque predicates using binary analysis and machine learning.

A.1 Tigress commands

In the followings, we list the combinations of obfuscation transformations used for our datasets, in their application order. Note that the combinations listed in *italic* are considered as clean samples since they do not generate opaque predicates.

- AddOpaque (16 or 32 times)
- AddOpaque, EncodeLiterals
- *EncodeLiterals*
- AddOpaque, EncodeArithmetics
- EncodeArithmetics, AddOpaque
- *EncodeArithmetics*
- AddOpaque, EncodeData
- EncodeData, AddOpaque
- *EncodeData*
- AddOpaque, EncodeArithmetics, EncodeLiterals, EncodeData
- EncodeData, EncodeArithemtics, EncodeLiterals, AddOpaque
- AddOpaque, Flatten
- Flatten, AddOpaque
- *Flatten*
- *Flatten, EncodeData, EncodeArithemtics, EncodeLiterals*
- Virtualize, AddOpaque
- *Virtualize*
- *Virtualize, EncodeData, EncodeArithemtics, EncodeLiterals*
- *Virtualize, Flatten*
- Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals

A.1.1 Commands options

```
# AddOpaque options
tigress --Transform=InitEntropy --Transform=InitOpaque --InitOpaqueStructs=
    list,array,env --Functions=main --Transform=AddOpaque --Functions=${3}
    --AddOpaqueCount=${NUM} --AddOpaqueKinds=call,fake,true

# Flatten
tigress --Transform=Flatten --FlattenDispatch=switch,goto --Functions=${3}

# Virtualize
tigress --Transform=Virtualize --VirtualizeDispatch=switch,direct,ifnest,
    linear --Functions=${3}

# EncodeLiterals
tigress --Transform=EncodeLiterals --Functions=${3} --EncodeLiteralsKinds=
    integer

# EncodeArithmetics
tigress --Transform=EncodeArithmetic --Functions=${3} --EncodeLiteralsKinds
    =integer

# EncodeData
tigress --Transform=EncodeData --LocalVariables=${4} --EncodeDataCodecs=
    poly,xor,add --Functions=${3}
```

Listing A.1: Tigress commands for sample generation

Appendix B

Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and semantic reasoning

Contents

In the following we present appendixes for the third contribution of this thesis, namely the fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning.

B.1 Tigress transformations

In the followings, we list the combinations of obfuscation transformations used for our datasets, in their application order.

- AddOpaque (16 or 32 times)
- AddOpaque, EncodeLiterals
- EncodeLiterals
- AddOpaque, EncodeArithmetics
- EncodeArithmetics, AddOpaque
- EncodeArithmetics
- AddOpaque, EncodeData
- EncodeData, AddOpaque
- EncodeData
- AddOpaque, EncodeArithmetics, EncodeLiterals, EncodeData
- EncodeData, EncodeArithemtics, EncodeLiterals, AddOpaque
- AddOpaque, Flatten
- Flatten, AddOpaque
- Flatten
- Flatten, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, AddOpaque
- Virtualize
- Virtualize, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, Flatten
- Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals
- Virtualize, Flatten, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals

- Jit
- Jit, AddOpaque
- Jit, AddOpaque, EncodeData, EncodeArithemtics, EncodeLiterals

B.1.1 Commands options

```
# AddOpaque options
tigress --Transform=InitEntropy --Transform=InitOpaque --InitOpaqueStructs
   =list,array,env --Functions=main --Transform=AddOpaque --Functions=${3}
    --AddOpaqueCount=${NUM} --AddOpaqueKinds=call,fake,true

# Flatten
tigress --Transform=Flatten --FlattenDispatch=switch,goto --Functions=${3}

# Virtualize
tigress --Transform=Virtualize --VirtualizeDispatch=switch,direct,ifnest,
   linear --Functions=${3}

# Jit
tigress -include $TIGRESS_HOME/jitter-amd64.c --Transform=Jit --Functions=
   ${3} --JitEncoding=hard

# EncodeLiterals
tigress --Transform=EncodeLiterals --Functions=${3} --EncodeLiteralsKinds=
   integer,string

# EncodeArithmetics
tigress --Transform=EncodeArithmetic --Functions=${3} --
   EncodeLiteralsKinds=integer

# EncodeData
tigress --Transform=EncodeData --LocalVariables=${4} --EncodeDataCodecs=
   poly,xor,add --Functions=${3}

```

Listing B.1: Tigress commands for sample generation

B.2 OLLVM transformations

In the followings, we list the combinations of obfuscation transformations used for our datasets, in their application order.

- bcf
- bcf, sub
- bcf, sub, fla
- bcf, fla, sub
- sub
- sub, bcf
- sub, bcf, fla
- fla
- fla, bcf
- fla, sub, bcf
- fla, bcf, sub

B.2.1 Commands options

```
# Bogus control-flow
clang ${1}.c -o ${1} -mllvm -bcf -mllvm -bcf_prob=50
clang ${1}.c -o ${1} -mllvm -bcf -mllvm -bcf_prob=100

# Control-flow flattening
clang ${1}.c -o ${1} -mllvm -fla
clang ${1}.c -o ${1} -mllvm -fla -mllvm -split

# Instruction substitution
clang ${1}.c -o ${1} -mllvm -sub


```

Listing B.2: O-LLVM commands for sample generation

Bibliography

[1] *Triton: A Dynamic Symbolic Execution Framework*. SSTIC, 2015.

[2] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *J. Network and Computer Applications*, 48:44–57, 2015.

[3] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. 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*, pages 79–84, 2016.

[4] The Algorithms. C. `https://github.com/TheAlgorithms/C/`. [Online; accessed 30-01-2019].

[5] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[6] Bertrand Anckaert, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In Moti Yung, Kaoru Kurosawa, and Reihaneh Safavi-Naini, editors, *Proceedings of the Sixth ACM Workshop on Digital Rights Management, Alexandria, VA, USA, October 30, 2006*, pages 47–58. ACM, 2006.

[7] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In Günter Karjoth and Ketil Stølen, editors, *Proceedings of the 3th ACM Workshop on Quality of Protection, QoP 2007, Alexandria, VA, USA, October 29, 2007*, pages 15–20. ACM, 2007.

[8] Daniel Apon, Yan Huang, Jonathan Katz, and Alex J. Malozemoff. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive*, 2014:779, 2014.

[9] Geneviève Arboit. A method for watermarking java programs via opaque predicates. In *In Proc. Int. Conf. Electronic Commerce Research (ICECR-5)*, 2002.

[10] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[11] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[12] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018.

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

[14] Sebastian Banescu, Christian S. Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 661–678. USENIX Association, 2017.

[15] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[16] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, 2012.

[17] Sébastien Bardin, Robin David, and Jean-Yves Marion. 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*, pages 633–651, 2017.

[18] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The BINCOA framework for binary code analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 165–170. Springer, 2011.

[19] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[20] Fabrizio Biondi, Michael A. Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. Effective, efficient, and robust packing detection and classification. *Computers & Security*, 85:436–451, 2019.

[21] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[22] Nir Bitansky, Ran Canetti, Shafi Goldwasser, Shai Halevi, Yael Tauman Kalai, and Guy N. Rothblum. Program obfuscation with leaky hardware. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 722–739. Springer, 2011.

[23] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 643–659, 2017.

[24] Boldizsar Bencsath. Duqu, Flame, Gauss: Followers of Stuxnet. https://www.rsaconference.com/writable/presentations/file_upload/br-208_bencsath.pdf. [Online; accessed 30-01-2019].

[25] Guillaume Bonfante, José M. Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In Ray et al. [153], pages 745–756.

[26] Matthew R. Boutell, Jiebo Luo, Xipeng Shen, and Christopher M. Brown. Learning multi-label scene classification. *Pattern Recognition*, 37(9):1757–1771, 2004.

[27] Murray Brand. Forensic analysis avoidance techniques of malware. *ECU Publications*, 01 2007.

[28] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[29] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[30] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[31] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.

[32] David Brumley, Pongsin Poosankam, Dawn Xiaodong Song, and Jiang Zheng. 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*, pages 143–157, 2008.

[33] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006, Proceedings*, pages 129–143, 2006.

[34] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[35] Jan Cappaert, Nessim Kisserli, Dries Schellekens, and Bart Preneel. Self-encrypting code to protect against analysis and tampering. 01 2006.

[36] Jan Cappaert and Bart Preneel. A general model for hiding control flow. In Ehab Al-Shaer, Hongxia Jin, and Marc Joye, editors, *Proceedings of the 10th ACM Workshop on Digital Rights Management, Chicago, Illinois, USA, October 4, 2010*, pages 35–42. ACM, 2010.

[37] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014.

[38] Zeynep Ceylan and Ebru Pekel. Comparison of multi-label classification methods for prediagnosis of cervical cancer. *International Journal of Intelligent Systems and Applications in Engineering*, 5(4):232–236, Dec. 2017.

[39] Serge Chaumette, Olivier Ly, and Renaud Tabary. Automated extraction of polymorphic virus signatures using abstract interpretation. In Samarati et al. [168], pages 41–48.

[40] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[41] Stanley Chow, Yuan Xiang Gu, Harold Johnson, and Vladimir A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In George I. Davida and Yair Frankel, editors, *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2001.

[42] Frederick B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[43] Christian Collberg, Sam Martin, Jonathan Myers, Bill Zimmerman, Petr Krajca, Gabriel Kerneis, Saumya Debray, and Babak Yadegari. The Tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/index.html`. [Online; accessed 30-01-2019].

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

[45] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[46] Christian S. Collberg, Jack W. Davidson, Roberto Giacobazzi, Yuan Xiang Gu, Amir Herzberg, and Fei-Yue Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.

[47] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 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*, pages 184–196, 1998.

[48] An ECRYPT White-Box Cryptography Competition. Ches 2017 capture the flag challenge - the whibox contest, 2017.

[49] Brad Conte. crypto-algorithms. `https://github.com/B-Con/crypto-algorithms`. [Online; accessed 30-01-2019].

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

[51] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. Feedback-driven binary code diversification. *TACO*, 9(4):24:1–24:26, 2013.

[52] Marie-Angela Cornelie. *Implantations et protections de mécanismes cryptographiques logiciels et matériels. (Implementations and protections of software and hardware cryptographic mechanisms)*. PhD thesis, Grenoble Alpes University, France, 2016.

[53] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, September 2006.

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

[55] Robin David. *Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes. (Approches formelles de désobfuscation automatique et de rétro-ingénierie de codes protégés)*. PhD thesis, University of Lorraine, Nancy, France, 2017.

[56] Robert L. Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program.

[57] Fabrice Desclaux. Miasm : Framework de reverse engineering. `https://github.com/cea-sec/miasm`, 2012. [Online; accessed 30-01-2019].

[58] Thomas G. Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[59] Thomas G. Dietterich. Ensemble methods in machine learning. In *Proceedings of the First International Workshop on Multiple Classifier Systems*, MCS '00, pages 1–15, London, UK, UK, 2000. Springer-Verlag.

[60] Stephen Drape. *Obfuscation of abstract data-types*. PhD thesis, University of Oxford, UK, 2004.

[61] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[62] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, 2012.

[63] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*, pages 27–38, 2016.

[64] Paolo Falcarin, Christian S. Collberg, Mikhail J. Atallah, and Mariusz H. Jakubowski. Guest editors' introduction: Software protection. *IEEE Software*, 28(2):24–27, 2011.

[65] Mohammad Reza Farhadi, Benjamin C. M. Fung, Philippe Charland, and Mourad Debbabi. 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*, pages 78–87, 2014.

[66] Rosa L. Figueroa, Qing Zeng-Treitler, Sasikiran Kandula, and Long H. Ngo. Predicting sample size required for classification performance. *BMC Med. Inf. & Decision Making*, 12:8, 2012.

[67] Julie D. Bennett Frank Nian-Tzu Chu, Wei Wu and Mohammed El-Gammal. Software obfuscation.

[68] Julie D. Bennett Frank Nian-Tzu Chu, Wei Wu and Mohammed El-Gammal. Thread protection.

[69] Yoav Freund. Boosting a weak learning algorithm by majority. In Mark A. Fulk and John Case, editors, *Proceedings of the Third Annual Workshop on Computational Learning Theory, COLT 1990, University of Rochester, Rochester, NY, USA, August 6-8, 1990.*, pages 202–216. Morgan Kaufmann, 1990.

[70] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In Lorenza Saitta, editor, *Machine Learning, Proceedings of the Thirteenth International Conference (ICML '96), Bari, Italy, July 3-6, 1996*, pages 148–156. Morgan Kaufmann, 1996.

[71] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, 1997.

[72] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Mach. Learn.*, 29(2-3):131–163, November 1997.

[73] Bin Fu, Sai Aravalli, and John Abraham. Software protection by hardware and obfuscation. In Selim Aissi and Hamid R. Arabnia, editors, *Proceedings of the 2007 International Conference on Security & Management, SAM 2007, Las Vegas, Nevada, USA, June 25-28, 2007*, pages 367–373. CSREA Press, 2007.

[74] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. 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*, pages 238–255, 2008.

[75] Peter Garba and Matteo Favaro. SATURN - software deobfuscation framework based on LLVM. In Paolo Falcarin and Michael Zunke, editors, *Proceedings of the 3rd ACM Workshop on Software Protection, SPRO@CCS 2019, London, Uk, November 15, 2019*, pages 27–38. ACM, 2019.

[76] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 40–49. IEEE Computer Society, 2013.

[77] Sudeep Ghosh, Jason Hiser, and Jack W. Davidson. A secure and robust approach to software tamper resistance. In Rainer Böhme, Philip W. L. Fong, and Reihaneh Safavi-Naini, editors, *Information Hiding - 12th International Conference, IH 2010, Calgary, AB, Canada, June 28-30, 2010, Revised Selected Papers*, volume 6387 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2010.

[78] Shantanu Godbole and Sunita Sarawagi. Discriminative methods for multi-labeled classification. In Honghua Dai, Ramakrishnan Srikant, and Chengqi Zhang, editors, *Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004, Proceedings*, volume 3056 of *Lecture Notes in Computer Science*, pages 22–30. Springer, 2004.

[79] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in Computer Virology*, 6(3):261–276, 2010.

[80] Satoshi Hada. Zero-knowledge and code obfuscation. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2000.

[81] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[82] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Not.*, 16(3):63–74, March 1981.

[83] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.

[84] Marti A. Hearst. Support vector machines. *IEEE Intelligent Systems*, 13(4):18–28, July 1998.

[85] Sean Heelan and Agustin Gianni. Augmenting vulnerability analysis of binary code. In Robert H'obbes' Zakon, editor, *28th Annual Computer Security Applications Conference, ACSAC 2012, Orlando, FL, USA, 3-7 December 2012*, pages 199–208. ACM, 2012.

[86] Hex-Rays. IDA Pro : Interactive DisAssembler. https://www.hex-rays.com/products/ida/index.shtml. [Online; accessed 30-01-2019].

[87] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[88] Simon Howard. c-algorithms. https://github.com/fragglet/c-algorithms. [Online; accessed 30-01-2019].

[89] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 611–620, 2009.

[90] Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Towards integral binary execution: implementing oblivious hashing using overlapped instruction encodings. In Deepa Kundur, Balakrishnan Prabhakaran, Jana Dittmann, and Jessica J. Fridrich, editors, *Proceedings of the 9th workshop on Multimedia & Security, MM&Sec 2007, Dallas, Texas, USA, September 20-21, 2007*, pages 129–140. ACM, 2007.

[91] Mike James. *Classification Algorithms*. Wiley-Interscience, New York, NY, USA, 1985.

[92] Albert B. Jeng and Chia Ling Lee. A study on online game cheating and the effective defense. In Moonis Ali, Tibor Bosse, Koen V. Hindriks, Mark Hoogendoorn, Catholijn M. Jonker, and Jan Treur, editors, *Recent Trends in Applied Artificial Intelligence, 26th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2013, Amsterdam, The Netherlands, June 17-21, 2013. Proceedings*, volume 7906 of *Lecture Notes in Computer Science*, pages 518–527. Springer, 2013.

[93] Liangxiao Jiang, Dianhong Wang, Zhihua Cai, and Xuesong Yan. Survey of improving naive bayes for classification. In *Proceedings of the 3rd International Conference on Advanced Data Mining and Applications*, ADMA '07, pages 134–145, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[95] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 60(5):493–502, 2004.

[96] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st Interna-*

tional Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, pages 3–9. IEEE, 2015.

[97] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. Vmattack: Deobfuscating virtualization-based packed binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 2:1–2:10. ACM, 2017.

[98] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*, page 170. IEEE Computer Society, 2003.

[99] Md. Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[100] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A qualitative analysis of java obfuscation. 01 2006.

[101] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[102] Raymond R. Kiddy. Method of obfuscating computer instruction streams.

[103] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, pages 61–70. IEEE Computer Society, 2012.

[104] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2008.

[105] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[106] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, and Jacob R. Lorch. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 314–327. IEEE Computer Society, 2006.

[107] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1137–1145, 1995.

[108] Sotiris B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[109] Aleksandrina Kovacheva. Efficient code obfuscation for android. In *Advances in Information Technology - 6th International Conference, IAIT 2013, Bangkok, Thailand, December 12-13, 2013. Proceedings*, pages 104–119, 2013.

[110] Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. 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*, pages 207–226, 2005.

[111] Arun Lakhotia, Davidson R. Boccardo, Anshuman Singh, and Aleardo Manacero Jr. Context-sensitive analysis without calling-context. *Higher-Order and Symbolic Computation*, 23(3):275–313, 2010.

[112] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Software Eng.*, 31(11):955–968, 2005.

[113] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 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*, pages 5:1–5:6, 2013.

[114] Chris Lattner and Vikram S. Adve. 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*, pages 75–88, 2004.

[115] Timea Lazlo and Akos Kiss. Obfuscating c++ programs via control flow flattening.

[116] Tao Li and Mitsunori Ogihara. Detecting emotion in music. In *ISMIR 2003, 4th International Conference on Music Information Retrieval, Baltimore, Maryland, USA, October 27-30, 2003, Proceedings*, 2003.

[117] Cullen Linn and Saumya K. Debray. Obfuscation of executable code to improve resistance to static disassembly. In Sushil Jajodia, Vijayalakshmi Atluri, and Trent Jaeger, editors, *Pro-*

ceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, Washington, DC, USA, October 27-30, 2003, pages 290–299. ACM, 2003.

[118] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 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*, pages 389–400, 2014.

[119] Richard Maclin and David W. Opitz. Popular ensemble methods: An empirical study. *CoRR*, abs/1106.0257, 2011.

[120] A Madi, O.K. Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and its Applications*, 7:67–82, 01 2013.

[121] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya K. Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In JooSeok Song, Taekyoung Kwon, and Moti Yung, editors, *Information Security Applications, 6th International Workshop, WISA 2005, Jeju Island, Korea, August 22-24, 2005, Revised Selected Papers*, volume 3786 of *Lecture Notes in Computer Science*, pages 194–206. Springer, 2005.

[122] Miquel Martínez, David de Andrés, Juan Carlos Ruiz, and Jesus Friginal. Analysis of results in dependability benchmarking: Can we do better? In *2nd IEEE International Workshop on Measurements & Networking, M&N 2013, Naples, Italy, October 7-8, 2013*, pages 127–131. IEEE, 2013.

[123] Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. Stuxnet Under the Microscope. `https://www.esetnod32.ru/company/viruslab/analytics/doc/Stuxnet_Under_the_Microscope.pdf`. [Online; accessed 01-04-2019].

[124] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[125] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, July 1976.

[126] R.Venkatesan M.H.Jakubowski and S.Sinha. System and method for protecting digital goods using random and automatic code obfuscation.

[127] Jiang Ming, Meng Pan, and Debin Gao. 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*, pages 92–109, 2012.

[128] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 757–768, 2015.

[129] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 757–768, 2015.

[130] Jiang Ming, Dongpeng Xu, and Dinghao Wu. 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*, pages 416–430, 2015.

[131] Akito Monden, Antoine Monsifrot, and Clark D. Thomborson. A framework for obfuscated interpretation. In James M. Hogan, Paul Montague, Martin K. Purvis, and Chris Steketee, editors, *ACSW Frontiers 2004, 2004 ACSW Workshops - the Australasian Information Security Workshop (AISW2004), the Australasian Workshop on Data Mining and Web Intelligence (DMWI2004), and the Australasian Workshop on Software Internationalisation (AWSI2004) . Dunedin, New Zealand, January 2004*, volume 32 of *CRPIT*, pages 7–16. Australian Computer Society, 2004.

[132] Ginger Myles and Christian S. Collberg. 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*, pages 314–318, 2005.

[133] Ginger Myles and Christian S. Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[134] Alexios Mylonas and Dimitris Gritzalis. Practical malware analysis: The hands-on guide to dissecting malicious software. *Computers & Security*, 31(6):802–803, 2012.

[135] Carey Nachenberg. Computer virus-antivirus coevolution. *Commun. ACM*, 40(1):46–51, 1997.

[136] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[137] Beng Heng Ng, Xin Hu, and Atul Prakash. A study on latent vulnerabilities. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 333–337, 2010.

[138] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[139] Jeongwook Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. Black Hat USA 2009, 2009.

[140] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free; or unleashing the potential of path-oriented protections, 2019.

[141] Oreans Technologies. Code virtualizer: Total obfuscation against reverse engineering. `http://www.oreans.com/codevirtualizer.php`. [Online; accessed 30-01-2019].

[142] Oreans Technologies. Themida – Advanced Windows Software Protection System. `http://www.oreans.com/themida.php`. [Online; accessed 30-01-2019].

[143] Linda M. Ott and Jeffrey J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the First International Software Metrics Symposium, METRICS 1993, May 21-22, 1993, Balimore, Maryland, USA*, pages 71–81. IEEE Computer Society, 1993.

[144] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA*, pages 308–316, 2000.

[145] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[146] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. *it - Information Technology*, 59(2):83, 2017.

[147] Fabrice Desclaux Philippe Biondi. Silver Needle in the Skype. `https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf`. [Online; accessed 30-01-2019].

[148] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In Niels Provos, editor, *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*. USENIX Association, 2007.

[149] Mila Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, Italy, 2007.

[150] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque predicates detection by abstract interpretation. In *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, pages 81–95, 2006.

[151] GNU Project. GNU Core Utilities. `https://www.gnu.org/software/coreutils/`, 2002. [Online; accessed 30-01-2019].

[152] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[153] Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM, 2015.

[154] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333–359, 2011.

[155] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Mach. Learn.*, 85(3):333–359, December 2011.

[156] Stéphanie Riaud. *Obfuscation de données pour la protection de programme contre l'analyse dynamique*. PhD thesis, Université de Rennes 1, 2015.

[157] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[158] Thomas Rinsma. Seeing through obfuscation: interactive detection and removal of opaque predicates. `https://github.com/Riscure/DROP-IDA-plugin`, 2017. [Online; accessed 30-01-2019].

[159] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 15–24. Springer, 1998.

[160] Lior Rokach. Ensemble-based classifiers. *Artif. Intell. Rev.*, 33(1-2):1–39, 2010.

[161] Lior Rokach and Oded Maimon. *Data Mining With Decision Trees: Theory and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2nd edition, 2014.

[162] Rolf Rolles. Unpacking virtualization obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[163] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[164] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, 2013.

[165] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel J. Quinlan, and Zhendong Su. 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*, pages 117–128, 2009.

[166] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW@ACSAC 2016, Los Angeles, California, USA, December 5-6, 2016*, pages 1:1–1:11, 2016.

[167] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 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*, pages 372–392, 2018.

[168] Pierangela Samarati, Sara Foresti, Jiankun Hu, and Giovanni Livraga, editors. *5th International Conference on Network and System Security, NSS 2011, Milan, Italy, September 6-8, 2011*. IEEE, 2011.

[169] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In Tomás Filler, Tomás Pevný, Scott Craver, and Andrew D. Ker, editors, *Information Hiding - 13th International Conference, IH 2011, Prague, Czech Republic, May 18-20, 2011, Revised Selected Papers*, volume 6958 of *Lecture Notes in Computer Science*, pages 270–284. Springer, 2011.

[170] Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar R. Weippl. Covert computation - hiding code in code through compile-time obfuscation. *Computers & Security*, 42:13–26, 2014.

[171] Sandro Schulze and Daniel Meyer. 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*, pages 62–68, 2013.

[172] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 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*, pages 317–331, 2010.

[173] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[174] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society, 2008.

[175] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[176] Luís M. Silva, J. Marques de Sá, and Luís A. Alexandre. Data classification with multilayer perceptrons using a generalized error function. *Neural Netw.*, 21(9):1302–1310, November 2008.

[177] Padhraic Smyth and David H. Wolpert. Linearly combining density estimators via stacking. *Machine Learning*, 36(1-2):59–83, 1999.

[178] Harold J. Johnson Stanley T. Chow and Yuan Gu. Tamper resistant software encoding.

[179] Harold J. Johnson Stanley T. Chow and Yuan Gu. Tamper resistant software encoding.

[180] StrongBit Technology. EXECryptor – bulletproof software protection. `www.strongbit.com/execryptor.asp`. [Offline; accessed 30-01-2019].

[181] Xiaoyuan Su, Taghi M. Khoshgoftaar, and Russell Greiner. Making an accurate classifier ensemble by voting on classifications from imputed learning sets. *IJIDS*, 1(3):301–322, 2009.

[182] Li Sun, Steven Versteeg, Serdar Boztas, and Trevor Yann. Pattern recognition techniques for the classification of malware packers. In Ron Steinfeld and Philip Hawkes, editors, *Information Security and Privacy - 15th Australasian Conference, ACISP 2010, Sydney, Australia, July 5-7,*

2010. Proceedings, volume 6168 of *Lecture Notes in Computer Science*, pages 370–390. Springer, 2010.

[183] Bjorn De Sutter, Bertrand Anckaert, Jens Geiregat, Dominique Chanet, and Koen De Bosschere. Instruction set limitation in support of software diversity. In Pil Joong Lee and Jung Hee Cheon, editors, *Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers*, volume 5461 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2008.

[184] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. In *3rd International Workshop on Software PROtection*, London, United Kingdom, November 2019.

[185] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh Ha Le. DoSE: Deobfuscation based on Semantic Equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, San Juan, PR, USA, December 3-4, 2018*, pages 1:1–1:12, 2018.

[186] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *IJDWM*, 3(3):1–13, 2007.

[187] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering, WCRE 2005, Pittsburgh, PA, USA, November 7-11, 2005*, pages 45–54, 2005.

[188] Xabier Ugarte-Pedrero, Igor Santos, Pablo García Bringas, Mikel Gastesi, and José Miguel Esparza. Semi-supervised learning for packed executable detection. In Samarati et al. [168], pages 342–346.

[189] VMProtect Software. VMProtect – New-generation software protection. `http://vmpsoft.com/products/vmprotect/`. [Online; accessed 30-01-2019].

[190] Zeljko Vrba, Pål Halvorsen, and Carsten Griwodz. Program obfuscation by strong cryptography. In *ARES 2010, Fifth International Conference on Availability, Reliability and Security, 15-18 February 2010, Krakow, Poland*, pages 242–247. IEEE Computer Society, 2010.

[191] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. 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*, pages 193–202, 2001.

[192] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 8–9, 2017.

[193] Xinran Wang, Yoon-chan Jhi, Sencun Zhu, and Peng Liu. Behavior based software theft detection. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 280–290, 2009.

[194] Zheng Wang, Ken Pierce, and Scott McFarling. BMAT - A binary matching tool for stale profile propagation. *J. Instruction-Level Parallelism*, 2, 2000.

[195] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[196] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[197] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Trans. Softw. Eng.*, 5(1):45–50, January 1979.

[198] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. Mimimorphism: a new approach to binary code obfuscation. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 536–546. ACM, 2010.

[199] Dongpeng Xu, Jiang Ming, and Dinghao Wu. Generalized dynamic opaque predicates: A new control flow obfuscation method. In *Information Security - 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016, Proceedings*, pages 323–342, 2016.

[200] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael R. Lyu. Manufacturing resilient bi-opaque predicates against symbolic execution. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 666–677, 2018.

[201] Babak Yadegari. *Automatic Deobfuscation and Reverse Engineering of Obfuscated Code*. PhD thesis, University of Arizona, Tucson, USA, 2016.

[202] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In Ray et al. [153], pages 732–744.

[203] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 674–691. IEEE Computer Society, 2015.

[204] Mark Vincent Yason. The art of unpacking. Black Hat USA 2007, 2007.

[205] R.Venkatesan Y.Chen and M.H.Jakubowski. Secure and opaque type library providing secure data protection of variables.

[206] Yonathan Klijnsma. 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-sca pdf`. [Online; accessed 30-08-2019].

[207] G. P. Zhang. Neural networks for classification: A survey. *Trans. Sys. Man Cyber Part C*, 30(4):451–462, November 2000.

[208] Min-Ling Zhang, Yu-Kun Li, Xu-Ying Liu, and Xin Geng. Binary relevance for multi-label learning: An overview. *Front. Comput. Sci.*, 12(2):191–202, April 2018.

[209] Min-Ling Zhang and Zhi-Hua Zhou. Multilabel neural networks with applications to functional genomics and text categorization. *IEEE Trans. on Knowl. and Data Eng.*, 18(10):1338–1351, October 2006.

[210] Min-Ling Zhang and Zhi-Hua Zhou. Ml-knn: A lazy learning approach to multi-label learning. *Pattern Recogn.*, 40(7):2038–2048, July 2007.

[211] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 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*, pages 61–75, 2007.

[212] Xiaotong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, and Santosh Pande. Hardware assisted control flow obfuscation for embedded processors. In Mary Jane Irwin, Wei Zhao, Luciano Lavagno, and Scott A. Mahlke, editors, *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2004, Washington DC, USA, September 22 - 25, 2004*, pages 292–302. ACM, 2004.

[213] Lukas Zobernig, Steven D. Galbraith, and Giovanni Russello. When are opaque predicates useful? Cryptology ePrint Archive, Report 2017/787, 2017. `https://eprint.iacr.org/2017/787`.

[214] Zynamics. BinDiff. https://www.zynamics.com/bindiff.html, 2013. [Online; accessed 30-01-2019].