Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes

Robin David

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Formal Approaches for Automatic Deobfuscation and Reverse-engineering of Protected Codes

THÈSE

présentée et soutenue publiquement le 6 Janvier 2017

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Robin DAVID

Composition du jury

Rapporteurs :	Roberto Giacobazzi	Professeur, Université de Vérone
	Arun Lakhotia	Professeur, Université de Louisiane
Examinateurs :	Jean Goubault-Larrecq	Professeur, ENS Cachan
	Sarah Zennou	Docteur, chargée de recherche, Airbus Group
	Steve Kremer	Directeur de recherche, LORIA Nancy
	Éric Freyssinet	Officier de gendarmerie, Membre associé LORIA
Directeur de thèse :	Jean-Yves Marion	Professeur, Université de Lorraine
Encadrant :	Sébastien Bardin	Chargé de recherche, CEA, LIST

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Laboratoire de Sûreté et de Sécurité Logicielle — LSL, CEA-LIST

Remerciements

Cette thèse n'aurait jamais vu le jour sans la contribution et l'aide de nombreuses personnes. En particulier je souhaite remercier *Sébastien Bardin* qui a fait preuve d'une patience infinie et d'une disponibilité à laquelle peu d'encadrants peuvent prétendre. Il a su m'inculquer une rigueur et une discipline scientifique qui ont porté leurs fruits tout au long de cette thèse. Je souhaite aussi remercier tout particulièrement *Jean-Yves Marion* qui en dépit de toutes ses attributions a réussi à m'accorder maintes de ses précieuses heures pour m'aider à progresser sur mon sujet et me donner son analyse objective et éclairée.

Je tiens à remercier profondément tous mes collègues du LSL avec qui passer ces trois années fut un immense plaisir. Je remercie toutes les personnes ayant pris le temps de répondre à mes questions; *Adel* pour avoir répondu à toutes mes requêtes de décodage d'instructions, *Patricia* pour sa relecture minutieuse et ses conseils avisés concernant l'administratif au CEA, *Florent* pour les discussions toujours très enrichissantes autout de la sécurité ainsi que *Guillaume* pour son passage fugace au CEA, avec qui partager le bureau fut autant agréable qu'inversement proportionnellement productif.

Par ailleurs, je remercie les cousins de Vérimag et *Josselin* avec qui collaborer fut enrichissant, *Fabrice* qui m'a encouragé à poursuivre en thèse et *Thanh Dinh* pour toutes ses discussions autour de l'obfuscation. Je souhaite remercier chaleureusement *Joan Calvet* qui a joué un rôle clé dans la validation expérimentale de ma thèse en me fournissant le meilleur cas d'utilisation auquel on puisse rêver.

Je souhaite remercier du plus profond du coeur mes parents et ma famille pour leur soutien inconditionnel en leur aide en toute circonstances. Enfin je remercie profondément tous mes amis qui m'ont aidé à sortir la tête de ma thèse, qui ont su être présents quand il le fallait et qui ont probablement pâti de mon implication acharnée sur mon sujet.

Pour mon grand-père qui m'avait toujours encouragé à persévérer aussi loin que possible dans mes études

Contents

List of Figures

Listings

List of Tables

Résumé long

Introduction

Contexte

Le terme générique *Logiciel malveillant*, ou “malware” en anglais, désigne un logiciel développé dans l'intention de nuire à un système ou à des personnes. Ce type de logiciels a énormément évolué en une décennie, passant du vol de données ou de coordonnées bancaires [IOA12] à des attaques programmées, commanditées et développées par des états ou organisations à des fins géopolitiques [Lan13; USC14]. Le coût induit par ce genre de menaces rend de plus en plus important et critique la mise en œuvre de protections. La première étape consiste à analyser et comprendre le fonctionnement de ce type de logiciels. Le code source n'étant pas disponible, des techniques d'analyses au niveau binaire doivent être développées. Malheureusement, ces programmes mettent en œuvre des techniques visant à ralentir la compréhension par l'analyste ou à gêner des algorithmes automatiques. Une de ces protections, *l'obfuscation*, vise à cacher le comportement ou la structure d'un programme. Ainsi, cette thèse a pour but d'étudier et de développer des algorithmes robustes de détection et de suppression d'obfuscation.

Défis Scientifiques

Le but de cette thèse est de fournir des algorithmes d'analyse efficaces de détection de code obscurci au niveau binaire. L'analyse binaire est reconnue plus difficile que l'analyse de code source pour plusieurs raisons : la distinction entre code et donnée n'est pas triviale. De plus, le graphe de flot de contrôle n'est pas forcément disponible [BHV11; BR10; MM16], l'identification des fonctions du programme n'est pas aisée et les données ne sont pas typées comme elles le seraient sur du code source. Notons enfin que la grande diversité d'architectures complique la création d'outils d'analyse. Effectuer ce genre d'analyses sur des programmes obscurcis est donc d'autant plus dur que ceux-ci mettent en œuvre des contre-mesures pour les contourner. Les deux principaux défis scientifiques sont :

- *Comment faire des algorithmes robustes qui passent à l'échelle sur du code obscurci ?*,

- *Comment localiser l'obfuscation et l'enlever pour rendre le code plus exploitable pour l'analyste ?*

Répondre à ces deux questions permettrait de grandes avancées dans le domaine de l'analyse de virus.

Contributions

Cette thèse s'inspire et prend racine dans les *méthodes formelles* initialement développées pour des analyses de sûreté de logiciels critiques. Nous mettons l'accent sur l'*Éxecution Dynamique Symbolique* (DSE) [CS13] théoriquement plus adaptée pour des questions d'obfuscation de par sa grande robustesse. Nos quatre principales contributions sont listées ci-dessous.

Contribution #1 Nous proposons deux variantes du DSE adaptées à l'obfuscation.

- la gestion des concrétisations et des symbolisations via un langage de règle CSml intégré dans le calcul du prédicat de chemin permettant de définir des politiques. Cela permet un réglage fin de la correction et de la complétude;
- une variante de l'algorithme de DSE mais fonctionnant en arrière et de manière bornée BB-DSE. Cet algorithme permet la résolution de certains problèmes d'obfuscation.

Contribution #2 Les algorithmes ont été implémentés dans différents outils, Binsec/se, Pinsec et Idasec puis testés sur des exemples concrets. Ces trois outils sont respectivement le moteur d'exécution symbolique, le moteur d'instrumentation dynamique et un plugin IDA pour l'exploitation des données. En particulier Binsec/se intègre différentes optimisations novatrices en terme de DSE dont la plus notable est le STMC-Read-over-Write permettant une gestion optimisée des lectures/écritures en mémoire. Ces outils ont été testés avec succès aussi bien sur des malware Windows que des binaires Linux tels que les *coreutils*.

Contribution #3 En sus, trois combinaisons d'analyse ont été élaborées afin de répondre à des problématiques annexes telles que le désassemblage ou la détection de vulnérabilités. Les trois combinaisons sont :

- combinaison de désassemblage dynamique et de désassemblage statique contrôlé par du BB-DSE. Le but est de l'appliquer à l'obfuscation afin de fournir un désassemblage plus précis tirant parti des informations d'obfuscation calculée par le BB-DSE,
- détection de Use-After-Free en collaboration avec le laboratoire Vérimag,
- détection de critères de tests infaisables appliquée dans une thématique de test sur du code source. Cette combinaison se base sur une combinaison entre interpretation abstraite en avant et calcul de plus faible-précondition (WP).

Contribution #4 La dernière contribution majeure est l'étude, la formalisation et l'expérimentation de deux algorithmes de détection d'obfuscation appliqués respectivement à deux obfuscations: les prédicats opaques et les corruptions de pile d'appel. Une validation expérimentale des deux algorithmes a été effectuée avec succès sur un large échantillon de packers et sur certains malwares comme X-Tunnel. Ce dernier développé par le groupe APT28/Sednit a pour but l'exfiltration de données et fut connu pour son utilisation lors du piratage du Comité national démocrate aux Etats-Unis [Alp16]. L'analyse de ce dernier a mis en évidence la présence de très nombreux prédicats opaques ayant pu être supprimés grâce à la méthode de détection sus-citée.

État de l'art

Obfuscation

D'un point de vue théorique un obfuscateur est défini par Barak [Bar+12] comme une transformation $\mathcal{O}$ sur un programme P satisfaisant trois propriétés : la fonctionnalité, le ralentissement polynomial et la boite-noire virtuelle. De cette définition découle l'impossibilité de faire un obfuscateur parfait dans le cas général. Cependant en pratique, de nombreux obfuscateurs existent. On approchera donc le problème de l'obfuscation d'un point de vu pragmatique avec la définition donnée par Collberg [CN09]. En effet, il caractérise une obfuscation à partir de 4 critères : l'efficacité, la puissance, la furtivité et le coût se referrant à la propriété de ralentissement polynomial.

Code Vs données Une obfuscation affecte généralement soit le flot de contrôle soit les données d'un programme. Une obfuscation du flot va tenter de brouiller la structure logique des instructions et des fonctions alors qu'une obfuscation de donnée va essayer de cacher certaines ressources du programmes comme des chaînes de caractères ou des constantes. Parmi les obfuscations de contrôle, on peut citer le "code flattening" servant à applatir le CFG, les obfuscations par signaux et exceptions servant à faire des sauts non-explicites ou encore le "code overlapping" permettant à deux instructions différentes de partager certains octets. Parmi les obfuscations de données, on peut citer le chiffrement des chaînes de caractères ou encore le "mixed-boolean arithmetic" [Zho+07] permettant notamment de cacher certaines constantes ou clés en les décomposant.

Désobfuscation

Le terme *désobfuscation* regroupe toute technique visant à contourner ou enlever les protections sus-citées. Elle peut donc être divisée en deux étapes différentes : la détection de l'obfuscation et la suppression, cette dernière étape étant potentiellement beaucoup plus complexe à mettre en œuvre. Basé sur les définitions ci-dessus, voici les trois actions envisageables une fois la détection effectuée :

- inverser la transformation d'obfuscation (généralement impossible),
- récupérer une information précise dans le code obscurci,

- simplifier le code obscurci pour faciliter des analyses ultérieures.

Nous orientons nos recherches vers la dernière possibilité, étant donné que le but de cette thèse est de faciliter l'analyse d'un ingénieur en retro-ingénierie.

État de la technique Dans l'industrie les méthodes employées sont généralement statiques ou dynamiques et dans la plupart des cas, syntaxiques. Ces approches sont parfaitement viables mais néanmoins peu résistantes à la combinaison d'obfuscation ou l'apparition de nouvelles obfuscations. En effet, l'analyse statique est souvent syntaxique et basée sur des heuristiques. L'analyse dynamique quant à elle, s'effectue généralement dans une sandbox ou directement manuellement via un debugger.

Méthodes formelles Les approches formelles statiques se basent pour certaines sur l'interprétation abstraite [Dal+06; Kin12] et visent particulièrement les techniques de machines virtuelles ou de "code flattening". Cependant, ces analyses considèrent souvent le CFG comme disponible et immuable sans aucune auto-modification. Ceci limite donc beaucoup le domaine d'application. A l'inverse, les approches dynamiques cherchent à contourner le polymorphisme ou le chiffrement des données [KPY07; Uga+15]. Les approches dynamiques étant généralement plus lentes, il est parfois difficile de les mettre en œuvre sur un grand nombre de programmes. Enfin, plusieurs approches symboliques ont été expérimentées pour traiter différents problèmes d'obfuscation [Bru+07; LPG13; YD15]. Cette thèse s'inscrit dans cette ligne de recherche et propose des algorithmes permettant le passage à l'échelle sur du code obscurci, ce qui reste le principal défaut des techniques sémantiques existantes.

Attaques sur la désobfuscation Une obfuscation tente généralement d'empêcher soit les techniques d'analyse statique soit les techniques d'analyse dynamique. L'analyse statique est hautement dépendante du graphe de flot de contrôle (CFG), toute obfuscation le ciblant est donc potentiellement efficace. L'obfuscation dynamique cherche généralement à empêcher l'exécution ou le debug du programme dans une machine virtuelle ou un environnement instrumenté. Enfin très peu d'obfuscations ciblent directement l'exécution symbolique car la mise en œuvre est difficile. Le concept de "secure triggers" [Fut+06] exploite généralement la difficulté d'inverser les fonctions de hachage pour protéger leur code et rendre inefficace toute analyse symbolique.

Exécution Symbolique Dynamique

L'*exécution symbolique* (SE) est une approche qui, pour un chemin donné dans un programme, va calculer un ensemble de relations sous forme d'une formule pour laquelle une solution est un jeu d'inputs permettant d'emprunter ce chemin [Kin76]. Il est ensuite possible d'énumérer les chemins pour couvrir et tester tout le programme. L'*exécution symbolique dynamique* ou encore *exécution concolique* utilise une exécution concrète (réelle) afin de déterminer le chemin à calculer. Les avantages sont nombreux : l'exécution est correcte (faisable en pratique), l'instruction suivante est toujours connue et le dépliage

des boucles est automatique. Cette approche est donc robuste aux différentes constructions inhérentes au binaire. Néanmoins, l'exécution est sous-approximée et dépendante des entrées utilisées. L'exploration est donc partielle (incomplète).

Prédicat de chemin Le calcul du prédicat de chemin est la composante essentielle du DSE. Pour un chemin π, φ est un prédicat de chemin de π si pour toute valuation des entrées t (solution de φ), l'exécution de programme P sur t noté $P(t)$ emprunte le chemin π. Il est *correct* si toutes les solutions couvrent π et il est complet si tout jeu d'entrées couvrant π est une solution. Intuitivement, un prédicat de chemin est la conjonction de toutes les conditions de sauts conditionnels rencontrées sur ce chemin. En DSE, sur un chemin concret, il est possible de remplacer des valeurs logiques par leurs valeurs lors de l'exécution afin de simplifier le prédicat de chemin (*concrétiser*). A l'inverse, il est aussi possible de *symboliser* une valeur afin de simuler des valeurs inconnues ou des effets non-déterministes. Ces deux mécanismes sont un point essentiel pour une analyse robuste et offrent un compromis entre correction et complétude. La Figure 1 montre respectivement un programme, le prédicat de chemin associé, le même prédicat où $a = 5$ a été concrétisé et le même prédicat où x_1 a été symbolisé avec un nouveau symbole. Faire varier les concrétisations et les symbolisations permet d'ajuster et choisir un compromis entre précision et complétude.

programme	prédicat chemin φ_1	concrétisation φ_2	symbolisation φ_3
inputs: a, b		a = 5	
x := a × b	$x_1 = a \times b$	$x_1 = 5 \times b$	$x_1 = fresh$
x := x + 1	$\wedge\ x_2 = x_1 + 1$	$\wedge\ x_2 = x_1 + 1$	$\wedge\ x_2 = x_1 + 1$
//assert x > 10	$\wedge\ x_2 > 10$	$\wedge\ x_2 > 10$	$\wedge\ x_2 > 10$

Figure 1: Prédicat de chemin, concrétisation et symbolisation

DSE au niveau binaire Toutes les analyses sont développées au niveau binaire sur un langage intermédiaire Dynamic Bitvector Automata (DBA) [DB15] fournissant une représentation concise et unifiée des instructions, indépendante du jeu d'instructions analysé. Chaque instruction assembleur se décompose en une ou plusieurs instructions DBA encodant la sémantique de l'instruction. DBA comme les autres représentations intermédiaires (IR) existantes est sans effet de bord et précis au niveau bit. Les valeurs sont représentées sous forme de bitvecteurs $bv \in Bv$ dont la taille est connue statiquement. Le cœur du langage est composé de quatre instructions : l'assignation $lhs := e$, le saut statique *goto* l, le saut dynamique *goto* e et le saut conditionnel $ite(c)?$ *goto* l ; *goto* l. Un lhs est soit une variable $v \in Var$ soit une écriture en mémoire @[e]. Une expression e est définie par $e ::= e \lozenge_b e \mid \lozenge_u e \mid @[\ e\ ] \mid v \mid bv \mid \bot \mid \top$ respectivement une opération binaire, une opération unaire, une lecture en mémoire, une variable, une constante de valeur indéfinie et valeur non-déterministe. L'exécution symbolique dynamique est donc calculée sur cette représentation intérmédiaire.

Contribution #1: Amélioration du DSE pour la désobfuscation

DSE flexible via des politiques de concrétisation/symbolisation

Un des leviers disponibles pour améliorer le passage à l'échelle du calcul de prédicat de chemin est la modulation des concrétisations et des symbolisations. Alors que les approches existantes abordent le problème comme un élément mineur de l'implémentation nous proposons un nouvel algorithme de calcul du prédicat de chemin intégrant un système de politique de concrétisation/symbolisation permettant de moduler le comportement du calcul de chemin (cf 1 via un langage de règles [Dav+16a]. Pour une variable donnée, trois actions sont possibles :

- **Concrétiser** ($\mathcal{C}$) remplace la valeur logique par la valeur à l'exécution. Cela sous-approxime le prédicat de chemin mais réduit la complexité de la formule,
- **Symboliser** ($\mathcal{S}$) remplace la valeur logique par un nouveau symbole (nouvelle variable libre). Cela sur-approxime en généralisant certaines valeurs du prédicat de chemin et permet donc de simuler des valeurs complètement inconnues (retour de fonctions, etc),
- **Propager** ($\mathcal{P}$) calcule la valeur logique sans aucune approximation (comportement par défaut).

Politique de C/S & CSml Le but est de définir un mécanisme clair et flexible permettant de moduler les concretisations et symbolisations tout en assurant la correction et ce pour chaque expression de l'exécution. L'idée est de concevoir un langage de spécification haut-niveau pour les politiques de C/S fournissant les propriétés suivantes:

- un langage et une sémantique claire qui soient simple et concis,
- indépendant vis-à-vis du moteur de DSE,
- possiblité d'encoder toutes les politiques de la litterature.

Cela a été effectué via CSml intégré dans Binsec/se offrant un langage de règles de C/S. Une règle est de la forme $guard \Rightarrow \rho$ où $guard$ permet de vérifier si la règle doit être déclenchée et ρ indique l'action à effectuer dans ce cas ($\{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$). Chaque règle est testée séquentiellement, la première satisfaisant la condition de $guard$ renvoie l'action associée. Formellement $guard$ est défini par $(\phi_{loc}, \phi_{ins}, \phi_{expr}, \phi_{\Sigma})$ qui sont respectivement un prédicat sur le point de contrôle, l'instruction, l'expression et l'état mémoire. Ces différents prédicats permettent de filtrer précisement l'expression sur laquelle effectuer l'action associée.

Expérimentations Différentes expérimentations on été effectuées formant la première évaluation quantitative et systématique des politiques de C/S dans le DSE. Nous avons évalué différentes politiques issues de la litterature, traitant les accès mémoires de manière différente. Cinq politiques ont été évaluées sur un total de 169 programmes tirés de différentes suites de benchs et des *coreutils* de Linux. Les résultats montrent qu'aucune politique ne surpasse toutes les autres et qu'en fonction du besoin, le choix de la politique peut avoir une grande influence. Ces résultats ont validé *a posteriori* le besoin d'un méchanisme générique et flexible pour moduler les C/S.

DSE arrière-borné

Le DSE appliqué en avant avec des politiques de C/S permet un meilleur passage à l'échelle sur de gros prédicats de chemins. Cependant, il est parfois insuffisant voire inadapté pour résoudre certains problèmes, en particulier les problèmes d'*infaisabilité*. On définit ce problème par : *Une requête d'infaisabilité vise à vérifier l'infaisabilité de certains évènements où de certaines configurations dans un environement donné.* Par exemple l'impossibilité de prendre une branche, de sauter à une adresse particulière ou encore d'avoir une valeur précise dans un registre donné d'une adresse particulière. Le DSE classique (en avant) vise à résoudre des problèmes de *faisabilité* (i.e. ateignabilité, faisabilité de chemin, valeur de registre etc) et ce, afin de générer de nouvelles entrées satisfaisant le problème à résoudre. Plusieurs problèmatiques d'obfuscation entrent dans le cadre des problèmes d'infaisabilité (détection de code mort, etc).

BB-DSE Le BB-DSE [DBM16] est un algorithme de DSE fonctionnant en arrière, et ce, de manière bornée pour assurer le passage à l'echelle. Il s'agit d'une approche orientée par les buts dans laquelle l'analyse démarre du point de contrôle de la propriété à prouver et calcule le prédicat de chemin en arrière un nombre borné k d'instructions. La principale conséquence du point de vue de la formule est de sur-approximer tous les états possibles, au-delà de la borne k. Une propriété prouvée insatisfiable sur un sous-chemin k reste donc insatisfiable sur le sous-chemin $k+1$. En effet, l'ajout de contraintes ne peut pas rendre une formule à nouveau satisfiable pour des raisons de monotonie. La figure 2 illustre le fonctionnement de l'algorithme. Cet algorithme ne remplace pas le DSE classique mais le complète pour un certain type de propriétés à prouver.

Formellement le DSE se base sur l'opération *post* tandis que le BB-DSE se base sur l'opération *pre* et plus particulièrement sur un nombre borné d'itérations de la fonction notée $pre^{\leq k}$. La problématique est d'ajuster la borne k afin de limiter le nombre de faux négatifs ou de faux positifs. Les expérimentations effectuées dans la **Contrib#4** démontrent l'utilité de l'approche pour certains problèmes d'obfuscation comme les prédicats opaques ou corruption de pile d'appel.

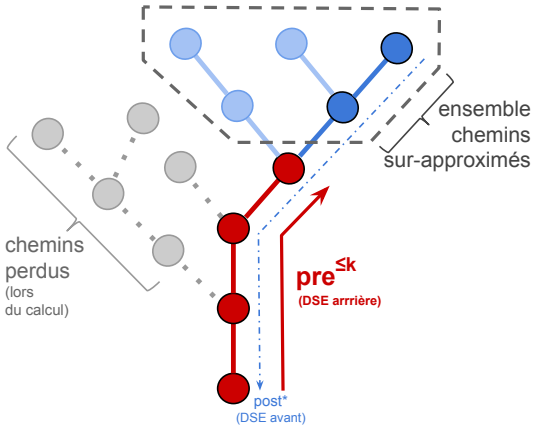


Figure 2: schéma pre^k

Contribution #2: Implémentation

Introduction

Nos algorithmes et techniques ont été implémentés dans trois composants : l'instrumenteur dynamique (DBI), le moteur de DSE et le solveur de formules. Le schéma 3 montre les interactions entre les différents composants. La résolution de formule est traditionnellement effectuée par des solveurs SMT ou CP externes tel que `Z3` [MB08], `CVC4` [Det+14] ou `boolector` [NPB15]. Cette thèse a mené à l'implémentation d'une instrumentation dynamique (Pinsec) et d'un moteur de DSE (Binsec/se). Enfin, un plugin IDA appelé Idasec a été développé pour l'exploitation des résultats d'analyse et les opérations de haut niveau.

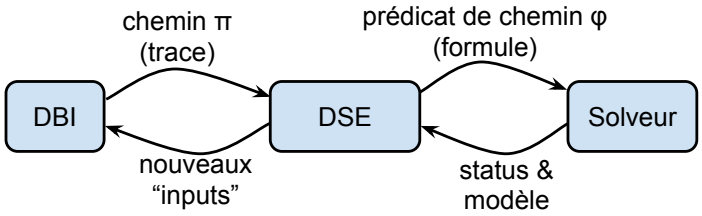


Figure 3: Schéma d'interaction des composants d'un DSE

Pinsec est basé sur Pin [Luk+05] permettant l'instrumentation de programme x86. Pinsec permet d'instrumenter des programmes Linux et Windows afin d'en générer une trace d'exécution contenant toutes les valeurs des registres et cases mémoire à l'exécution. Il permet aussi de récupérer les paramètres des fonctions via des "stubs", de récupérer ou de patcher à la volée des valeurs arbitraires lors de l'instrumentation ou encore de communiquer avec le DSE via un protocole d'échange de messages.

Binsec/se est développé dans la plateforme d'analyse Binsec [DB15] et y apporte un moteur de DSE [Dav+16b]. Le moteur supporte tous les algorithmes mentionnés ci-dessus et fournit un grand choix d'options de configuration. Il est notamment possible de paramétrer la politique de C/S, la convention d'appel, le solveur, les optimisations sur les formules ou encore le timeout. Les options peuvent notamment être fournies sous forme d'un fichier de configuration en JSON. Avec cela, n'importe quelle analyse peut être implémentée via les API fournies. Par ailleurs, Binsec/se peut recevoir et traiter directement les requêtes par le réseau. En ce qui concerne les solveurs il supporte `Z3`, `CVC4`, `boolector` et `Yices` grâce à une intégration boite noire via le format SMTLIB2.

Idasec est développé en Python et s'intègre au désassembleur IDA [Hex16] via son mécanisme de plugins. Parmi ses fonctions basiques, il permet d'obtenir la mnémonique d'une instruction, visualiser une trace d'exécution ou encore surligner le chemin emprunté sur le CFG. Ce plugin peut se comporter comme un intermédiaire entre Pinsec et Binsec lorsque les analyses sont effectuées à distance en flux continu.

Optimisations des formules

L'optimisation des formules générées est une composante essentielle pour le passage à l'échelle des analyses et compte parmi les contributions de cette thèse. Les optimisations présentées ci-dessous ambitionnent une réduction soit en espace soit en temps. Le gain de temps est privilégié car la contrainte de temps est le principal facteur limitant en DSE. Parmi les optimisations développées nous distinguons deux categories ; (1) les optimisations offrant un pré-processing minimal visant à uniformiser les formules générées ce qui, de manière opportuniste, améliore les performances de certains solveurs et (2) les optimisations dédiées à la simplification des tableaux.

Optimisations de pré-processing Les différentes optimisations de pré-processing implémentées sont:

- *backward pruning*, vise à retirer tous les termes d'une formule n'intervenant pas dans la résolution des contraintes,
- *propagation constante*, optimisation standard qui sans améliorer significativement le temps de résolution ou la taille de la formule, produit une forme plus canonique
- *rebase*, réduit le nombre d'identifiants de variables crées via la réutilisation d'identifiants plus anciens et la simulation des opérations arithmétiques. Par exemple, dans le cas $esp_1 := esp_0 - 4$; $esp_2 := esp_1 - 4$ la variable esp_2 est redéfinie par $esp_2 := esp_0 - 8$. L'avantage est de fournir une base de comparaison commune entre esp_1 et esp_0 en les rendant facilement comparables vis-à-vis de esp_0.
- *prédicats haut-niveau*, vise à remplacer des opérations de comparaison bas niveau par des conditions plus naturelles. Par exemple la condition associée à la comparaison $cmp\ ecx, 24$; $jl\ xx$ en `x86` est $((ecx - 24) <_s 0) \neq (ecx^{\{31\}} \neq 24^{\{31\}})\&\&(ecx^{\{31\}} \neq (ecx - 24)^{\{31\}})$ alors que l'opération de haut niveau est : $ecx < 24$.

Optimisations de tableaux Le Read-over-Write est une propriété sur la théorie des tableaux qui indique que pour un tableau donné il est logiquement équivalent de substituer la lecture à un index donné par la valeur précédement écrite à ce même index. Effectuer cette opération permet donc de réduire le nombre de lectures en mémoire. Dans la théorie des tableaux, l'opération *write* renvoie un nouveau tableau mis à jour avec la valeur écrite. Un tableau est donc une suite, une séquence d'écriture dans un tableau initial vide. Effectuer l'opération de RoW nécessite donc de parcourir linéairement cette chaîne et ce, pour chaque opération *read*. La complexité est donc approximativement au pire cas quadratique en la taille de la chaîne de *write*. Une des contributions de ma thèse est de proposer un *modèle novateur* de représentation des écritures permettant d'effectuer cette optimisation en temps logarithmique dans le meilleur cas ou identique en complexité à l'approche classique dans le pire cas. L'efficacité dépend notamment de la politique de C/S employée vis-à-vis des lectures/écritures en mémoire. Cette thèse compare les deux approches et met en avant le gain en terme de temps de résolution des formules avec le store-map RoW. En sus, le *memory flattening* est une deuxième optimisation sur les tableaux qui sert à supprimer la théorie des tableaux de la formule lorsque le Read-over-Write à été suffisamment efficace pour supprimer tous les *store* et ramener tous les *read* sur la mémoire initiale à des indices constants.

Contribution #3: Combinaisons d'analyses

Désassemblage minutieux

Une instruction est *authentique* ou avérée si lors de l'exécution du programme elle est exécutée. Un désassembleur est *sûr* s'il désassemble uniquement des instructions authentiques (et non pas des données). Il est dit complet s'il retrouve toutes les instructions authentiques du programme. Les trois algorithmes de désassemblage classiques sont:

- le *désassemblage récursif* suit le flot du programme pour le désassembler. Cette approche est rapidement limitée par les sauts dynamiques (saut à une valeur connues seulement à l'exécution);
- le *balayage linéaire* désassemble séquentiellement chaque octet des sections exécutables d'un binaire. Le problème de cette approche est le "*sur-désassemblage*". En effet, de nombreux octets de padding ou de données seront décodés comme des instructions. L'approche n'est donc pas sûre;
- le *désassemblage dynamique* utilise les branches observées lors d'une exécution pour déterminer les cibles de saut. Cette méthode est sûre mais incomplète dans la mesure où une seule exécution est couverte.

Combinaison Le but de la combinaison proposée est d'enrichir un algorithme de désassemblage dynamique avec un désassemblage statique récursif contrôlé grâce à des informations d'obfuscation calculées par exécution symbolique. L'idée est donc, non pas

de désassembler le plus d'instructions possibles, mais de désassembler plus justement les instructions.

Les enrichissements par rapport à un algorithme classique sont les suivants :

- utilisation des valeurs dynamiques pour désassembler les cibles de saut dynamique (comme pour un désassemblage dynamique),
- utilisation des informations de prédicats opaques pour ne pas désassembler les branches mortes,
- utilisation des informations de corruption de pile d'appel pour désassembler les cibles de saut d'un `ret` corrompu et ne pas désassembler le `returnsite` d'un call dans lequel on ne retourne jamais.

Evaluation Les benchmarks effectués visent à évaluer la précision du désassemblage de la combinaison par rapport aux algorithmes classiques dans les outils dits grand public. La comparaison est faite avec `Objdump` fonctionnant en balayage linéaire, IDA mélangeant désassemblage linéaire et récursif et BINSEC en mode récursif. Le tableau 1 montre les résultats sur cinq binaires obscurcis avec des prédicats opaques. Les résultats montrent que IDA, `Objdump` et BINSEC (récursif) désassemblent les branches mortes alors que la combinaison dite minutieuse ne désassemble que les instructions réellement exécutables et offrant un gain maximum de 32%.

Table 1: Désassemblage "sparse" de prédicats opaques

	pas obf.	parfait	IDA	Objdump	BINSEC		gain vs IDA
					rec.	**sparse**	(sparse)
simple-if	37	**185**	240	244	240	**185**	23,23%
huffman	558	**3226**	3594	3602	3594	**3226**	10,26%
matrix_multiply	249	**854**	1075	1080	1075	**854**	20,67%
bin_search	105	**833**	1110	1115	1110	**833**	24,95%
bubble_sort	121	**1026**	1531	1537	1531	**1026**	32,98%

Détection de vulnérabilité de Use-After-Free

Introduction Ces travaux ont été effectués en collaboration avec le laboratoire Vérimag et plus particulièrement Josselin Feist. Cette combinaison a pour but d'étendre les analyses symboliques développées au niveau binaire mais appliquées à de la détection de vulnérabilité et plus particulièrement des Use-After-Free (UaF). Cette vulnérabilité permet si elle est exploitée, l'exécution arbitraire de code grâce à la réutilisation impropre de pointeurs préalablement libérés. Cette combinaison s'articule autour d'une analyse statique par interprétation abstraite visant à extraire un ou plusieurs *slice* de programmes contenant les évenements clés d'un UaF c-a-d $malloc, free$ et use. Ensuite, une couverture

des chemins est effectuée en DSE sur ces *slice* afin de détecter la présence d'une vulnérabilité. En complément différentes heuristiques dédiées ont été implémentées afin de guider l'exploration des chemins de manière plus efficace dans le contexte de la détection d'UaF. L'analyse statique effectuant l'extraction du *slice* et les heuristiques de choix de chemins ont pour but commun de contourner le problème de *l'explosion combinatoire des chemins* inhérent à l'exploration des chemins en DSE.

Validation expérimentale Cette combinaison a permis la détection de plusieurs vulnérabilités. La plus notable est une vulnérabilité dans la librairie de traitement d'image JasPer ayant menée à la CVE-2015-5221 [Mit15]. Cette vulnérabilité a mis en évidence l'utilité de la combinaison dans la mesure où des outils de fuzzing échouent à détecter la vulnérabilité. Par ailleurs, les heuristiques de guidage pour le DSE réduisent significativement le temps d'énumération des chemins menant à la détection de la vulnérabilité.

Détection de critères de couverture infaisables [code source]

Introduction Ces travaux de thèse et en particulier les différentes méthodes de combinaisons étudiées ont donné lieu à une application dans le domaine du test logiciel au niveau source. Pour une campagne de test, certains critères de couverture doivent être satisfaits afin de considérer le logiciel comme testé. Les critères connus sont : la couverture de décision (DC), la couverture de condition (CC) ou encore les conditions de couverture multiples(MCC). Afin de tester ces critères nous utilisons les labels [Bar+14] comme formalisme pour encoder ces différents critères de manière générique. Concrètement, au niveau C, un label est une annotation de code traitée ensuite par le moteur de test afin d'en vérifier la validité.

Problématique Ajoutés automatiquement dans tout le code, certains de ces critères se trouvent être infaisables de par la structure du programme. Outre la réduction du taux effectif de couverture, beaucoup de temps est perdu afin d'essayer de couvrir ce critère. Il est donc important de pouvoir détecter en amont ces différents critères infaisables.

Combinaison Inspiré des combinaisons et techniques précédentes, mes travaux proposent une combinaison avant/arrière permettant la détection des critères de couverture infaisables. Cette combinaison se base sur une analyse de valeurs (VA) par interprétation abstraite pour l'analyse en avant et sur du calcul de plus faible précondition (WP) pour l'analyse en arrière. Le but est de calculer une sur-approximation du programme par VA afin d'inclure certains des ces invariants comme hypothèse (donc considéré valide) sous forme de prédicat ACSL [Cuo+12]. Le but est d'améliorer le pouvoir de décision du WP, et ce, sans aucune annotation supplémentaire. Cette combinaison VA $\oplus$ WP est donc effectuée en boîte grise grâce à cette communication d'invariants entre VA et WP. Cette analyse s'effectue en trois étapes: (1) le calcul (une seule fois) de la sur-approximation des états avec VA, (2) pour tous les prédicats non-valides, l'extraction et ajout comme hypothèses des invariants relatifs aux variables du prédicat. (3) analyse WP pour vérifier la validité du prédicat.

Implémentation Cette analyse est développée dans la plateforme Frama-C qui intègre deux greffons, Value et WP, permettant respectivement le raisonnement avant et arrière. La campagne de test est effectuée avec LTest [Bar+14] et Pathcrawler [Wil+05], eux aussi deux greffons de Frama-C.

Expérimentations Des benchmarks ont été effectués sur 12 fichiers issus de la suite Siemens et de la suite Verisec. Des critères de couverture usuels tels que la couverture de branches ou encore la couverture de conditions ont été étudiés et encodés sous forme de labels. Sur un total de 1270 critères de tests, 121 se sont révélés infaisables avec une vérification manuelle. En sus, 84 ont été détectés par VA et 73 par WP tandis que la combinaison des deux a permis d'en détecter 118 soit 98%. De façon intéressante, la combinaison permet de détecter certains critères que ni VA ni WP ne sont capables de détecter seuls.

Contribution #4: Évalution expérimentale

L'efficacité des différents algorithmes de DSE développés a été validée expérimentalement sur deux obfuscations différentes et appliquées à deux cas d'utilisation. Les deux obfuscations traitées sont les prédicats opaques et les corruptions de pile d'appel pour lesquels deux méthodologies de détection sont formalisées et implémentées. Ces deux algorithmes ont ensuite été évalués expérimentalement sur un large ensemble de packers et sur le malware X-Tunnel.

Prédicats Opaques

Définition La première obfuscation traitée dans cette thèse est celle basée sur les *prédicats opaques* (PO). Un PO est une condition s'évaluant toujours à vrai (ou toujours faux) mais dont cette propriété est difficile à déduire à priori. Cette technique permet notamment de gonfler artificiellement le code d'un programme en ajoutant du code mort dans les branches mortes.

Taxonomie La première taxonomie des PO fut proposée par Collberg [CTL97] et présente différentes manières d'encoder des invariants dans les programmes servant ensuite à la création de PO. Les différents type d'invariants proposés sont :

- **Invariants arithmétiques** qui utilisent généralement des propriétés d'arithmétique modulaire pour créer des équations sans solution ou bien toujours vraies. Par exemple, $7y^2 - 1 \neq x^2$ est toujours vrai quelques soient les valeurs de x ou y et ce même avec les overflow possibles sur des entiers machine;
- **Invariants de structure de données** qui utilisent certaines propriétés des structures de données pour encoder les prédicats. Par exemple on peut créer un tableau dont toutes les cases paires sont divisibles par 2 ($x \ mod \ 2 = 0$) ou encore sur une liste triée où $x < x + 1$ est toujours vrai;

- **Invariants sur les pointeurs** qui se basent sur la difficulté de faire des analyses d'alias et donc savoir si deux pointeurs sont équivalents. Toutes les manières sont bonnes et ce genre d'invariant est facile à créer au niveau source;

- **Invariants de concurrence** qui se basent sur des invariants créés avec des "*race condition*" entre différents threads temporisés de manière adaptée;

- **Invariants d'environement** qui correspondent à tout invariant lié au système, typiquement en x86 la pile est alignée donc $esp \bmod 4 = 0$ est toujours vrai.

Détection Dans ma thèse, la détection se focalise sur les PO arithmétiques plus largement répandus. Le but est de détecter l'infaisabilité d'une des deux branches d'un prédicat. Le BB-DSE est particulièrent adapté à ce genre de problème. Toute la difficulté consiste à choisir la borne k adaptée pour englober toutes les dépendances du calcul du prédicat. De par l'aspect dynamique du DSE, une des branches a nécessairement été prise, il ne reste plus qu'à tester la "prenabilité" de l'autre branche par BB-DSE. À noter que le $pre^{\leq k}$ sur une trace teste uniquement un sous-chemin. L'algorithme va donc tester tout nouveau sous-chemin candidat pour essayer de déterminer l'opacité du prédicat. Si le chemin est insatisfiable alors le prédicat est opaque; si le chemin est satisfiable alors il n'est pas possible de statuer dans la mesure où un sous-chemin n'a peut-être pas été testé. Dans ce contexte, un faux positif est un prédicat faussement identifié comme opaque et un faux négatif un prédicat opaque non identifié comme tel. A noter que le $pre^{\leq k}$ est incomplet et peut rater des chemins. L'analyse peut donc déclencher des faux positifs et des faux negatifs (prédicats opaques ratés).

Benchmarks Plusieurs benchmarks ont été effectués. Les benchmarks se basent sur O-LLVM [Jun+15] fournissant des passes d'obfuscation dans LLVM et visent à tester l'efficacité de l'analyse en environement contrôlé sur des binaires connus. Pour cela, tous les *coreutils* Linux ainsi que 5 binaires de test ont été obfusqués puis analysés. L'obfuscation intégrant de l'aléa, il fut possible de dériver plusieurs versions obfusquées d'un même binaire et ce, afin d'obtenir 200 binaires obscurcis. Les benchmarks ont été effectués avec différentes valeurs de k afin d'évaluer la valeur la plus adaptée. Les résultats ont mis en évidence la valeur **k=16** comme étant la valeur optimale, maximisant la détection des PO tout en limitant les faux positifs (3.17%) et les faux négatifs (0%).

Corruption de pile d'appel

Définition La corruption de pile d'appel désigne l'action de violer l'assertion selon laquelle une instruction `ret` retourne à l'instruction suivant le précédant `call` (appelé `returnsite`). Dans ce cas le `ret` est dit violé/falsifié. Au niveau machine, un `ret` est un saut dynamique sur l'adresse en haut de pile. Cette adresse est donc facilement modifiable afin de brouiller le flot de contrôle et brouiller la détection des bornes de fonction. L'exemple de violation le plus simple est le pattern $push\ xx; ret$ qui écrit une valeur sur la pile puis saute à cette adresse avec le `ret`.

Taxonomie Peu de travaux ayant été réalisés à ce sujet, cette thèse propose une taxonomie des différents types des violations/corruptions possibles caractérisées par l'alignement, l'intégrité et la multiplicité. Cette caractérisation permet d'obtenir des indices quant à la nature de la violation, à savoir, un bug ou une action intentionnelle. Les trois caractéristiques sont:

- *l'intégrité*, c-à-d la valeur de retour a-t-elle été modifiée? Autrement dit : y-a-t'il eu une corruption ? Les statuts sont désignés par `[genuine]` ou `[violated]`,
- *l'alignement*, c-à-d le `ret` retourne-t-il au même offset sur la pile ? Autrement dit retourne-t-il avec la même valeur pour `esp`. Les deux statuts possibles sont `[aligned]`, `[disaligned]`,
- *la multiplicité* de corruption, plusieurs valeurs de sauts sont-elles possibles ? Les statuts sont `[single]` ou `[multiple]`.

Détection La détection se base sur le BB-DSE. La différence se situe dans le choix de la borne k, qui sera ici adapté à la distance entre le `ret` et le `call` afin de pouvoir déterminer si le retour s'effectue à la bonne adresse. Afin de caractériser la corruption trois requêtes SMT sont effectuées : (1) test de la valeur pushé au `call` et lue au `ret` pour détecter la corruption, (2) comparaison des valeurs de `esp` pour caractériser l'alignement et (3), test d'une nouvelle valeur de saut en cas de corruption.

Benchmarks Afin de valider l'approche, un benchmark a été effectué avec Tigress [Col+12] permettant l'ajout de corruption de pile d'appel avec les patterns *push*; *call*; *ret*; *ret* et *push*; *ret*. Différents programmes ont été obscurcis de cette manière. Sur 218 `ret`, 77 étaient légitimes et 141 corrompus. L'analyse a obtenu un taux de détection de 100% sans aucun faux positifs ni faux négatifs (à relativiser dû à la faible diversité des patterns inserés).

Évaluation à grande échelle de packers

Binaires analysés Cette étude s'intéresse aux packers qui constituent habituellement, la première, voire l'unique couche de protection pour beaucoup de codes malveillants. Bien que les packers puissent être utilisés pour protéger des logiciels légitimes ils sont tout autant utilisés pour "packer" des virus. Un ensemble de 33 packers open-source et commerciaux sur Windows sont évalués sur un binaire inoffensif (`hostname`) agissant néanmoins comme témoin du bon fonctionnement du packer. Ce benchmark est issu d'une précédente évaluation effectuée dans la litterature [Bon+15].

Critères d'évaluations Les packers sont évalués sur la base des analyses de prédicats opaques et de corruptions de pile d'appel. Effectuer l'analyse en boîte noire sans connaissance préalable des codes analysés est une manière pertinente de tester l'éfficacité des analyses sur de véritables programmes obscurcis. À des fins d'évaluation, le nombre de couches d'auto-modification est aussi enregistré. La trace dynamique générée est limitée

à dix millions d'instructions. Les packers dont la trace atteignait cette limite n'ont pas été analysés car ils n'ont pas atteint le *payload*. L'analyse de prédicats opaques a été effectuée avec $k = 16$, et ce choix se base sur les résultats des précédents benchmarks sur O-LLVM.

Table 2: Expérimentation sur packers

Packers	#Tr.len	Obfuscation detectée					
		Prédicats opaques (k_{16})				Corruption de pile	
		OK	PO	To	Couvert	OK(a/d)	Viol(a/d/s)
ACPacker	1.802.554	73	209	0	9	0 (0/0)	45 (42/1/42)
ACProtect v2.0	1.813.598	74	159	0	9	0 (0/0)	48 (45/1/45)
Aspack v2.12	377.349	32	24	0	136	11 (7/0)	6 (1/4/1)
BoxedApp v3.2	/	-	-	-	-	-	-
Crypter v1.12	1.170.108	263	24	0	136	125 (94/0)	78 (0/30/32)
EP Protector v0.3	250	10	1	0	2	4 (2/0)	0 (0/0/0)
Expressor	635.356	42	8	0	39	14 (10/0)	0 (0/0/0)
Petite v2.2	260.025	60	19	0	45	4 (1/0)	0 (0/0/0)
Yoda's Crypter v1.3	240.900	38	1	0	16	4 (3/0)	9 (0/1/0)

Résultats Le tableau 2 donne les résultats pour quelques-uns des packers analysés. (a/d/s) signifie aligné, désaligné et single pour la caractérisation des corruptions. Les résultats montrent que les deux analyses sont robustes et efficaces sur des traces de plusieurs millions d'instructions. Aussi bien des prédicats opaques que des corruptions de pile d'appel ont été détectés. Ainsi, plus de 150 prédicats opaques et près de 50 corruptions de pile d'appel ont été détectées dans `ACProtect`.

Plus en détail, les résultats permettent de constater que certains packers n'utilisent aucune des deux obfuscations tel que `Expressor`, tandis que d'autres comme `Crypter` ou `ACProtect` les utilisent de manière intensive. Une vérification manuelle a permis de valider la grande majorité des résultats observés. Les résultats laissent aussi à penser que l'instrumentation dynamique a parfois été déjouée par le packer comme c'est peut-être le cas pour `Molebox` ou `EP Protector` dont la trace est étrangement petite (250 instructions). Par ailleurs, l'analyse de corruption de pile d'appel a permis de détecter pour certains packers l'instruction `ret` corrompue effectuant la transition finale entre le code du packer et le payload. Dans ce cas, ce dernier est systématiquement détecté comme violé.

Cas d'étude : X-Tunnel

Contexte X-Tunnel est un proxy chiffrant utilisé pour l'exfiltration de données de machines non-directement reliées à internet. Ce malware a été utilisé lors de différentes attaques ciblées à visées géopolitiques menées notamment contre des infrastructures de l'OTAN [Tre14], le parlement Allemand [von15] ou encore le parti démocrate américain (DNC) [Alp16]. Le groupe derrière ces attaques est dénommé APT28, Sednit ou encore Sofacy group [CCD16].

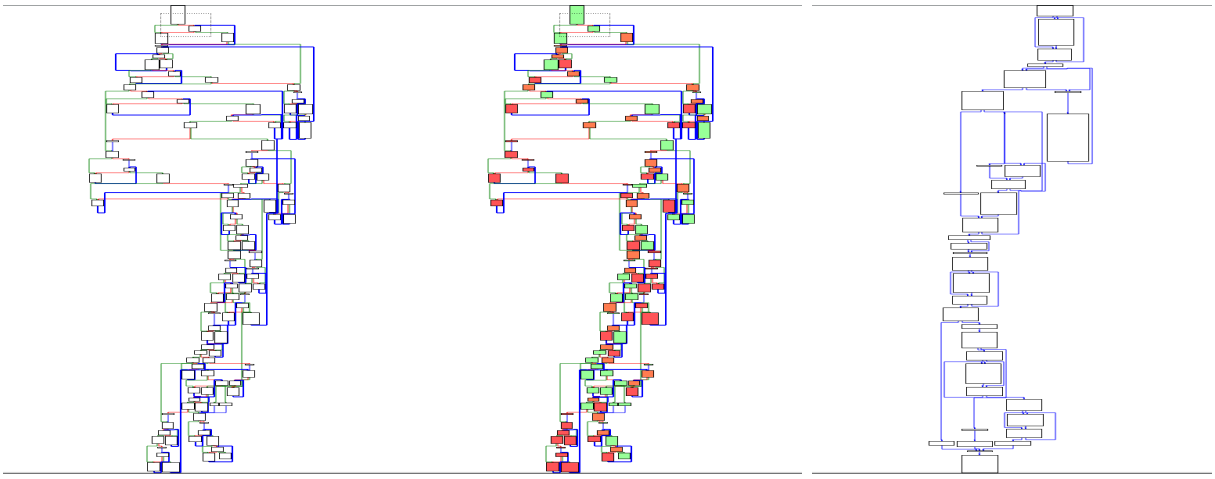
Contexte d'analyse L'analyse se focalise sur 3 échantillons de X-Tunnel compilés à plusieurs mois d'intervalle et dont les deux derniers ont la particularité d'être obfusqués. La nécessité d'analyser les binaires obfusqués est motivée par la question de savoir si de nouvelles fonctionnalités ont été ajoutées. Le malware se connectant à des serveurs C&C et attendant des connexions de clients pour effectuer le tunneling, il est préférable d'effectuer l'analyse de manière complètement statique. De plus, les binaires ne semblent pas utiliser d'auto-modification. L'ensemble du code, bien qu'obfusqué, est donc disponible. Un examen rapide du code a révélé la présence de prédicats opaques. Le principal objectif est de les supprimer afin d'en révéler, éventuellement, de nouvelles fonctionnalités dans une analyse ultérieure.

Développement de l'analyse L'exécution symbolique est effectuée de façon statique (SE) ce qui ne change pas l'analyse de prédicats opaques puisque celle-ci est effectuée de façon complètement symbolique, qui plus est, sur de petits prédicats de chemins (16 instructions). L'analyse développée est guidée par Idasec qui génère des sous-chemins pour chaque prédicat à tester. Ces chemins sont ensuite envoyés pour résolution à Binsec/se configuré en mode serveur. En fonction du résultat renvoyé, Idasec assure le marquage des prédicats opaques et surtout des parties de code mort. Afin de rendre les résultats exploitables, trois traitements ont été implémentés dans Idasec:

- extraction(synthèse) du prédicat opaque haut-niveau (afin de les répertorier),
- identification des instructions *fallacieuses* servant uniquement à calculer un PO,
- extraction d'un CFG réduit basé sur la suppression du code mort et des instructions fallacieuses.

Résultats Pour chacun des 2 échantillons obscurcis, l'analyse a duré environ 1h30 pour environ 35000 prédicats chacun (Core i7-4800MQ 2.7GHz, 16Go RAM). Ce temps inclut toutes les communications réseaux, la création des chemins et le post-traitement. Les deux prédicats identifiés sont $7x^2 - 1 \neq x^2$ et $\frac{2}{x^2+1} \neq y^2 + 3$. Plus de 30% de toutes les conditions dans le code ont été identifiées et validées comme opaque. Une évaluation approximative a permis d'identifier parmi tous les prédicats, 10% de prédicats valides identifiés comme opaques (faux positifs) et seulement 2.3% de prédicats opaques non détectés (faux négatifs). L'analyse a permis de supprimer dans chaque échantillon environ 44% du code identifié comme mort ou inutile. La figure 4 montre un exemple de fonction après extraction du CFG réduit.

En conclusion, l'analyse de prédicats opaques a permis d'élaguer considérablement les fonctions obfusquées et d'en extraire des représentations non-obfusquées beaucoup plus compactes. Les deux programmes une fois désobscurcis obtiennent un nombre d'instructions similaires à la version non-obscurcie. De plus, en vertu de la faible différence de temps entre chaque version, il est peu probable que de nouvelles fonctionnalitées aient été ajoutées.



(a) CFG original (b) CFG marqué (c) CFG extrait

Figure 4: Exemples de CFG avant et après réduction

Conclusion

Résumé des contributions

Variantes du DSE pour la désobfuscation Les principales contributions de ma thèse sont la formalisation, l'implémentation et le test d'un algorithme de calcul de prédicat de chemin pour le DSE intégrant les concrétisations et les symbolisations de manière flexible. Ceci est rendu possible via l'élaboration d'un méta-langage CSML permettant la spécification de politiques. Cette flexibilité a permis de tester et mettre en évidence l'impact des concrétisations et des symbolisations sur le temps de résolution des prédicats. Afin d'accentuer ce gain en temps, cette thèse décrit, formalise et implémente différentes optimisations sur les formules dont la plus notable est le STMC-RoW. Différentes expérimentations ont permis de mettre en évidence la plus-value de ces optimisations sur le temps de résolution.

L'algorithme de BB-DSE est une autre contribution majeure de mes travaux. En effet, peu d'études ont été effectuées sur le DSE arrière et ses avantages, notamment pour les problèmes d'infaisabilité. Cette thèse formalise un algorithme robuste et précis permettant de résoudre ce type de problèmes, appliqué à l'obfuscation. Cet algorithme s'est révelé très efficace pour la détection de prédicats opaques et la détection de corruption de pile d'appel. Bien qu'il ne remplace pas le DSE en avant, cet algorithme le complète.

Étude de cas L'obfuscation étant la pierre angulaire de cette thèse, mes travaux proposent deux algorithmes d'identification et de caractérisation pour deux obfuscations, les prédicats opaques et les corruptions de pile d'appel. Ces algorithmes ont été utilisés avec succès, aussi bien sur des packers commerciaux que sur des codes malveillants. Ces contributions ont permis de mettre en évidence des phénomènes et des structures inattendues. Par exemple, le prédicat opaque $\frac{2}{x^2+1} \neq y^2 + 3$ identifié dans X-TUNNEL n'avait à notre connaissance jamais été vu ailleurs. Autre effet inattendu, l'analyse de corruption de pile

d'appel a permis dans certains cas de mettre en évidence l'instruction précise effectuant la transition entre le code du packer et le payload, lorsque celui-ci utilise l'instruction `ret` pour faire le saut.

Combinaison d'analyses Les combinaisons d'analyses étudiées ont permis de mettre en évidence l'intérêt et les bénéfices de certaines combinaisons avant/arrière et ce, aussi bien sur du code binaire que sur du code source. Dans le cadre de cette thèse, la combinaison la plus notable est le désassemblage "sparse" qui combine un désassemblage statique avant avec une exécution symbolique arrière bornée. La combinaison avec le DSE est d'autant plus efficace qu'elle est intégrée dans l'algorithme et le processus de désassemblage et non pas comme une passe de post-processing.

Contributions à la communauté Une idée clé est de rendre les algorithmes utilisables en pratique par des ingénieurs en rétro-ingéniérie et utilisables au niveau industriel. Pour cela, tous les algorithmes ont été développés dans des outils open-source, notamment Pinsec, Binsec/se et Idasec. Ces trois outils fournissent respectivement une instrumentation dynamique, un moteur d'exécution symbolique et un plugin IDA pour une exploitation plus aisée des données. Ces travaux ont été présenté dans des conférences comme BlackHat Europe 2016 [DB16a] ou bien dédiées à la communauté de la sécurité [BD16].

Perspectives

Cette thèse ouvre la voie à de nombreuses applications et améliorations comme la détection d'obfuscations à base de machines virtuelles ou encore la création de signatures sémantiques du code désobfusqué. Dans ce sens, la prochaine étape clé est d'intégrer les résultats de désobfuscation dans les travaux d'analyse morphologique basés sur de l'isomorphisme de graphes, développés au LORIA. Ceci dans le but de calculer des signatures de malwares plus précises et efficaces. Ainsi, cette thèse a montré que des méthodes formelles telles que le DSE peuvent être employées pour des problématiques de sécurité en fournissant des analyses sémantiques efficaces pour la retro-ingéniérie et l'analyse de malwares. Cela ouvre la voie à de futurs algorithmes de détection sémantique des codes malveillants, potentiellement plus efficaces que les algorithmes actuels.

Part I

Introduction & Background

Chapter 1

Introduction

1.1 Context

Malware is a generic term grouping all software developed with the intention to harm and to threaten computer systems or their users. These malicious softwares have moved from cracking jokes in the late 90' to mass mailing worms like `MyDoom` and banking trojans like `Zeus` [IOA12] in 2007 to evolve nowadays with ransomwares and government-grade cyber-espionage tools like `Stuxnet` or `Flame`. In this last scenario, malwares are the cornerstone of stealth and persistent attacks. These advanced attacks generally involve deception, zero-day exploits, various bots across the globe and a whole ecosystem of malicious softwares like infected documents, droppers, rootkits or bootkit. The sophistication of such attacks has grown tremendously the last few years and the collateral costs it implies for companies become more and more significant. For example, the famous `Stuxnet` malware managed to infect a Siemens Programmable Logic Controller (PLC) used in nuclear centrifuges in the Natanz Iranian nuclear facilities. It presumably destroyed up to 1000 centrifuges amounting for 10% of the total centrifuges assets [Lan13]. Another famous case is the Sony Pictures Entertainment hack in December 2015 that led to the leak of 100 To of data. Among this data, personal employees information and four unreleased movies. The direct investigation and remediation costs were evaluated up to 35M$ and more than 100M$ for the loss of earnings. The usage of a specifically crafted worm [USC14] made this hack possible. At a smaller scale, this kind of sabotage and blackmail occurs frequently. It was notably the case for the Hollywood Presbyterian Medical Center who paid 17K$ to recover data ciphered by a ransomware. A first step toward preventing these infections and understanding the motivations and the origin of such attacks is to analyze these malwares at binary-level.

As no source code is usually available for malware, the binary executable should be analysed directly to understand its behavior and goals. But as this area of research has grown rapidly leading to the emergence of antivirus companies and incident response companies, malware authors have also designed some counter-measures. Indeed, they started to protect their malwares as their main goal is to remain stealth and undetected as long as possible. Among those protections, *obfuscation* takes a significant place. It

embraces any mean aiming at slowing down the analysis of a program, either by an analyst or an automated algorithm. While it has taken a significant place in the software protection field like the video-game industry, it is also now widely used in the malware ecosystem. Thus, addressing obfuscation is one of the main challenges in the malware analysis process.

My thesis was performed in collaboration with two laboratories. First my hosting laboratory, the Safety and Security Lab of the *Commissariat à l'Énergie Atomique et aux energies alternatives* is specialized in formal methods for software analysis. It is most notably known for its open-source C software analyzer Frama-C [Cuo+12]. Secondly, the "Laboratoire de Haute Sécurité (LHS)" of LORIA Nancy focusing on virus analysis, dynamic analysis and notoriously known for its morphological malware signature technology. My whole thesis takes place in an Agence Nationale de la Recherche (ANR) project called Binsec aiming at developing a Binary Analysis platform geared for software security. The project is composed of the CEA, Irisa, LORIA, Vérimag and Airbus Group.

1.2 Research challenges

The first challenge to address toward analyzing malware is to analyze programs at binary-level. Despite the fact that analyzing binary programs helps avoiding some uncertainty about the compilation process [BR10] and is independent from the source language, it also raises some other challenges [BHV11; BR10]. Indeed, at binary-level there is no clear distinction between code and data. Furthermore, all the information available in the source code, like function bounds, function parameters and the control flow graph is not available. Moreover, there is no high-level typing especially for the memory which is seen as a flat untyped array. Thus, making formal and reliable algorithm is a challenging task [MM16]. Finally, the large diversity of processor architectures (`x86`,`ARM`, `MIPS`..) makes the modeling of the binary instruction a tedious task. We solely in this thesis on `x86` architectures as it is the most widespread architecture for desktop computers and by consequence for malwares.

Performing such analyses on malwares is even more challenging as they usually make use of multiple anti-analysis tricks. An inherent goal of obfuscation is to make the behavior of the program less understandable both for the analyst and automated algorithms by swelling the code size and shadowing the real behavior at the same time. Since my thesis focuses on real world malwares, the first research challenge is:

RC1: How to make analysis algorithms scale on real-world obfuscated malwares ?

Dealing with obfuscation can be considered as a two step process. First, it is required to detect the obfuscation with reliable algorithms. Detecting obfuscation is characterized by the ability to locate obfuscation and to differenciate what is part of the obfuscation and what is part of the original program, the payload. Once this step is achieved a second

phase consists in thwarting the obfuscation or providing a non-obfuscated representation of the program. This lead to the second research challenge:

RC2: How to locate and how to remove obfuscation ?

In this context, my thesis focuses on Dynamic Symbolic Execution (DSE) [SMA05; Wil+05] (a.k.a concolic) to detect different obfuscations techniques via generic algorithms. DSE provides good promising properties to detect some obfuscations. However, as it works on a path enumeration basis, an inherent shortcoming of this approach is its difficulty to scale on real programs. This is a limitation that all existing symbolic engines are facing. We do not necessarily aim for sound nor complete DSE-based analyses but rather analyses that provides relevant obfuscation informations. The first research problem addressed is:

Problem 1. *Elaborating relevant DSE-based deobfuscation algorithms that scale on oversized & highly obfuscated codes*

A second problem is how to remove it or at least how to emphazise it in pursuance of distinguishing it from the payload. Performing such task might be particularly difficult without prior knowledge of the program behavior and functionalities. The second problem addressed is:

Problem 2. *Developping algorithms to highlight or discard obfuscation in order to make the code more understandable for the reverse-engineer*

Solving these two research problems would provide great advances in the academic research on binary analysis and malware analysis. Finding ways to address obfuscation with formal and sound algorithm would greatly enhance malware understanding capabilities and would lead to great practical application in the malware analysis field.

1.3 Contributions

Foundations of my thesis lay on the edge of safety based formal methods and dynamic security analysis for malware comprehension. From a global perspective, it aims at studying formal static, dynamic and symbolic approaches for deobfuscation. My thesis gather four major contributions described below.

1.3.1 Contribution #1: DSE geared for deobfuscation

The first main contribution of my thesis is the conception of two DSE algorithms which main particularity is to better scale on obfuscated codes than standard approaches:

- An essential part of DSE is the management of concrete values and new symbols injected in the formula. As it tends to be hardcoded in existing DSE tools, we formalize and define a new flexible path predicate computation algorithm allowing a

fine tuning between soundness and completeness. For that, we define a specification for modulating concretization and symbolization (C/S) within a DSE engine. This is subsequently defined as a rule-based language, CSml, that can easily be integrated in the path predicate computation. This langage is designed to have a clear semantic, to be independant from the DSE engine and to be able to encode all existing policies from the litterature but also new ones. Experiments showed that no policy performs better than another, it all depends on the context. Thus, it validates the need of such flexible mechanisms.

- As standard DSE algorithms are performed in a forward manner, we propose a new algorithm to modulate the symbolic execution by performing it backwardly in a bounded manner. This backward-bounded DSE(bb-dse) formalized in Chapter 5 allows to address specific problems i.e. *infeasibility* queries, while forward DSE handles feasibility queries. *An infeasibility query intents to check infeasibility of some events or some configurations in a given environement.* The main strength of bb-dse is its scalability on any program regardless of the trace length. Experiments performed showed the effectivness of the algorithm to detect *infeasibility* based problems and more especially obfuscation problems.

1.3.2 Contribution #2: Implementation & Optimizations

Implementation To validate these algorithms, all of them have been implemented and validated on real life examples, thanks to the implementation from scratch of a fully featured DSE engine based on Guillaume Jeanne's early work. As a matter of transparency and reproducibility, the whole toolkit is available in open-source [ANR16; Dav16]. This platform is formed by a dynamic instrumentation engine based on Pin [Luk+05] (∼5000loc C++) allowing to generate execution traces and basic debugging interaction. Working on both Linux and Windows, it also provides interesting features like function parameters retrieval or self-modification layers tracking. Secondly Binsec/se, the main DSE engine (∼7000loc OCaml) integrated in the Binsec platform (see Section 6.3.2) managing the path selection, the path predicate computation and the interaction with external solvers. This engine supports the two DSE algorithms highlighted above but also the different formula optimizations described hereafter. Last, Idasec is an IDA plugin allowing to lift and process data computed by the DSE engine, for post-analysis data exploitation. It provides many features among which it allows to highlight a dynamic trace in the Control Flow Graph (CFG) or to trigger an analyse in Binsec/se directly from Idasec.

Optimizations For a path π, various optimizations can be performed when computing the path predicate φ_π. We proposed two kinds of optimizations. First, basic pre-processing optimizations aiming at providing more uniformity performances across solvers. Second, we designed optimizations aiming at simplifying array manipulations in formula, which tends to be the most costly operations of solving. Among standard pre-processing algorithms we can denote constant propagation or backward pruning which remove all unused terms for the predicate to solve. Then, the rebase optimization aims at limiting the number of terms created by reusing older term definitions (see Section 7.1.3).

Array optimizations are used to significantly improve the solving time. The store-map chain Read-over-Write optimization (STMC-RoW) is a variant of the standard Read-over-Write. On arrays, the standard read-over-write aims at replacing a logical `select` in an array by its previous stored value (if any). Our optimization enjoys a very good complexity on arrays with constant indexes ($\mathcal{O}(log(n))$) and performs in the worst case as the standard RoW ($\mathcal{O}(n)$) for one operation. Besides that, the *memory flattening* allows to completely remove the array theory when the STMC-RoW managed to remove all `store` operations.

1.3.3 Contribution #3: Analysis combinations

Trying to get the best of the two algorithms developed (cf. **Contribution #1**) we studied different combinations approaches, like static/dynamic, forward/backward, over-approximated/under-approximated applied to the deobfuscation but also on vulnerability discovery and software testing purposes. The idea suggested by these combinations is to take advantage of the different approaches, considering that dealing with obfuscation is a multiple facets work and requires to deal with its different aspects. We formalize and develop three combinations aiming at solving three very different problematics:

- The *sparse disassembly* is a combination targeting obfuscation whose main goal is to improve a dynamic disassembly algorithm with a static disassembly controled by symbolic execution enforcing not to disassemble obfuscated dead code or obfuscated function `call`/`ret`. The first entity of the combination is a static/dynamic recursive disassembly algorithm. The dynamic aspect helps getting a sound disassembly for indirect or calculated jumps which is not the case for static disassemby. This algorithm is then intertwined with a static recursive disassembly and a BB-DSE algorithm to drive the disassembly not to disassemble spurious code and to disassemble some hidden control flow edges. Experiments shows that this combination performs better than IDA which over-disassemble various bytes.

- The Use-After-Free (UaF) vulnerability finding in legitimate programs. This work was performed in collaboration with Verimag and especially Josselin Feist that drove most of the design and experiments. This analysis combines a static abstract interpretation analysis aiming at finding relevant code functionalities (ie. `malloc`, `free`, `use`) in order to extract one or more *slices* of the program. Then a DSE algorithm path coverage algorithm is triggered on the *slice* to find possible vulnerabilities. The strength of the combination lays in the drastic reduction of possible path to cover by the DSE thanks to the slice computed by static analysis. The combination allowed to find few previously unknown vulnerabilities and especially in `JasPer` with CVE-2015-5221 [Mit15].

- The third combination [Bar+15a] slightly moves appart from binary-level security and finds an application in source-level software testing. To test a software, we usually check its compliance to some requirements, represented by properties to check at some program locations (coverage criteria). These systematic requirements might be unverifiable due to the program structure, which tends to slow down the testing

process. By combining abstract interpretation and weakest-precondition calculus, the analysis manages to detect most of these unsatisfiable tests requirements in order to remove them and speed-up the testing process. This combination named VA $\oplus$ WP was implemented in the C analysis platform Frama-C [Cuo+12]. On the set of benchmarks assessed, the combination managed to detect 98% of infeasible requirements. It significantly improves the accuracy of coverage measurements.

1.3.4 Contribution #4: Experimental Validation

The last contribution is the throughout study of two obfuscation techniques: opaque predicates and call stack tampering in a real-world context. We propose two detection methods based on the previously described techniques (cf.**Ctb#1**). We first, define and characterize the different kinds of possible opaque predicates and call stack tampering and explain how to address them with BB-DSE. We then perform two case-studies to validate the detection algorithms:

First, we perform a study of existing packers. This allowed to validate experimentally the efficiency of our analyses on real obfuscated binaries. It also gives a better understanding of the packers inner working and emphasizes the different tricks used in theirs codes (see Section 10.1).

Second, we perform an in-depth analysis of the X-TUNNEL ciphering proxy used by the Sednit/APT28 group as part of their cyber-espionage attacks to exfiltrate data [CCD16]. Later versions of this program are heavily obfuscated with what appeared to be opaque predicates. The analysis formalized and developped in this thesis managed not only to find the obfuscation within the code but also to extract a reduced CFG for every function of X-TUNNEL thus greatly simplifying a subsequent reversing of the program.

Conclusion

All these contributions improve the main underlying goal of this work which is to empower the reverse-engineering process by giving the analyst semantic information about the program and especially obfuscation. Subsequently, all the cards are placed in his hands for a better and deeper understanding of the binary being analyzed, with the assistance of different tools provided.

Publications This thesis led to the writing of 5 research papers, among which 4 are already published [Bar+15b; Dav+16a; Dav+16b; Fei+16] and one is still under review [DBM16]. This work was presented during several talks and seminars, including BlackHat Europe 2016 [DB16a].

1.4 Thesis outline

The manuscript is split in five parts. Part I provides some background about obfuscation and deobfuscation along with the basics about Dynamic Symbolic Execution. Chapter 2 lays the basic definitions and draws the global landscape of existing obfuscation and deobfuscation techniques used prior to this thesis. Chapter 3 provides all the necessary notations and definitions but also basic reminders about the dynamic symbolic execution inner working. People familiar with these notions can directly jump to the second part. Part II formalizes and describes the two DSE variants aiming at detecting obfuscation and reflect **Contribution #1**. Chapter 4 formalizes all the contributions brought to the path predicate computation in the DSE algorithm. In the same vein Chapter 5 formalizes and describes the Backward-Bounded DSE algorithm allowing to tune the DSE by performing it in the reverse direction. Part III discusses both the implementation performed as part of this thesis (Chapter 6) and the optimizations developped in the symbolic engine (Chapter 7). This part develops **Contribution #2**. Then, Part IV puts all these algorithms in action in order to validate the benefits they provide. This part describes and puts in perspective the last two contributions (**#3, #4**). Chapter 8 shows how they are used to deobfuscate opaque predicates and call stack tampering. Chapter 9 explores the way they can be combined with other existing analyses in order for instance, to leverage the obfuscations data into a disassembly algorithm. As an application, Chapter 10 presents three use-cases taking advantage of all the techniques and analysis defined beforehand. Finally, Part V discuss and puts in perspectives all the other potential applications of this thesis.

Chapter 2

Background

2.1 Obfuscation

2.1.1 Introduction

Over the years, obfuscation has taken a significant place in the software protection field. This term generally embraces any mean aiming at slowing down the analysis of a program for either an analyst or an automated analysis. As such, it has become quite popular for intellectual property protection, like the video-game industry via Digital Right Management (DRM) systems [Den16; MG14]; also it gained popularity in the malware development ecosystem. The only property that should be preserved by obfuscation is the semantic of the program i.e. its behavior. While some obfuscation techniques simply alter the syntactic form of a program, for example by twisting its CFG or adding spurious instructions, obfuscating the real behavior of the program itself is far more complex since the semantic of the program should be preserved after obfuscation. Even though watermarking can be considered as an obfuscation which hides some properties int the program, we focus in this thesis on obfuscation employed in malwares.

2.1.2 Definitions

Let's consider a program P. Barak et al. [Bar+12] defined an obfuscator $\mathcal{O}$, as a probabilistic compiler transformation of P in $\mathcal{O}(P)$ which satisfies the three following properties:

- **Functionality** $\mathcal{O}(P)$ computes the same function as P (provide the same observable behavior),
- **Polynomial slow-down** For all P, $\mathcal{O}(P)$ execution time is at most polynomially slower than P execution time or polynomially bigger than P size,
- **Virtual black-box** Everything efficiently computable with $\mathcal{O}(P)$ can also be computable with only an *oracle access* to P.

Under this, definition obfuscation is not possible in all cases in a generic way. However, obfuscation exists in practice and we choose a more pragmatic approach. Collberg provides

another definition more practical [CN09]. He first defines an asset as a set of properties on a program P and some inputs $\mathcal{I}$ defined by $asset(P, \mathcal{I})$. Then a metric m that characterizes the difficulty to compute the asset is defined by $m(P, asset(.)) \in [0, 1]$. $\mathcal{O}(P)$ is an obfuscation transformation if:

$$m(\mathcal{O}(P), asset(.)) > m(P, asset(.))$$

In addition to the semantic preserving property, Collberg's work [CN09] characterizes an obfuscation according to three other characteristics:

- **effectivness**. reversing the obfuscation process requires more resources than creating it,
- **potency**. does not weaken other obfuscation,
- **stealth**. splitted in two: *local stealth* an adversary cannot find where the transformation was applied and *steganographic stealth* an adversary can not find out whether a transformation has taken place or not,
- **cost**. overhead in space or time implied by the transformation.

More recently, some works proposed a theoric definition of obfuscation based on indistinguishability [Gar+13; SW13]. These definitions are very promising in practice but suffer for now of performances issues.

2.1.3 Taxonomy of Obfuscating Transformations

Far from being exhaustive, this section gives a little insight of the different kinds of existing obfuscation along with various examples.

Control & Data The main distinguishing feature of an obfuscation is whether it targets the CFG whether it targets its data (variables constants, strings, data structures ..).

Common obfuscation aiming at disrupting the CFG intents to make edges implicit or relying on values known at runtime so that a static analyzer will hardly infer them. Among existing techniques we can note opaque predicates and call/stack tampering discussed respectively in Sections 8.1 and 8.2. There is also *code diffusion* [Tra14] which intends to maximize some gadgets sharing between functions to blur the distinction between them. Signals, and exceptions uses callback handlers to jump at any location of the program without clearly visible edge. Listing 2.1, taken from a dumb virus `Win32.Eva` uses an invalid address exception to jump at a given location (example on Figure 2.2 label ①). At a stronger level, Virtual-Machine obfuscation completely recreates a new CFG by using an opcode based langage as interpreter of the original program [Rol09].

```
xor   edx, edx       // edx set to 0
push  ptr fs:[edx]   // Save current SEH handler (always at fs+0x0 in the TEB)
mov   fs:[edx], esp  // Replace the current handler
inc   ptr [edx]      // Trigger invalid address (try to read at address 0)
```

Listing 2.1: `Win32.Eva` exception triggered

As a counterpart to control obfuscation, data obfuscation hides the valuable assets of the program potentially giving hints to the reverser such as cryptographic constant values, imported functions and program strings. A common obfuscation is to cipher strings or sensitive information in the program postponing the deciphering at runtime. Another technique consists in altering data-structures by spliting them, merging scalar values or scrumbling them, using Mixed-Boolean Arithmetic (MBA) [Zho+07]. Finally, let us describe an obfuscation used in some malwares to hide library functions used. It first loads the shared library dynamically, then, given a custom hash function h and the hash value of the function to load H_f, the obfuscation will iterate all the library functions and apply the hash function on each function name. If a name s matches the hash values such as $h(s) = H_f$ then the obfuscator found the right function. If such obfuscation is used, names are never statically present in the code but only a hash values artifacts. Finally, self-modification is potent enough to hide both resources and the CFG of the program.

Static & Dynamic Another important characteristic of an obfuscation is to know whether it targets the static code and thus the static analysis or whether it targets the dynamic analysis by triggering the obfuscation at runtime. Static analysis, is highly dependent on the accuracy of the disassembly. Unfortunately, it can easily be fooled by obfuscation techniques as the ones presented hereabove. The disassembly can basically be fooled at three levels [MM16]:

- **Code discovery.** Namely decoding instructions. Indeed, as there is no distinction between code and data, the disassembly can wrongly disassemble data or wrongly left undecoded some code. Malwares are taking advantage of this interleaving to break the decoding process. An example is shown on Figure 2.2 label ③, where due to a spurious data byte `0x2d` IDA wrongly disassembled the code sequence at address `0x40101A`. Likewise, code-overlapping [Bon+15] exploits the fact that instruction are not atomic and some bytes of an instruction decoded separately can take another semantic. Figure 2.1 shows an example taken from the `TELock` packer and discussed by Bonfante et al [Bon+15].

- **CFG construction.** Namely generating the CFG. The standard approach is to replace conditional jumps by indirect or computed jumps as shown in listing 2.2. It prevents static recovering of the jump target addresses.

- **CFG partitioning.** Namely recovering high-level structures like function bounds etc. These operations can be confused by CFG flattening and all algorithms designed to inline, duplicate, interleave functions. Code diffusion is one of them.

Obvsiouly, any self-modifiying code obfuscation would make any static analysis very difficult if not worthless.

```
cmp    edi, esi
mov    eax, @X
mov    ecx, @2
cmov   ecx, eax
jmp    ecx
```

Listing 2.2: Conditional jump obfuscation

Addresses 0x01006e	7a	7b	7c	7d	7e	7f	80	81
Bytes	fe	04	0b	eb	ff	c9	7f	e6
#1 jump at 0x01006e7a	inc [ebx+ecx]			jmp +1				
#2 jump at 0x01006e7e					dec ecx		jg 0x01006e68	

Figure 2.1: `TELock` code overlapping

Conversely, dynamic analysis sidesteps most static analysis issues as it only goes through executed code. Contrariwise, it requires to deal with anti-debug, anti-tampering and potentially one or more process or threads interacting together. In the same way, due to the dynamic aspect of self-modification, it might be tedious to debug and to locate dynamically generated code. To prevent reverse-engineering, some malicious software are using checksum techniques to prevent any tampering of the code or debugging (as usual breakpoints patch the location by `0xCC` to add an `INT 3` interruption). The listing 2.3 shows how to compute the CRC of a function chunk in order to check its integrity against a pre-computed `HASH` value. This example uses the GCC label-as-a-value feature allowing to manipulate label addresses as values in C. As an additional protection, most malwares use anti-VM, anti-debug heuristics not to run in virtual-machine, sandboxes or controlled environement. Figures 2.2 in (2) shows a manner to check if the program is being debugged without direct call to `isDebuggerPresent`. It uses `GetProcAddress` to retrieve the address of the function which is then called by `call ecx`. Conveniently, `IsDebuggerPresent` is then removed from imports. Finally, listings A.1, A.2 and A.3 given in annexes, shows respectively: (1) how to check if a certain key is present in the Windows registry, (2)if a given process is running and (3) if the program runs suspiciously slower than native performance. In the later case it could indicate a debugged or instrumented process. These tricks are commonly used to evade dynamic analysis. All these examples are very basics but many other advanced techniques are playing with the peculiarities of sandboxes and virtual machines to avoid detection.

```
1  void *begin, *end;
2
3  int test(int x, int nb, int dummy) {
4    if(dummy) {
5      begin = &&begin;
6      end = &&end;
7      return 0;
8    }
9    int r=1;
10   int i;
11   begin:
12     for (i=0;i<nb;i++)// area
13       r = r * x;// checksumed
14   end:
15   return r;
16 }
17
18 int hash_fun() {
19   const int HASH = 3801;
20   int sum = 0;
21   char* p=begin;
22   while (p<(char*) end) {
23     sum += (unsigned char)*p++;
```

```
24    }
25    return sum == HASH;
26  }
27
28  void main() {
29    test(0,0,1);
30    if(hash_fun())
31      printf("Hash OK !\n");
32    else
33      printf("Fail hash\n");
34  }
```

Listing 2.3: anti-tampering checksuming

Needless to say that some techniques like self-modification are potent enough to disrupt both static and dynamic analyses. Notably, self-modification engines that perform polymorphism only when functions are called (on-demand) are very difficult to analyse as the whole code is never available in memory at a given time t.

Protection against symbolic reasoning. There has been very little work on obfuscations trying to circumvent symbolic approaches. Mixed-Boolean Arithmetics are crafted to be hardly targetable by symbolic execution. Two other techniques are used to impede symbolic execution, first hashing functions (hardly reversible) [Sha+08a] and second linear unsolved conjectures [Wan+11]. The later takes advantage of the difficulty for the symbolic execution to reason about loops. Finally, some works proposed to shadow conditional code obfuscation (input dependent condition) in order to create "secure triggers" [Fut+06]. One famous application of secure triggers based on hashing function is the `Gauss` malware which payload has never been deciphered so far [Goo16a].

The obfuscation field is a constantly evolving field with more and more evolved obfuscation mechanisms, enumerating all of them is impossible. Among them, TLB Splitting [Tor15] uses some processor features to decipher on the fly the code at a location where catching the deciphered code is out of reach for classical debugger and dynamic approaches.

2.2 Deobfuscation

2.2.1 Introduction

The term *deobfuscation* groups all techniques aiming at evading protected described hereabove. It can by caracterized as a two step process: (1) detecting the obfuscation, (2) reverting or deleting it. This later step is potentially harder in practice if not impossible for some obfuscations. Once detected, it worths considering three actions:

- reverting the obfuscating transformation (usually impossible)
- gather relevant information in the obfuscated code
- simplifying the obfuscated code to facilitate later analyses

My thesis focuses on the last option, and tries to provide automated analyses aiming at clarifying the code in order to help the reverse-engineer. Furthermore, narrowing the spectrum of analyses to the ones addressed in this thesis, we solely focus on approaches

Table 2.1: Obfuscation target & Potency summary

	Target		Against		
	Control	Data	Static	Dynamic	Symbolic
Conditional Code/Secure Triggers	✓		✓	✓	✓
Mixed-Boolean Arithmetic	∼	✓	✓		✓
Opaque predicates	✓		✓		
Call/Stack tampering	✓		✓		
Code diffusion	✓		✓		
Signal/Exceptions	✓		✓	✓	
Virtual marchine (VM)	✓		✓	✓	
Polymorphism (ciphering, packing..)	✓	∼	✓	✓	
Resources ciphering		✓	✓	✓	
Checksuming (code, data)	✓	✓		✓	
anti-debug (VM, debug..)	✓			✓	
Code-overlapping	✓		✓		
Aggregation/Encoding (variables split, merge, reordering..)		✓	✓	✓	
Aggregation/Encoding (functions split, merge, reordering..)	✓		✓		
Code parallelization	✓			✓	
CFG flattening	✓		✓	✓	
jump encoding (indirect, calculated)	✓		✓	✓	
Variable aliasing	✓	✓	✓	✓	

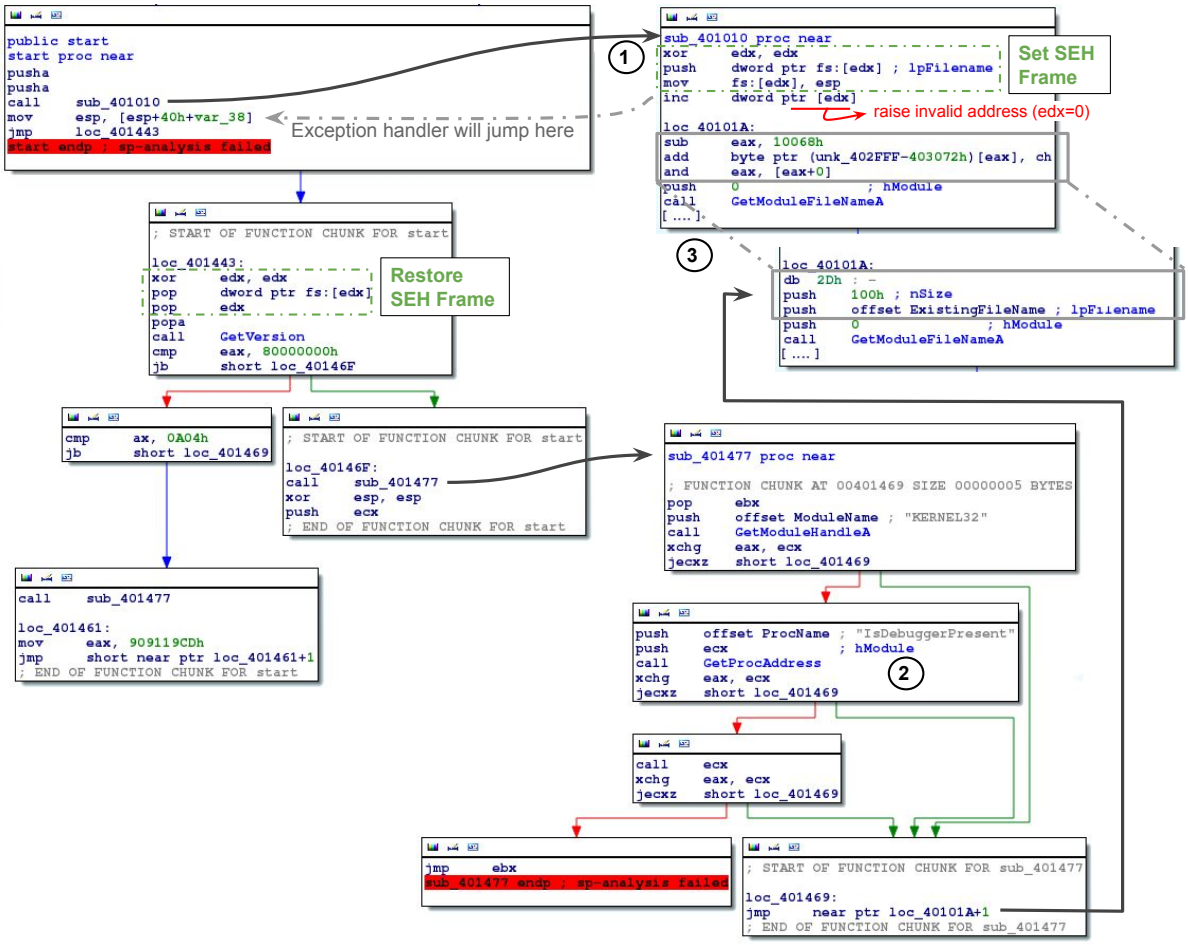


Figure 2.2: Win32.Eva Obfuscation tricks

explicitely targeting obfuscation (not simply disassembly matters) and using formal methods. Far from being exhaustive, this section gives an overview of existing deobfuscation methods available at the time of study.

2.2.2 Static approaches

Multiple data-flow analyses have been proposed to address obfuscation such as gen/kill, data dependency, alias analysis and abstraction interpretation [NNH99]. The last one can be thwarted by obfuscation such as CFG flattening, as the main loop structure will force abstract interpretation to merge its states leading to very unprecise states. To address this problem in virtualization-obfuscated binaries, Kinder proposed to enrich the abstract lattice with a Virtual Private Counter (VPC) to keep memory states splitted and recover precision [Kin12]. Also, using abstract interpretation, Preda et al, proposed to address opaque predicates with specific abstract domains modeling the attacker [Dal+06]. The idea was to refine the domain to get the best detection rate. Various researches are also carried to handle self-modification statically and especially metamorphism by abstract intepretation [PGD15].

Limits. Static approaches usually assume the CFG availability, no multi-threading and most of the time no self-modifying code. Although, polymorphism is a common practice which breaks many assumptions made by static analysis. We will discuss in later sections how our works tries to address these limitations.

2.2.3 Dynamic approaches

The main interest of dynamic analysis is to address self-modification. In this case the goal is to obtain the program after mutation. A generated piece of code can itself be self-modifying creating some nested layers of polymorphism. A program embedding one another is called *packer*. Various models of self-modification have been proposed named as waves [GMR09; Uga+15] or phases [Coo+09]. As an application, various research studies proposed automatic unpacker tools like Eureka [Sha+08b], Polyunpack [Roy+06] and Renovo [KPY07] which intend to be generic enough to work on unknown packers.

Another problematic, addressed by dynamic analysis, is cryptographic functions identification [CFM12]. While static approaches can be made using known constants, dynamic analyses can, for instance, compute the entropy of a function output. A regular distribution of output values could indicate some kind of cryptographic functions.

Limits. The main weakpoint of dynamic analysis is code coverage. Unless, doing fuzzing to cover as many paths as possible, the analysis is used to be performed on a single execution trace, which might not be reproducible. Exception is made by the dynamic tracing tool `panda` that allow full reproducibility of execution traces. Additionally, the analysis has to deal with all the anti-debugging and anti-tampering tricks used by malware which tends to be an endless cat and mouse game. Eventually, it is sometime simply infeasible to execute the software due to hardware constraints or environment constraints (connecting to a remote host etc). An example is given in Section 10.2.

2.2.4 Symbolic approaches

While existing approaches are essentially divided into static methods and dynamic methods, *symbolic approaches* offer a valuable balance between both. As a semantic approach, Lakhotia has published a technique named *binjuice* which aims at extracting a semantic summary of code chunks. Any useless instructions from a semantic point of view would be ignored [LPG13]. More in the scope of this thesis, Brumley in 2007 proposed techniques based on DSE to automatically analyse malicious binaries [Bru+07]. Very recently Ming [Min+15] proposed an approach to deobfuscate opaque predicates with DSE. That technique suffers some scalability issues. Lately, Yadegary explored ways to deobfuscate various CFG obfuscations, ROP-based obfuscation and packing using the combination of a bit-level taint and a generic DSE analysis [Yad+15; YD15]. This approaches also suffered some scalability issues due to the size of program analyzed.

Limits. Main limitations of existing symbolic execution approaches is the scalability. Indeed, underlying solvers hardly scale to big programs generating numerous constraints. As a consequence, it is thus essential to lighten the computation load of solvers in order to make symbolic execution based deobfuscation.

Chapter 3

Dynamic Symbolic Execution

3.1 Introduction

Symbolic Execution (SE) is a popular and fruitful formal approach to automatic (code-based) software testing. Given a path in a program, the key insight of Symbolic Execution (SE) is the possibility in many cases to compute a formula (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.

Basis were laid in the 70's by King [Kin76], but the technique found a renewal of interest in the mid 2000's [CS13] when it was mixed with concrete execution [SMA05; Wil+05] and combined with the growing efficiency of Satisfiability Modulo Theories (SMT) solvers. SE has quickly become the most promising technique for code-based automatic test generation, and has led to impressive case studies [Avg+14; Cad+11] and a promise of industrial adoption at large scale [GLM12]. Its usage for security purposes has also been considered, thanks to its straightforward adaptation to binary-level analysis [BH11; Son+08]. SE has successfully been applied in a wide range of security applications such as vulnerability discovery [Avg+11; HG12] and malware analysis [Bru+07], the topic of this thesis.

DSE Dynamic Symbolic Execution [SMA05; Wil+05] also known as *concolic execution* takes advantage of a concrete execution path to perform the computation. It provides the following advantages:

- the sound execution of the program. The path is sure to be feasible in practice,
- the next instruction executed is always known. This property is particularly helpful at binary-level when dealing with indirect/computed jumps or instructions potentially raising exceptions,
- by design, all the loops are unrolled.

The main drawback is that the trace followed is an *under-approximation* of the program execution which is inputs dependent. At this point, unless specified otherwise, the term Symbolic Execution will be used for Dynamic Symbolic Execution.

Concolic engines. This study led to the development of BINSEC/SE [Dav+16b] a full featured DSE engine. Yet, several other initiative already exists at binary-level like Mayhem [Cha+12] built at the top of the Binary Analysis Platform [Bru+11]. We can also note S2E [CKC12] based on KLEE [CDE08], Triton [SS15], Angr [Sho+16] and the closed-source ConcoLynx [YD15].

3.2 Binary-level semantic: DBA

At binary level, to avoid dealing with all the instructions set of all the architectures analyses are usually developped on intermediate representation. An intermediate provides a concise set of instructions encoding the semantic of instructions. Thus, an analysis designed on such intermediate representation is meant to work regardless of the backend architecture `x86`, `ARM` etc. DBA for Dynamic Bitvector Automata [DB15] is defined as an intermediate representation of assembly instructions, working on a subset of instructions independents from the architecture. Similar languages provide such functionalities like BIL [Bru+11], VEX, REIL [DP09]. Advantages of DBA like most other IR is a side-effect free and bit-precise language. DBA has the advantage of providing higher-level instruction design for program analysis like $assert, assume$ or $malloc, free$. The shortcomings are the lack of floating-point instruction modeling, the lack of thread, exception and the lack of self-modification modeling in the language.

3.2.1 Expressions & Types

All scalar variables are represented as bitvector $bv \in Bv$, a list of bits for which the size is statically known. The memory Mem is represented as bitvector addressed with the operator @ which represents both read and write operations depending whether it is used in the left-hand side (lhs) of an assignement instruction or not.

Expressions represent bit-level operations performed on bitvectors. An expression $e \in \mathbb{E}xpr$ can be a bitvector value bv a variable $v \in Var$ typically registers or the composition of operations on these two, like unary and binary operations repespectively $\Diamond_u$ and $\Diamond_b$. At the top of that, two specific values $\bot$ and $\top$ defines respectively an undefined value and a non-deterministic value. $\bot$ is used to represent undefined behaviors like some flags on certain operations in `x86`. $\top$ represent a non-deterministic value. Table 3.1 gives the grammar of expressions.

3.2.2 Intermediate Langage

A program P, is defined by a map Δ from location to instructions: $\Delta : loc \to \mathbb{I}nstr$. The core language is composed of basic instructions like assignement, both static, conditional and dynamic jumps and an ending instruction. We denote Σ a concrete memory state. Σ is a complete function mapping each variable $v \in Var$ to a value $bv \in Bv$ and mapping variable Mem to an array such as: $Bv^{\{addr\_size\}} \mapsto Bv^{\{8\}}$. The operational semantic is defined such that an instruction updates the memory state and the current location accordingly.

Table 3.1: DBA Expressions grammar

bv	$\in$	Bv ($\{0,1\}^n$ with n size of the bitvector)
v	$\in$	Var
$\Diamond_u$	::=	$-$ | $\neg$
$\Diamond_b$	::=	$+$ | $-$ | $\times$ | $/_{u,s}$ | $\%_{u,s}$ | $\|\|$ | $\&\&$ | $\oplus$ | $\gg_{u,s}$
		| $\ll$ | :: | $<_{u,s}$ | $\leq_{u,s}$ | $=$ | $\neq$ | $>_{u,s}$ | $\geq_{u,s}$
e	::=	$e \Diamond_b e$ | $\Diamond_u e$ | $@[\ e\ ]$ | v | bv | $\bot$ | $\top$

At the top of the core language, DBA includes various annotations and utilitarian instructions, aiming at modeling specific behavior and used for analysis specific purposes. For instance, *malloc* and $free$ are used to model an allocation and a "free" in memory. Similarly, *assert* and *assume* can be used afterward depending on the analyses.

Assembly instructions are highly likely to be decoded as multiple instructions sharing the same address in the program and forming a DBA block denoted $\mathcal{B}$. Instructions within a block are ordered using an $id \in \mathbb{N}$ allowing to pinpoint each of them separately. Within a block, an instruction can jump to any instructions, but the language enforces jumps accross instructions to be performed on blocks entrypoint. Thus, any instruction in the program is addressed by $l \in loc$ such as $loc \in (Bv^{\{addr\_size\}} \times \mathbb{N})$. The complete grammar is given in table 3.2. A condition c is defined as an expression that evaluates to a bitvector of size 1 written $Bv^{\{1\}}$.

Table 3.2: DBA instructions

l	$\in$	loc, $loc \in (Bv^{\{addr\_size\}} \times \mathbb{N})$
lhs	::=	v | $v\{i,j\}$ | $@[\ e\ ]$
$inst$	::=	$lhs := e$
		| $goto\ e$ | $goto\ l$
		| $ite(c)?\ goto\ l\ ;\ goto\ l$
		| $assert\ c$ | $assume\ c$
		| $malloc\ e$ | $free\ e$
		| $stop$
stmt	::=	$< l, inst >$
program	::=	ε | $stmt$ | $stmt\ ;\ stmt$

Some other extensions have been added to DBA, allowing to segment memory in region along with some permissions [DB15]. -

3.2.3 Usages

As a starter, Listing 3.1 gives the DBA semantic decoding of the `imul eax, dword ptr [esi + 0x14], 7`, instruction in `x86`. As a first advantage, using such modeling allows a more simple syntactic processing. Indeed, it is easier to match for instance an instruction assigning a given register rather than having to deal with the entire instruction set of the architecture with all possible parameters.

In terms, of static analysis DBA is being used as an intermediate representation semantic for abstract interpretation, simulation or high-level predicate recovery [DBG16] within the BINSEC platform and we use it for symbolic execution in this thesis.

```
res32 := (@[(esi(32) + 0x14(32))] ×(s) 7(32))
temp64 := ((exts @[(esi(32) + 0x14(32))] 64) ×(s) (exts 7(32) 64))
OF := (temp64(64) ≠ (exts res32(32) 64))
SF := ⊥
ZF := ⊥
CF := OF(1)
eax := res32(32)
```

Listing 3.1: `imul eax, dword ptr [esi + 0x14], 7`

3.3 Definitions

3.3.1 Path predicate

Given a program P over a vector V of input variables taking values in a domain $D \triangleq D_1 \times \ldots \times D_m$, a test datum t for P is a valuation of V, i.e. $t \in D$. The execution of P over t, denoted $P(t)$, is a path (or run) $\pi \triangleq (l_1, \Sigma_1) \ldots (l_n, \Sigma_n)$, where the l_i denote control-locations (or simply locations) of P and the Σ_i denote the successive internal states of P (i.e valuation of all global and local variables as well as memory-allocated structures) before the execution of each l_i.

Definition. φ is a path predicate for π if for any input valuation $t \in D$, t satisfies φ iff $P(t)$ covers π.

Correctness φ is said to be *correct* if any solution covers the path π.

Completness φ is *complete* if any input covering the path π is a solution.

program	path predicate φ_1	concretization φ_2	symbolization φ_3
inputs: a, b		a = 5	
x := a × b	$x_1 = a \times b$	$x_1 = 5 \times b$	$x_1 = fresh$
x := x + 1	$\wedge\ x_2 = x_1 + 1$	$\wedge\ x_2 = x_1 + 1$	$\wedge\ x_2 = x_1 + 1$
//assert x > 10	$\wedge\ x_2 > 10$	$\wedge\ x_2 > 10$	$\wedge\ x_2 > 10$

Figure 3.1: Path predicate, concretization and symbolization

A path predicate is intuitively the logical conjunction of all branching conditions and assignments encountered along that path. Figure 3.1 presents a simple program path (two assignments and a branching condition `x > 10` taken to *true*) together with three possible path predicates for that path. It is straightforward to check that φ_1 is correct and complete (a valuation of `a` and `b` satisfies φ_1 iff their execution satisfies the assertion). φ_2 is correct but incomplete due to the additional constraint $a = 5$ added by the concretization and φ_3 is complete but incorrect because of the symbolization that removed the constraint $x_1 = a \times b$ ($fresh$ is a new unconstrained variable) .

3.3.2 Symbolic Execution

Let's consider the set of finite[1] paths of a program P denoted Π^P. A symbolic algorithm builds iteratively a set of tests by exploring all the feasible paths. The three major components of the SE algorithm are the following:

- $\mathbb{S}el$, a path selection strategy, aiming at choosing the most appropriate path to explore for the intented analysis. This function is defined by: $\mathbb{S}el : \Pi^P \to \pi$
- $\mathbb{C}$, the path predicate computation function. It computes the predicate of the path in some predefined theories denoted by T. In our case in the `bitvector` and `array` (see. Section 6.4). This function is defined by: $\mathbb{C} : \pi \to \varphi$
- $\mathbb{S}ol$, the satisfiability checking using an automatic solver. This function takes a formula $\phi \in \Phi$ with $\Phi \subset T$ and returns either SAT with a solution t (i.e a test input) or UNSAT without solution. Formally we note $\mathbb{S}ol_T : \Phi \to \begin{cases} (\text{SAT}, t) \\ (\text{UNSAT}, \emptyset) \end{cases}$ the satisfiability function that checks a formula in the given theory T. Note that most SE tools rely on Off-The-Shelf (OTS) solvers (see. Section 6.4).

Therefore, we can define the symbolic execution algorithm in its entirety.

Definition For a program P and a set of finite paths Π^P, the symbolic execution algorithm denoted $\mathcal{F}_{DSE}$ is defined by:

$$\mathcal{F}_{DSE} : (\mathbb{S}el \circ \mathbb{C} \circ \mathbb{S}ol)(\Pi^P)$$

Algorithm 1 shows how the Symbolic Execution algorithm works with its three components, the path selection function (line 4), the path predicate computation (line 3) and the solving function (line 6).

Dynamic Symbolic Execution enhances SE by interleaving concrete and symbolic execution. The dynamically collected information can help the three main DSE operations $\mathbb{S}el$, $\mathbb{C}$ and $\mathbb{S}ol$ by adjusting and choosing the relevant approximations (cf. Figure 3.1 and Chapter 4). Its direct impact is on the sets of paths Π^P that is created by concrete

[1] enforced by bounding the depth of the path

Algorithm 1: Symbolic Execution algorithm

Input: a program P with finite set of paths $\Pi(P)$
Output: TS a set of pairs (t, π) such that $P(t)$ covers π
$TS := \emptyset$;
$S_{paths} := \Pi(P)$;
while $S_{paths} \neq \emptyset$ **do**
  $\pi := \mathbb{S}el(S_{paths})$; $S_{paths} := S_{paths} \backslash \{\pi\}$;
  $\varphi_\pi := \mathbb{C}(\pi)$;
  switch $\mathbb{S}ol(\varphi_\pi)$ **do**
    case *sat(t):* **do** $TS := TS \cup \{(t, \pi)\}$;
    case *unsat:* **do** skip ;
  end
end
return TS;

execution of the program. The underlying property that appears is that the path predicate is meant to be satisfiable. Otherwisely said $\forall\ \pi \in \Pi^P, \varphi \vDash \text{SAT}$, or more formally:

$$\forall \pi \in \Pi^P, \exists\ t \in D\ \textit{such that}\ (\mathbb{C} \circ \mathbb{S}ol)(\pi) = SAT, t$$

3.4 Path predicate computation

Intuitively, the path predicate computation is similar to concrete execution, except that the computation is performed on logical formulas rather than on concrete values. We denote by Σ^* the symbolic memory state which maps all variables $v \in Var$ to symbolic values φ (logical terms on logical variables ranging over Bv) and the distinguished variable Mem to a logical array from addresses to bytes. The path predicate φ is a first-order logic formula over logical variables and logical arrays. At a given point of execution, the internal state of the algorithm is composed of l, Σ^*, φ, respectively a location, a symbolic memory state and the current path predicate. The algorithm starts from the initial location l_0, *the initial state* Σ_0^* *associating a fresh logical variable to each program variable and a fresh logical array to* Mem, and $\varphi_0 \triangleq$ true. It then proceeds instruction by instruction along the execution. At the end the computed path predicate φ is returned. These notations are summarized in table 3.3.

Recall that the execution trace being fixed, the successor of each branching instruction is known. The operational semantic of the path predicate computation algorithm over DBA is given in Figure 3.2 where $\rightsquigarrow$ represents path predicate computation itself while $\vdash_e$ represents the symbolic evaluation of a DBA expression.

For instance, rule *var* $\frac{}{\Sigma^*, v \vdash_e \Sigma^*(v)}$ states that the symbolic evaluation of variable v is the symbolic value stored for v in the current symbolic memory state Σ^*, denoted $\Sigma^*(v)$. Rule $l - goto\ e$ $\frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \varphi \wedge to\_addr(l_e) = \varphi_e}{l, \Sigma^*, \varphi, goto\ e \rightsquigarrow l_e, \Sigma^*, \varphi', \Delta(l_e)}$ allows the symbolic evaluation of a dynamic jump branching to location l_e. The rule reads as follows: expression e is symbolically evaluated into the symbolic value φ_e and location l_e which is converted to a concrete

Table 3.3: Notation : forward symbolic execution

Symbol	Meaning
Φ	Set of all first-order logic formulas
Σ	The concrete state of variables and memory
Σ^*	The state of logical variables and memory
Δ	Mapping of location to instructions ($loc \rightarrow \mathbb{I}nstr$)
φ	Logical formula defining the path taken
Ω	The execution environment defined by : l, Σ^*, φ

address with $to\_addr : loc \rightarrow Bv$. Thus, the constraint $to\_addr(l_e) = \varphi_e$ is added to the path predicate modeling the fact that the execution flow must jump to l_e to ensure soundness. Remaining unexplained symbols are:

- the symbols gathered into $\Diamond_u^*$ and $\Diamond_b^*$ are the logical counterparts of the unary and binary symbols of concrete expressions, e.g "+" is evaluated to the logical operator `bvadd` of the bitvector theory;
- select/store are the standard logical operators from the theory of arrays, representing the readings and writings to specific array indexes;
- $fresh$ designates a new logical variable in the formula.

Property 1. *The path predicate computation algorithm of Figure 3.2 is correct and complete, i.e. it returns a correct and complete path predicate.*

This concludes the chapter recalling background notations and definitions about dynamic symbolic execution. This lays the basis for the algorithms improvements formalized in the next Part (II).

$$\mathbb{E}\textbf{xpr}: \quad cst\ \frac{}{\Sigma^*, bv \vdash_e bv} \qquad var\ \frac{}{\Sigma^*, v \vdash_e \Sigma^*(v)} \qquad binop\ \frac{\Sigma^*, e_1 \vdash_e \varphi_1 \quad \Sigma^*, e_2 \vdash_e \varphi_2 \quad \varphi \triangleq \varphi_1 \diamond_b^* \varphi_2}{\Sigma^*, e_1 \diamond_b e_2 \vdash_e \varphi}$$

$$unaryop\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \diamond_u^* \varphi_e}{\Sigma^*, \diamond_u e \vdash_e \varphi'} \qquad @\ \frac{\Sigma^*, e \vdash_e \varphi}{\Sigma^*, @\ e \vdash_e select(\Sigma^*(Mem), \varphi)}$$

$$\mathbb{I}\textbf{nstr}: \quad goto\ l_1\ \frac{}{l, \Sigma^*, \varphi, goto\ l_1 \rightsquigarrow l_1, \Sigma^*, \varphi, \Delta(l_1)} \qquad l_e - goto\ e\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \varphi \wedge to\_addr(l_e) = \varphi_e}{l, \Sigma^*, \varphi, goto\ e \rightsquigarrow l_e, \Sigma^*, \varphi', \Delta(l_e)}$$

$$T - ite\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \varphi \wedge \varphi_e = true}{l, \Sigma^*, \varphi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_1, \Sigma^*, \varphi', \Delta(l_1)} \qquad F - ite\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \varphi' \triangleq \varphi \wedge \varphi_e = false}{l, \Sigma^*, \varphi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_2, \Sigma^*, \varphi', \Delta(l_2)}$$

$$var\ assign\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \Sigma^*_{new} \triangleq \Sigma^*[v \leftarrow fresh] \quad \varphi' \triangleq \varphi \wedge fresh = \varphi_e}{l, \Sigma^*, \varphi, v := e \rightsquigarrow l+1, \Sigma^*_{new}, \varphi', \Delta(l+1)}$$

$$@\ assign\ \frac{\Sigma^*, e \vdash_e \varphi_e \quad \Sigma^*, e' \vdash_e \varphi_{e'} \quad \Sigma^*_{new} \triangleq \Sigma^*[Mem \leftarrow fresh_m] \quad \varphi' \triangleq \varphi \wedge fresh_m = store(\Sigma^*(Mem), \varphi_{e'}, \varphi_e)}{l, \Sigma^*, \varphi, @\ e' := e \rightsquigarrow l+1, \Sigma^*_{new}, \varphi', \Delta(l+1)}$$

$$stop\ \frac{}{l, \Sigma^*, \varphi, stop \rightsquigarrow \textbf{return}\ \varphi}$$

Figure 3.2: Path predicate computation.

Part II

Dynamic Symbolic Execution Extensions

Chapter 4

Concretization/Symbolization cost modulation

This chapter introduces a new DSE algorithm allowing to modulate in a flexible manner concretizations and symbolizations performed during the symbolic execution. This is made possible via the formalization of C/S policies integrated into a rule-based language CSml and implemented in Binsec/se. Benchmarks performed shows a negligible time overhead while proving the usefulness of such a flexible mechanism.

4.1 Introduction

Problem While a purely symbolic approach is worth considering, the strength of modern SE tools is to symbolically evaluate only a (small) trace fragment. *Concretization* uses run-time values in order to under-approximate the path predicate, while *symbolization* over-approximate the path predicate by introducing new logical variables. The former allows to handle in a precise but limited way parts of an execution which are either missing (e.g. system calls) or too costly to reason about (e.g. cryptographic functions), while the latter allows to generalize certain program steps, keeping the reasoning exhaustive but less precise.

Actually, choices of concretization and symbolization (C/S) are a crucial part of modern SE tools, together with path predicate computation and path selection. Yet, while the latter are either well-understood (path predicate) or under active research efforts (path selection), C/S policies have been much less studied. Especially, design choices behind implemented C/S policies are often barely explained, and most SE tools either propose only hard-coded C/S policies, or give full control on a line per line manner in the code [CKC12].

Goal and contributions We propose to address these problems through a clear separation of concerns between (1) a specification mechanism for C/S policies in SE, and (2) a new SE algorithm parametrized by an arbitrary C/S policy specification as sketched in Figure 4.1.

- We formalize what a C/S policy is and *we revisit the standard path predicate computation algorithm taking C/S policy into account* (Section 4.2.1). This is the first time such a parametric view of the core algorithm behind SE is provided. We clearly show where the C/S policy matters and we discuss correctness issues.
- We propose CSML, a *rule-based specification language for defining C/S policies*, together with its semantics (Section 4.3). The language is simple, yet powerful enough to encode standard C/S policies. Again, correctness issues are discussed.
- As a second application, these results have been implemented on top of the BINSEC framework [Dav+16b], yielding *the first SE tool with fully customizable C/S policies through high-level specifications* (Section 4.4). First experiments demonstrate that the *overhead induced by this genericity is very low* (Section 4.5).
- Finally, we present *the first quantitative comparison of C/S policies* (Section 4.5.1), focused on policies dedicated to the handling of memory operations. We compare five policies on 169 programs. We found that, while new policy PP* performs better on most examples, there is still a high variability of results between the policies depending on the considered example. This is a strong *a posteriori* argument for a generic C/S specification mechanism.

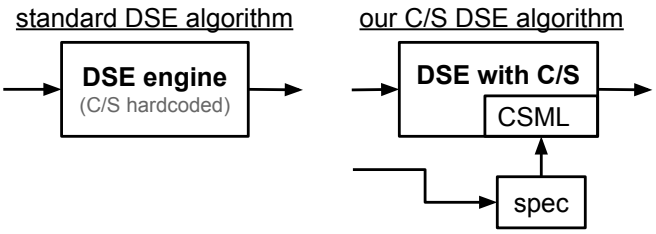


Figure 4.1: DSE engine with C/S overview

Outcome This work proposes a clear separation of concerns between the core SE algorithms and C/S specification, paving the way for flexible SE tools with easy to configure C/S policies. Additional benefits include: (1) better documented policies, facilitating their understanding, comparison and reuse; (2) the systematic study of concretization and symbolization (including both analytic and quantitative analysis) in order to better understand their impact and to identify interesting trade-offs; and finally (3) the fine-tuning of dedicated policies tailored to specific programs or needs.

State of the Art Most SE tools implement a single hard-coded built-in C/S policy, which can favor either *scalability* (i.e., by considering most values as concrete) or *completeness* (i.e., by keeping more symbolic values). For instance, the pioneering tools DART [GKS05] and CUTE [SMA05] fall in the former category (memory addresses, results from external library calls and part of non-linear expressions are concretized), while

PathCrawler keeps the computation fully symbolic [Wil+05] and EXE [Cad+08] stands in between. More recent engines can even build on more sophisticated heuristics in the hope of reaching a sweet spot between scalability and completeness, typically based on tainting [HG12; SAB10] or dataflow analysis [Cha+12]. We showed in Section 4.3.4 how such policies can be specified in CSml.

4.1.1 Definitions

Let's define the three decisions that can be taken on an expression during the path predicate computation.

Concretization ($\mathcal{C}$) uses run-time values in order to *under-approximate* the path predicate, allowing to handle in a precise but limited way parts of an execution which are either missing (e.g. system calls) or too costly to reason about (e.g. hash functions). For instance, concretizing read and write addresses in memory significantly reduces the complexity of the path predicate since the theory of arrays is computationally hard to solve

Symbolization ($\mathcal{S}$) *over-approximates* the path predicate by introducing a fresh logical variable, allowing to generalize certain program steps, keeping the reasoning exhaustive but less precise. For example, symbolizing `eax` after a system call is a good way to simulate all possible return values of the call. It can also simulate the non-deterministic values of certain instruction like `cpuid` or `rdtsc`

Propagation ($\mathcal{P}$) computes the path predicate as explained in Section 3.4, without any extra-approximation.

4.1.2 Path predicate correctness

While performing DSE, we might be driven into ignoring some functions not executing them symbolically or to ignore some instructions not decoded in the intermediate representation. While in practice, it helps to narrow the computation to the main code of interest, it might also yield incorrect path predicates. Rarely, it can be due to a bad encoding of the instruction semantic in the given IR.

Definition 1. *The decoherence or loss of correctness of the path predicate is the phenomenom by which logical values are being made incorrect w.r.t the program execution due to unhandled semantic side-effect of the original program execution.*

Figure 4.2 gives an exemple of decoherence that might occur when purposely not tracing a given function here `myfun`. If not handling properly registers or memory that might have been modified by symbolizing them, we can provoke such decoherence. In this example, the stack pointer `esp` has not properly been updated after the function return because the `stdcall` except the callee to clean-up its stack. Therefore, two kinds of decoherence may happened:

Addr	x86-stdcall	DBA	Path predicate φ_π	Path constraints
	...	...	inputs:$esp0, esi0$	
@0	`push esi`	`@[esp-4] := esi` `esp := esp - 4`	$@[esp0-4] = esi0$ $\wedge\ esp1 = esp0 - 4$	$esp0 - 4 = 0x42$

@1	`call myfun`	`esp := esp-4`
		`@[esp] := @2`
		`goto @myfun`
			$\wedge\ esp2 = esp1 - 4$
			$\wedge\ @[esp2] = @2$
				$esp2 = 0x3e$

[FUNCTION CODE EXECUTION SKIPPED]				
@2	`pop esi`	`esi := @[esp]` `esp := esp+4`	$\wedge\ esi1 = @[esp2]$ $\wedge\ esp3 = esp2 + 4$	$esp2 = 0x42$

Figure 4.2: Decoherence exemple by skipping function execution

- a **sat-decoherence.** is a decoherence breaking the execution soundness while keeping a satisfiable path predicate. In the example Figure 4.2, if considering a simple logical propagation ($\mathcal{P}$), it is expected that `pop esi` will get the value pushed on the stack by `push esi`. However, in the path predicate we have:

$$\begin{aligned} &@[esp0 - 4] = @[esp2] \\ &@[esp0 - 4] = @[esp1 - 4] \Leftrightarrow \\ &@[esp0 - 4] = @[esp0 - 8] \Leftrightarrow \\ &\Longrightarrow \text{SAT} \end{aligned}$$

  The load is performed at the wrong offset. But, as this offset is not constrainted, the path predicate is still SAT.

- an **unsat-decoherence.** is an excecution soundness loss which turn the path predicate to be unsatisfiable. In the example Figure 4.2, if we concretize the load and store addresses we obtain more path constraints (last column). When comparing the logical addresses of the load and store we have:

$$\begin{aligned} esp0 - 4 = 0x42 \ \wedge\ esp2 = 0x42 & \\ esp0 = 0x3e \ \wedge\ esp1 - 4 = 0x42 & \Leftrightarrow \\ esp0 = 0x3e \ \wedge\ esp1 = 0x3e & \Leftrightarrow \\ esp0 = 0x3e \ \wedge\ esp0 - 4 = 0x3e & \Leftrightarrow \\ esp0 = \mathbf{0x3e} \ \wedge\ esp0 = \mathbf{0x3a} & \Leftrightarrow \\ \Longrightarrow \text{UNSAT} & \end{aligned}$$

  The `esp0` cannot hold two different values in the same time, thus the path predicate is UNSAT. (we also have $esp2 = 0x3e \wedge esp2 = 0x42$ trivially UNSAT).

4.1.3 C/S policy introduction

Both concretization and symbolization allow to make SE more robust to real programs, through improving practical solvability. It can also tremendously reduce the decoherence phenomenom described in the previous section. Yet, they come at the price of losing either completeness (concretization) or correctness (symbolization). The decision upon which a value is concretized or symbolized is in general *hard-coded* inside the path predicate computation of existing tools. *Our goal, is precisely designing a flexible and clear specification mechanism for such decisions.*

Let's consider the instruction `x := @[a×b]` for which `a` and `b` had respectively values 7 and 3 at runtime. The logical formula associated with that instruction would be $x = select(M, a \times b)$ where M is a logical variable representing the memory. Intuitively, a C/S policy defines what actions are to be performed in terms of *concretization* and *symbolization* for that instruction. Assuming we want to concretize read operations in memory there is at least three ways to perform it:

- `[incorrect]` $x = select(M, 21)$ is incorrect as the property $a \times b = 21$ is lost
- `[minimal]` $x = select(M, 21) \wedge a \times b = 21$
- `[atomic]` $x = select(M, 21) \wedge a = 7 \wedge b = 3$

While `[minimal]` and `[atomic]` seem equivalent, `[atomic]` forces the values taken by a and b while `[minimal]` can still choose a and b as long as the product is 21. In addition `[minimal]` does not get rid of the $\times$ operator, which can come at cost for more complex operators or simply if the underlying theory used does not support it. None of these three policies is clearly the best, all depend on the context hence the need for a clear specification is required.

4.2 DSE algorithm with C/S modulation

4.2.1 C/S policy enriched DSE algorithm

As recall, the goal of a C/S policy is to decide whether a given expression should be propagated ($\mathcal{P}$) the normal behavior as in the standard algorithm (cf. Figure 3.2), concretized ($\mathcal{C}$) hence replace by its runtime value or whether it should be symbolized ($\mathcal{S}$) hence replaced by a $fresh$ (unconstrainted) symbol. We reuse here, all the notations used in the Chapter 3 about symbolic execution background.

We denote $\rho \triangleq \{\mathcal{C}, \mathcal{S}, \mathcal{P}\}$ the set of possible decisions and $\mathbb{S}tate$ the set of concrete memory states. We define a C/S policy *csp_expr* as a function that takes as input a location $l \in loc$, an instruction $i \in \mathbb{I}nstr$, a concrete memory state Σ and an expression $e \in \mathbb{E}xpr$, and returns a decision $d \in \rho$. Formally:

$$csp\_expr : loc \times \mathbb{I}nstr \times \mathbb{E}xpr \times \mathbb{S}tate \rightarrow \rho$$

Path predicate computation with C/S policy The C/S policy is queried inside the path predicate computation algorithm each time an expression must be evaluated. Intuitively, the standard algorithm given in Figure 3.2 is the case where the decision is always *propagate* ($\mathcal{P}$), starting from an initial state where all variables are symbolic.

A version of the path predicate computation algorithm revisited for taking C/S policies into account is presented in Figure 4.3. The main difference with the standard approach is that, the symbolic evaluation operator $\vdash_e$ is slightly modified to $\vdash_{cs\bullet}$ to use the policy. This operator now returns a symbolic expression together with a formula to be added to the current path predicate. Thus, before evaluating an expression, $csp\_expr(l, i, e, \Sigma)$ is queried in order to know which action shall be performed on it (lower part of Figure 4.3):

- $\mathcal{S}$: the expression e is replaced by a new symbol;
- $\mathcal{P}$: the expression e is logically evaluated, applying the C/S policy on each sub-terms [with $\vdash_{cs\circ}$];
- $\mathcal{C}$: the expression e is logically evaluated [with $\vdash_{cs\circ}$], then the resulting logical expression is constrained to be equal to the concrete evaluation $eval_\Sigma(e)$ of e, in order to preserve correctness of concretization.

Note that $\vdash_{cs\bullet}$ [and $\vdash_{cs\circ}$] return both a symbolic expression *and* a formula, the latter representing the constraints potentially induced by concretizing some subterms of the expression to evaluate.

The semantic defined in 4.3 yields various interesting properties:

Property 2. *An execution with the $\mathcal{P}$ policy is strictly equivalent to the standard DSE semantic depicted in Chapter 3.*

Property 3. *A policy using $\mathcal{P}$ and $\mathcal{C}$ keeps the correction but loses the completness.*

Property 4. *A policy using $\mathcal{P}$ and $\mathcal{S}$ keeps the completness but loses correction.*

4.3 CSML: C/S Specification Langage

The main goal is to design a high-level specification language for C/S policies able to encode major policies found in the literature (see. Section 4.3.4). Especially we want to distinguish between `[incorrect]`, `[minimal]` and `[atomic]`. Besides expressiveness, the following properties are also desirable: (1) a clear semantics; (2) simplicity and concision; (3) independence from the code under analysis or from a particular SE tool; (4) "executability", meaning we want to synthesize the code enforcing a C/S policy from its specification in order to use it in a SE tool.

We achieve these goals through CSML, a high-level *rule-based language* offering various functionalities like pattern matching or subterm checking on the instruction and expression being processed. That language can be integrated in the modified DSE algorithm formalized in the previous Section 4.2.1.

$$\mathbb{E}\textbf{xpr}: \quad cst\ \frac{}{\Sigma^*, bv \vdash_{cs^\circ} bv, true} \quad var\ \frac{}{\Sigma^*, v \vdash_{cs^\circ} \Sigma^*(v), true} \quad binop\ \frac{\Sigma^*, e_1 \vdash_{cs^\bullet} \phi_1, \varphi_1 \quad \Sigma^*, e_2 \vdash_{cs^\bullet} \phi_2, \varphi_2}{\Sigma^*, e_1 \diamond_b e_2 \vdash_{cs^\circ} \phi_1 \diamond_b^* \phi_2, \varphi_1 \wedge \varphi_2}$$

$$unaryop\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \phi' \triangleq \diamond_u^* \phi_e}{\Sigma^*, \diamond_u e \vdash_{cs^\circ} \phi', \varphi_e} \quad @\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \phi \triangleq select(\Sigma^*(Mem), \phi_e)}{\Sigma^*, @\ e \vdash_{cs^\circ} \phi, \varphi_e}$$

$$\mathbb{I}\textbf{nstr}: \quad goto\ l_1\ \frac{}{l, \Sigma^*, \varphi, goto\ l_1 \rightsquigarrow l_1, \Sigma^*, \varphi, \Delta(l_1)} \quad l_e - goto\ e\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \varphi' \triangleq (\varphi \wedge \varphi_e \wedge to\_addr(l_e) = \phi_e)}{l, \Sigma^*, \varphi, goto\ e \rightsquigarrow l_e, \Sigma^*, \varphi', \Delta(l_e)}$$

$$T - ite\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \varphi' \triangleq \varphi \wedge \varphi_e \wedge \phi_e}{l, \Sigma^*, \varphi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_1, \Sigma^*, \varphi', \Delta(l_1)} \quad F - ite\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \varphi' \triangleq \varphi \wedge \varphi_e \wedge \neg\phi_e}{l, \Sigma^*, \varphi, ite(e):\ l_1;\ l_2 \rightsquigarrow l_2, \Sigma^*, \varphi', \Delta(l_2)}$$

$$var\ assign\ \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi_e, \varphi_e \quad \Sigma^*_{new} \triangleq \Sigma^*[v \leftarrow fresh] \quad \varphi' \triangleq (\varphi \wedge \varphi_e \wedge fresh = \phi_e)}{l, \Sigma^*, \varphi, v := e \rightsquigarrow l+1, \Sigma^*_{new}, \varphi', \Delta(l+1)}$$

$$@\ assign \frac{\Sigma^*, e \vdash_{cs^\bullet} \phi, \varphi_e \quad \Sigma^*, e' \vdash_{cs^\bullet} \phi', \varphi_{e'} \quad m' \triangleq store(\Sigma^*(Mem), \phi', \phi) \quad \varphi_m \triangleq (\varphi \wedge \varphi_e \wedge \varphi_{e'} \wedge fresh_m = m')}{l, \Sigma^*, \varphi, @\ e' := e \rightsquigarrow l+1, \Sigma^*[Mem \leftarrow fresh_m], \varphi_m, \Delta(l+1)}$$

$$\Sigma^*, e \vdash_{cs^\bullet}: \left\{ \begin{array}{lll} fresh, true & \text{if} & \rho = \mathcal{S} \\ \phi_e, \varphi_e & \text{if} & \rho = \mathcal{P},\ \ \Sigma^*, e \vdash_{cs^\circ} \phi_e, \varphi_e \\ C_\phi, \varphi_e \wedge (C_\phi = \phi_e) & \text{if} & \rho = \mathcal{C},\ \ \Sigma^*, e \vdash_{cs^\circ} \phi_e, \varphi_e \text{ and } C_\phi \triangleq eval_\Sigma(e) \end{array} \right\} \rho \triangleq csp\_expr(l, i, e, \Sigma)$$

Instruction, location l and concrete state Σ are propagated inside all $\vdash_{cs^\bullet}$ rules, but we omit it for clarity.

Figure 4.3: Path predicate computation with C/S policy

4.3.1 Language specification

We now describe CSml, our rule-based language whose syntax is given in Table 4.1. It allows a scalable level of description from the generic high-level policy to a precise and highly targeted one.

Basic principles A rule is of the form $guard \Rightarrow \rho$ where the guard allows to check if the rule should be fired (typically using *pattern-matching* and *subterm checks*) and ρ is the decision to be returned by the policy. As explained before, rules are queried with the current location, instruction, expression and concrete memory state. Namely, guards are denoted by $\phi_{loc} :: \phi_{ins} :: \phi_{expr} :: \phi_{\Sigma}$, where:

- ϕ_{loc} is a predicate on the location,
- ϕ_{ins} is a predicate on the instruction,
- ϕ_{expr} is a predicate on the expression,
- ϕ_{Σ} is a predicate on the concrete memory state.

Rules are tested sequentially, and the first matching rule returns its associated action. If no rule gets triggered, then a default rule is applied. In each rule, guard predicates are also checked sequentially, so that ϕ_{loc} must be satisfied in order to check ϕ_{ins} and so on. Note that any of these predicates can be replaced by a wildcard $*$, which always evaluate to true. Finally, guard predicates may communicate in a limited way through *meta-variables* and *placeholders*.

Table 4.1: Policy language

policy	::=	rules default | default
rules	::=	rule | rule rules
rule	::=	guard $\Rightarrow \rho$
default	::=	$default \Rightarrow \rho$
guard	::=	$\phi_{loc} :: \phi_{ins} :: \phi_{expr} :: \phi_{\Sigma}$
ϕ_{loc}	::=	loc | [loc..loc] | *
ϕ_{ins}	::=	$\langle pinstr \rangle$ | *
ϕ_{expr}	::=	$\langle pexpr \rangle$ | $expr^+ \prec \text{term}^+$ | $expr^+ \preceq \text{term}^+$ | $*$
term^+	::=	$expr^+$ | $instr^+$
ϕ_{Σ}	::=	$P(expr)$

. $expr^+$: extended expression, allowing placeholders (!)
. *pexpr*: expr. pattern, allowing placeholders (!) and meta-variables (?)
. $instr^+$ and *pinstr*: the same w.r.t. instructions

Matching and meta-variables Predicates for ϕ_{expr} and ϕ_{ins} typically allow to check if the input expression (resp. instruction) matches a given pattern *pexpr* (resp. *pinstr*). A pattern is similar to an expression (resp. instruction), but with two additional kinds of variables.

Meta-variables (prefixed by "?") allow to match any term. Once successfully matched, these terms become available as placeholders in the subsequent guard predicates. When a meta-variable does not need to be reused, one can employ an *anonymous meta-variable* $?\star$.

Placeholders (prefixed by !) take the value of their corresponding meta-variable (e.g., $?e$ for $!e$) once matched. The distinguished placeholder $!_\square$ contains the current expression being processed. Given a pattern p, we denote the predicate "match p?" by $\langle p \rangle$. As an example, we concretize the expression being added to `esp`, in any assignment instruction:

$$* \;::\; \langle ?\star \;:=\; \texttt{esp} \;+\; ?e \,\rangle \;::\; \langle !e \,\rangle \;::\; * \;\Rightarrow\; \mathcal{C}$$

Here, $?e$ is defined in ϕ_{ins} and then available in ϕ_{expr} through $!e$. The rule reads as follows: if the current instruction assigns the sum of `esp` and some expression e (being captured by the meta-variable $?e$) to any lhs (meta-variable $?\star$), and the current expression matches e, (predicate $\langle !e \rangle$), then it should be concretized. Or, put another way: if we are evaluating an expression e in the context of an instruction where e is added to `esp` and assigned to some lhs, then e should be concretized.

Another example, slightly more complex is given in the table 4.2. This policy means: "if we are evaluating an expression e in the context of an assignment where e is used as the write address, then e is concretized, otherwise it is propagated."

Table 4.2: Concrete Store policy

$*$	::	$\langle @?e \;:=\; ?\star \rangle$	::	$\langle !e \rangle$	::	$*$	$\Rightarrow \mathcal{C}$;
default							$\Rightarrow \mathcal{P}$;

Subterm This language is already quite expressive, but still does not allow to match a nested sub-expression depending on its context. The typical usage is, when willing to concretize a read address, we want to know if the given expression is in the scope of a read operation. This can be solved by introducing the $\preceq$ operator (resp. $\prec$), allowing to check if an expression is a subterm (resp. strict subterm) of another one. The rule specifying that any read or write expression must be concretized is written:

$$* \;::\; \langle ?i \,\rangle \;::\; (@\; !_\square) \prec !i \;::\; * \;\Rightarrow\; \mathcal{C}$$

In this rule $?i$ matches a whole instruction.

The rule can be read as follows: when evaluating an expression e (given by the special placeholder $!_\square$) such that the expression $@e$ is a subterm of the current instruction (captured by $?i$) then e should be concretized. Note that $\prec$ and $\preceq$ can be applied to (extended) terms containing placeholders ($expr^+$ and $instr^+$ in Table 4.1).

Other predicates ϕ_{loc} consists in checking that the input location l is either equal to a given location loc or within a range of locations [loc..loc']. ϕ_Σ is a predicate over $\mathbb{E}xpr$ that can query information from Σ. For example, we can imagine performing some concretization and/or symbolization depending of the runtime value of the expression being evaluated. ϕ_Σ will be lightened in the next section with *extended memory states*.

4.3.2 CSml properties

CSml ensures interesting properties on the C/S policy being defined. In order to present them, we need a few additional definitions. A CSml rule is said to be *well-defined* if it is well-typed and if placeholders are used in an appropriate manner. Note that the well-definedness of a rule is automatically checkable. A C/S policy is *well-defined* if it is a total function (deterministic behavior, defined on any input), and it is *correct* if it leads to the computation of a correct path predicate (cf. Section 3.4). Then, by construction, we have the good following properties[2]:

Property 5. *A set of well-defined* CSml *rules defines a well-defined C/S policy.*

Property 6. *A set of well-defined* CSml *rules employing only $\mathcal{C}$ and $\mathcal{P}$ defines a correct C/S policy.*

4.3.3 Advanced Features

We propose here several useful extensions to the CSml core language.

Richer decisions We enrich the set of possible decisions by allowing a domain restriction on both $\mathcal{S}$ and $\mathcal{P}$, now denoted $\mathcal{S}_D$ and $\mathcal{P}_D$, where D is an interval constraint (resp. singleton constraint) of the form $[a..b]$ (resp. $[a]$) with a and b expressions evaluating to bitvector values. This feature is useful typically for limiting the domain of a fresh logical variable, but it can also be used to encode incorrect concretization or restricted propagation (cf. below). Note that the domain does not need to be defined by constant values, for example its definition can involve runtime evaluation of bitvector expressions, through function $eval_\Sigma : \mathbb{E}xpr \to Bv$

- incorrect concretization of r/w expressions `[incorrect]`

$$* :: \langle ?i \rangle :: (@\ !_\square) \prec !i :: * \Rightarrow \mathcal{S}_{[eval_\Sigma(!_\square)]}$$

- restriction of r/w expressions [BH11]

$$* :: \langle ?i \rangle :: (@\ !_\square) \prec !i :: * \Rightarrow \mathcal{P}_{[eval_\Sigma(!_\square)-10..eval_\Sigma(!_\square)+10]}$$

[2]Property 5 comes also from the sequential ordering of rules and the presence of a default rule.

Richer subterm constraints We allow chaining of subterm constraints, e.g. $e \prec pe_1 \prec \ldots \prec pe_n \prec \text{term}^+$, together with the use of anonymous meta-variables inside the pe_i. This allows finer subterm relationship, such as checking that an expression is a subterm of an expression pattern, used itself within another expression.

- recursive concretization of r/w expressions:

$$* :: \langle ?i \rangle :: !_\square \prec (@ ?\star) \prec !i :: * \Rightarrow \mathcal{C}$$

Compared with `[minimal]` concretization, this rule enforces the concretization of all subterms of a r/w expression. This policy is also slightly different from atomic concretization **CS3** which encoding is shown hereafter.

Richer predicates We can also allow more predicates in the language. This can be done at two stages: either enriching the four classes of predicates already defined, or adding new classes of predicates. For the first category, it could be useful to have a `var` predicate indicating if a term is a variable or not. An application is to restrict concretization to atomic variables:

- atomic concretization of r/w expressions `[atomic]`

$$* :: \langle ?i \rangle :: \texttt{var}(!_\square) \wedge !_\square \prec (@ ?\star) \prec !i :: * \Rightarrow \mathcal{C}$$

For the second category, it could be useful to consider a predicate class ϕ_{step} regarding the step of the execution, allowing for example to define a C/S policy step by step in a trace-oriented manner, which may sometimes come handy.

Extended memory states In the same vein, it can be interesting to enrich the predicate ϕ_Σ working on a concrete memory state Σ into a predicate ϕ_{Σ^+} working on an extended concrete memory state Σ^+. Basically, an extended concrete memory state is a concrete memory state enriched with additional runtime information collected. A typical example of such a predicate is $\widehat{\mathcal{T}}_{\Sigma^+}(e)$, indicating whether an expression e is tainted or not in a given extended memory state Σ^+, using dynamic taint information [SAB10].

Note that this extension makes the C/S policy dependent on the services provided by the underlying dynamic execution engine. While it is fair to assume that a concrete evaluation function $eval_\Sigma$ is available on any dynamic execution engine, more exotic queries on Σ^+ may not be available. We assume that C/S policies querying unsupported Σ^+-function (or Σ^+-predicate) are syntactically rejected.

C/S injection Besides C/S at the level of symbolic evaluation, another common pattern is to enforce concretization and/or symbolization through direct modification of the symbolic memory state. This is particularly useful to handle unknown or hard-to-reason-about functions (e.g. system calls, cryptographic function) with side-effects or returning complex data structures. Note that this kind of C/S is different from the one we have considered so far, since it modifies *permanently* the value of a *lhs* (inside Σ^+), while *csp_expr*

affects a *single evaluation* of *any expression*. For example, C/S injection allows to declare that at some location, variable `eax` receives a fresh value (which will last along the trace until `eax` is rewritten), while *csp_expr* allows to declare that at some location, variable `eax` evaluates as if it were unconstrained (with no impact on the remainder part of the trace). C/S injection can be handled similarly to C/S in expression evaluation. Due to space limitation, we only sketch the idea. Contrarily to *csp_expr*, concretizations defined by *csp_mem* are not ensured to be correct, as the symbolic memory state is modified without any additional (correctness) constraint. We thus denote it $\mathcal{S}c$. We introduce a new function:

$$csp\_inject : loc \times \mathbb{I}nstr \times \mathbb{S}tate \rightarrow (lhs \mapsto \{\mathcal{S}c, \mathcal{S}\})$$

which takes as argument a location, an instruction and a symbolic state, and returns a map from *lhs* to decisions, which are here limited to $\mathcal{C}$ and $\mathcal{S}$. Intuitively, the map represents the modifications which have to be performed on the current symbolic memory state Σ^* before the symbolic execution goes on.

Discussion Altogether, these extensions provide a very fine control over the C/S policy, allowing for example to encode the subtle differences between correct concretization, incorrect concretization, recursive concretization and atomic concretization.

4.3.4 Application:Encoding Standard C/S Policies

To illustrate how our language works, we show the encoding of several state-of-the-art policies from the literature, not yet covered in previous sections.

For instance, CUTE and DART [GKS05; SMA05] concretizes both read and write addresses, as well as part of non-linear operations (here: left operand of any $\times$ operator). The associated policy is shown in Table 4.3.

Table 4.3: CUTE/DART policy

$*$	::	$\langle ?i \rangle$	::	$(@\ !_\square) \prec !i$	::	$*$	$\Rightarrow \mathcal{C}$
$*$	::	$\langle ?i \rangle$	::	$(!_\square \times ?\star) \prec !i$	::	$*$	$\Rightarrow \mathcal{C}$
default							$\Rightarrow \mathcal{P}$

A variant for memory operations consists in concretizing also all non-tainted expressions [HG12]. The corresponding policy is shown in Table 4.4, where $\widehat{\mathcal{T}}_{\Sigma^+}(e)$ indicates whether an expression e is tainted or not in a given extended memory state Σ^+ (cf. Section 4.3.3).

Table 4.4: CUTE/DART policy with tainting

$*$	::	$\langle ?i \rangle$	::	$(@\ !_\square) \prec !i$	::	$*$	$\Rightarrow \mathcal{C}$
$*$	::	$*$	::	$*$	::	$\neg\widehat{\mathcal{T}}_{\Sigma^+}(!_\square)$	$\Rightarrow \mathcal{C}$
default							$\Rightarrow \mathcal{P}$

The approach followed in EXE [Cad+08] in case of multi-level dereferencement consists in concretizing all r/w expressions but the most nested one. The encoding of such a policy is shown in Table 4.5.

Table 4.5: EXE policy

$*$	::	$\langle ?i \rangle$	::	$(@\ !_\square) \prec (@\ ?\star) \prec !i$	::	$*$	$\Rightarrow$	$\mathcal{C}$
default							$\Rightarrow$	$\mathcal{P}$

It is important here to use $\prec$ rather than $\preceq$

Finally, the policy in Mayhem [Avg+16; Cha+12] consists in concretizing all write expressions while keeping read expressions symbolic as long as they cannot take too many values (otherwise, concretizing them). We need here to consider Σ^+ enriched with an interval analysis. The encoding is then given in Table 4.6, where $card_I(e)$ gathers the number of possible values for e from the interval information available in Σ^+.

Table 4.6: Mayhem policy

$*$	::	$\langle @?a\ := ?\star \rangle$	::	$\langle !a \rangle$	::	$*$	$\Rightarrow$	$\mathcal{C}$
$*$	::	$\langle ?i \rangle$	::	$(@!_\square) \prec !i$	::	$card_I(!_\square) < 1024$	$\Rightarrow$	$\mathcal{P}$
$*$	::	$\langle ?i \rangle$	::	$(@!_\square) \prec !i$	::	$*$	$\Rightarrow$	$\mathcal{C}$
default							$\Rightarrow$	$\mathcal{P}$

Summary Table 4.7 presents a summary of the kinds of C/S policies CSml can encode, together with the required extension of the language. It is remarkable that CSml can encode all popular C/S policies despite a limited language. Hence, we think that our rule-based language manage to capture the crucial aspects of current C/S policies.

Limits We do not know of any major existing C/S policy that cannot be encoded into CSml. Yet, the framework has some limitations, coming from both the ordered evaluation of guard predicates and the very restricted communication between those predicates. Here are two such limitations.

- A C/S policy does not depend on the symbolic state we are building, for example we cannot decide to concretize a term if all its leaves (variables) are already concretized.
- A C/S policy does not depend on the formula we are solving. For example, we cannot compute a path predicate, pass it to a solver (or to a simplifier) and then request concretization or symbolization depending on the solver's output.

Note, however, that the extended memory state Σ^+ does allow to overcome most of the above limitations, assuming one is willing to store (resp. to query) very complex information into (resp. from) Σ^+. In our view, Σ^+ should be used with care, only as a last resort.

Table 4.7: Encoding of C/S policies

policy	language
minimal concretization `[minimal]`	core language
recursive concretization	extended $\prec$
atomic concretization `[atomic]`	extended $\prec$ and `var`
incorrect concretization `[incorrect]`	extended decisions
r/w full-concrete [dart/cute]	core language
r/w full-symbolic [pathcrawler]	core language
r/w domain restriction [osmose]	extended decisions
r/w multi-level [exe]	extended $\prec$
r/w taint-based [HG12]	extended Σ^+
r/w dataflow-based [mayhem]	extended Σ^+

4.4 Implementation

The C/S policy mechanism presented so far has been integrated into Binsec/se [Dav+16b]. An overview of the modified architecture is shown in Figure 4.4. The C/S policy is specified in a textual format close to CSml. Subsequently, while the SE engine creates the path predicate, the C/S policy is queried for each encountered expression (Section 4.2.1) via a hook function, instantiated from the CSml specification.

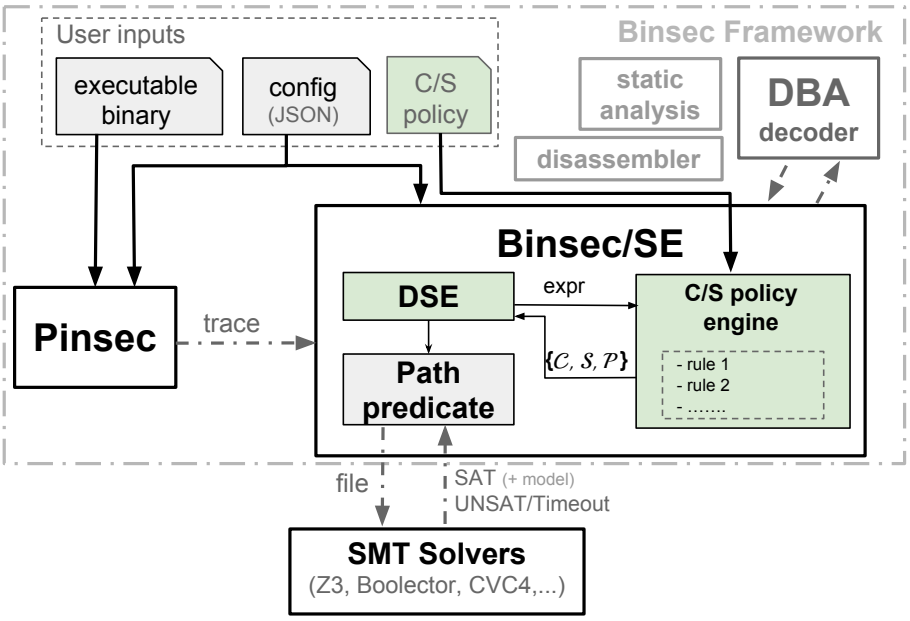


Figure 4.4: CSml support in Binsec/se

This version of Binsec/se is currently *the first SE tool supporting high-level specifica-*

tion of a wide range of C/S policies. The core engine is fully functional: all experiments of Section 4.5 have been carried out with it. The whole CSml core language is supported. Extensions still under development are extended memory states (taint and heap information), C/S injection, richer decisions, subterms and predicates.

4.5 Experiments

In this section three experiments are carried out with CSml and Binsec/se. The aim is to evaluate the performance and to compare the benefits brought by different policies. The three different evaluations are the following:

- First, we perform a large scale study on various benchmarks: some from samate [NIS16], some malwares and all the *coreutils*. The objective was to study the impact of different policies targeting memory reads and writes. This evaluation aims at answering the following two research questions:
  - **RQ 1**: Do C/S policies have a significant impact on SE in terms of the quality of results?
  - **RQ 2**: Is there a best C/S policy for read/write operations – among standard policies?
- Second, we study the overhead of rule-based C/S specification in comparison to hardcoded ones. This benchmark the following two research questions:
  - **RQ 3** What is the extra-cost of rule-based C/S specification ?
  - **RQ 4** Is it affordable, i.e is the extra-cost low w.r.t solving time ?
- Last, we are interested in studying the impact of C/S policies in terms of path predicate solving time. The goal is to study the solving time evolution with regard to the path length and the solver used. The research question addressed is:
  - **RQ 5** Do all solvers handle all policies equally ?

Policies. For this three experiments we are considering 5 different C/S policies handling diffferently memory reads and writes. Policies are: CC, CP, PC, PP*, PP, where the first letter indicates whether read addresses are concretized ($\mathcal{C}$) or propagated ($\mathcal{P}$). The second letter indicates the same for write addresses in memory. PP* is an original policy where all read/write addresses are kept logic but the base and stack pointer are concretized (i.e on `x86` registers `esp` and `ebp` are concretized). While CC and PP are standard policies the others are rather new. Reasoning logically about the memory is known to be difficult for DSE engines and SMT solvers. As a consequence these policies are chosen to assess and benchmark the policies on memory operations.

4.5.1 Quantitative Evaluation

This experiment is performed over 167 programs (x86 executable codes) – composed of programs from NIST/SAMATE [NIS16] (a standard benchmark for program analysis), all Unix *coreutils* and several Windows malware from VXHeaven [VXH16], for a total of 45,242 solver queries. Details can be found in Table 4.8. All instructions are traced, except calls to library functions which are stubbed by symbolic values ($fresh$ logical variables). The solver is `Z3`, with a time-out of 30 seconds.

Table 4.8: Benchmark characteristics

category	#prog	# trace instr		# trace branch		
		max	avg	max	avg	total
samate	50	5,000	2,772	1,177	333	16,554
coreutils	100	5,000	1,572	1,053	171	17,198
malware	17	5,000	3,739	1,539	675	11,490
total	167	5,000	2,151	1,539	270	45,242

We measure the influence of these policies in the following way: for each benchmark program, we consider an arbitrary (but reproducible) initial concrete execution and we ask the SE engine to iteratively invert every condition along the initial execution, leading to a set of new path predicate computations and solver queries. We record for each policy the number of queries which have been successfully solved (SAT), proved infeasible (UNSAT) or which have triggered a time-out (TO). Note that only the first category leads to new test input and (hopefully) better code coverage.

Results and conclusion Part of results are summarized in Tables 4.9 and 4.10. Firstly, [**RQ 1**] the choice of C/S policy may greatly affect the outcome of SE: there are $\geq$5x more SAT results on 20/167 examples, and up to 286x more SAT results on one program. Secondly, [**RQ 2**] there is no clear hierarchy between the considered policies. Indeed, even if PP* performs very well on many examples — PP* is the best policy on 41/167 examples, and it is optimal on 117/167 examples (Table 4.10), the global number of successfully solved instances is quite similar for any policies but PP (Table 4.9). Actually, while a more symbolic policy leads in theory to more satisfiable queries, it may also come at the price of harder-to-solve formulas and time-outs. These results are a strong argument in favor of a generic C/S mechanism.

The major *threats to validity* are the representativeness of the experimental setting (policies, programs) and internal bugs in the SE tool. We mitigate these threats through using standard policies and variants of them as well as a large program set coming from three distinct well-known and publicly-available benchmarks. Moreover, we rely on publicly-available tools (SE, solver) and results have been crosschecked for internal validity.

Table 4.9: Summary of #SAT, #UNSAT and #TO (167 programs and 45,242 queries, TO 30sec)

	#SAT	#UNSAT	#TO
CC	4,518	40,712	12
PC	4,436	38,897	1,909
CP	4,651	39,310	1,281
PP*	4,515	31,320	9,407
PP	3,340	25,037	16,865

total number of queries: 45,242

Table 4.10: Best and optimal policies

	samate		coreutils		malware		total	
	opt	best	opt	best	opt	best	opt	best
CC	20	0	44	1	5	0	69	1
PC	20	2	49	4	6	1	75	7
CP	23	1	61	11	4	0	88	12
PP*	36	12	71	24	10	5	117	41
PP	33	9	36	7	7	2	76	18

total number of programs: 167 - best (resp. opt): number of programs for which the considered policy returns the strictly highest (resp. highest) number of SAT answers, w.r.t. the other policies

4.5.2 Rule-Basd Language overhead

We evaluate the overhead of our parametric C/S policy mechanism. We want to answer the two following questions: **RQ 3:** What is the extra-cost of rule-based C/S specification, especially w.r.t. hard-coded policies (by mean of callback functions) and no C/S policy at all? **RQ 4:** Is it affordable, i.e. is the extra-cost low w.r.t. solving time?

Protocol We reuse the experimental setting of the previous evaluation. We consider two metrics: the cost of formula creation – which is directly affected by C/S policies, and the ratio between formula creation and formula solving. We record these metrics for the 5 previous policies, implemented either in CSml or through native callbacks, and we consider a baseline consisting of SE without any C/S policy.

Table 4.11: Overhead evaluation

		min	max	average
base	PP	0.04%	3%	0.3%
rule-based C/S policy	CC	0.1%	17%	1.2%
	CP	0.1%	23.5%	1.45%
	PC	0.08%	12.8%	0.85%
	PP*	0.08%	12.3%	0.95%
	PP	0.05%	4%	0.48%
hard-coded C/S policy	CC	0.05%	8.5%	0.5%
	CP	0.05%	8.2%	0.5%
	PC	0.05%	8%	0.45%
	PP*	0.05%	6%	0.45%
	PP	0.04%	3%	0.3%

Ratio between the cost of path predicate computation (impacted by C/S) and the whole cost (i.e. formula creation + formula solving). Note that the time for formula solving does not depend on the way C/S is implemented (rules, hard-coded, no C/S).

Results and conclusion Table 4.11 reports the ratio between formula creation and formula creation plus solving. Note that solving time does not depend on the way C/S is implemented. [**RQ 3**] CSml does lead to a more expensive path predicate computation (average: x3 w.r.t. hard-coded callbacks and up to x5 w.r.t. no C/S at all, at worst x7 on some examples), yet [**RQ 4**] the cost of predicate computation is still negligible (average of 1.45% for the most expensive C/S policy; maximum of 23% on some easy-to-solve path predicates) w.r.t. the cost of predicate solving. Hence, our rule-based C/S mechanism brings extra-flexibility at only a very slight extra-cost.

Threats to validity: besides issues discussed in the previous evaluation, the considered policies are rather simple w.r.t. the expressive power of CSml. While the study is of

interest because these policies are representative, further investigations are required for *complex* CSml policies.

4.5.3 Computation time benchmark

The first experiments have been carried out on a single program path which length is 377309 instructions long. On this path, we solved the path predicate for path length ranging from 1 to 377309 with the 5 different C/S policies. The goal was to evaluate the impact of the C/S policy on the solving time with regard to the path length. We expect to be able to classify which solver better handle each kind of C/S policies. Note that benchmarks were performed with the highest level of optimization (CRH) as described in Chapter 7.

By solver. The Figure 4.5, shows the results performed on the 5 policies and 5 different solvers using a 2 minutes timeout. As expected CC outperform the other policies as it constraints both read and write addresses in memory. Contrariwise PP barely overcome the 2550 path depth thanks to `boolector`. PP* provides a good alternative as it concretizes read/writes on the stack. The only solver to solve the whole path is `Yices` in CC followed by `mathsat` that succeed in solving a 81,960 instructions long path predicate.

By policies. The Figure 4.6, shows the results by policies and for each solver. As solvers deal differently with the different theories we wanted to check if some policies were better handled by a given solver. Results, speak by themselves while `Yices` surpass all other solvers for policies reduce memory (array) operation it performs very bad on PP policy. Contrarily, `boolector` is the worst for CC but outperform all the others solvers for CP,PC and PP. As an outcome, this experiment shows that the solver choice have a strong impact on the policy employed.

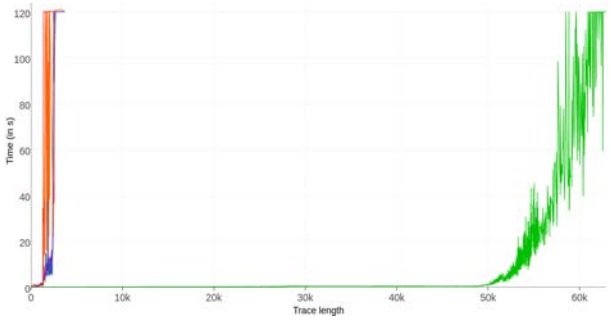
Table 4.12: Best solver summary

	CC	CP	PC	PP*	PP
best (#tr.len)	`Yices` 377,309	`boolector` 5,456	`boolector` 6,160	`boolector` 2,700	`boolector` 2,550
worst (#tr.len)	`boolector` 50,938	`mathsat` 174	`mathsat` 210	`mathsat` 987	`mathsat` 987

These two experiments allowed to highlight the solving time disparity between policies, strenghtening the approach of modular policies. It also highlighted a great disparity between solvers which, as we can notice, deal very differently with concrete/symbolic memory accesses. Furthermore, this highlight which solvers best fits each policy and gives a good sampling (truth table) of what solving time is to except depending on the policy and the solver. Table 4.12 gives the overall overview of which solver best and worst

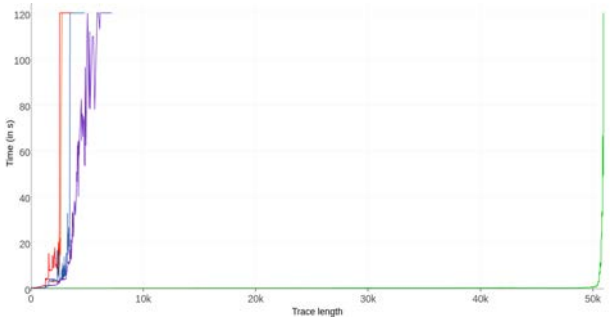
Z3	**CC**	**CP**	**PC**	**PP***	**PP**
500	0.02	0.35	0.34	0.03	0.92
1000	0.03	0.66	0.83	0.03	1.03
10,000	0.07	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	62540	2581	2553	2442	1353

(a) Z3 solving time: Table & Graph



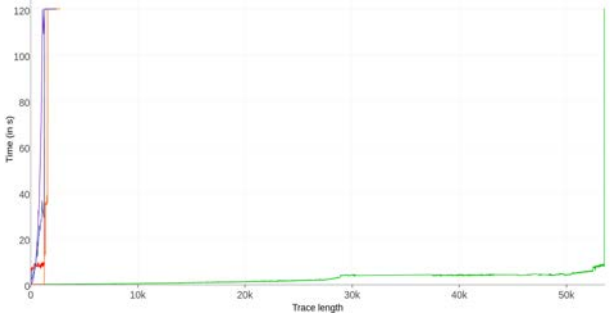
boolector	**CC**	**CP**	**PC**	**PP***	**PP**
500	0.02	0.2	0.1	0.08	0.46
1000	0.02	0.26	0.16	0.14	0.93
10,000	0.05	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	50938	3456	6160	2700	2550

(b) boolector solving time: Table & Graph



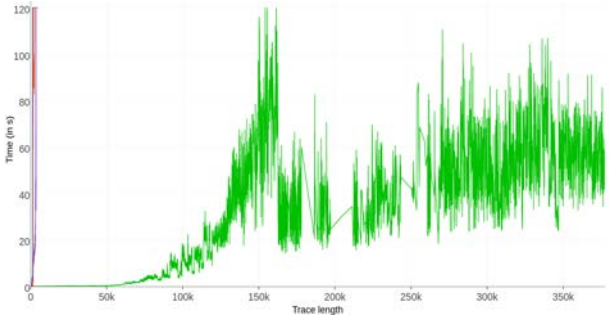
CVC4	**CC**	**CP**	**PC**	**PP***	**PP**
500	0.04	10.25	11.17	0.06	9.54
1000	0.04	30.4	74.5	0.11	7.42
10,000	0.44	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	53540	1260	1200	1575	1280

(c) CVC4 solving time: Table & Graph



Yices	**CC**	**CP**	**PC**	**PP***	**PP**
500	0.02	1.54	0.19	0.02	0.17
1000	0.02	3.76	0.75	0.03	0.2
10,000	0.06	X	X	X	X
100,000	10.7	X	X	X	X
377309	54.37	X	X	X	X
max	377309	1280	3492	2484	1525

(d) Yices solving time: Table & Graph



mathsat	**CC**	**CP**	**PC**	**PP***	**PP**
500	0.03	X	X	0.03	82.1
1000	0.03	X	X	0.05	X
10,000	0.32	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	81960	174	210	1270	987

(e) mathsat solving time: Table & Graph

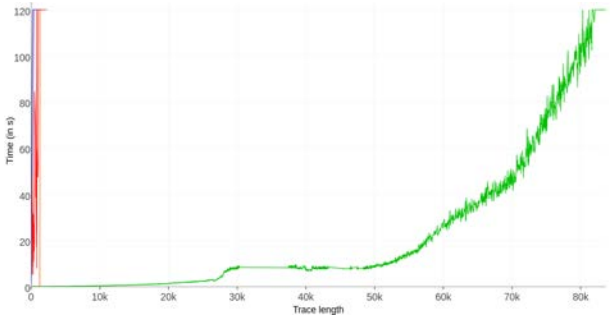
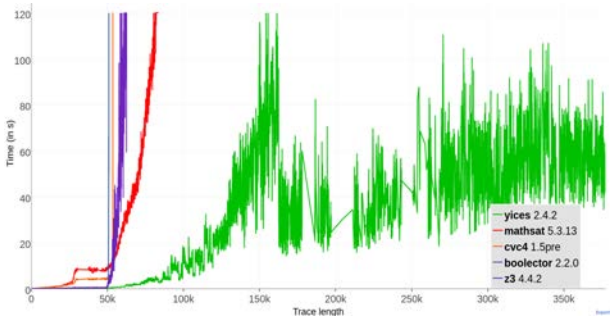


Figure 4.5: C/S policies benchmark by solvers

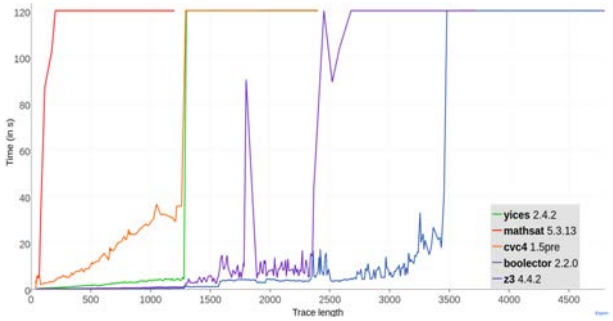
CC	Z3	Btr	CVC4	Yices	Mtsat
500	0.02	0.02	0.04	0.02	0.03
1000	0.03	0.02	0.04	0.02	0.03
10,000	0.07	0.05	0.44	0.06	0.32
100,000	X	X	X	10.7	X
377309	X	X	X	54.37	X
max	62540	50938	53540	377309	81960

(a) CC solving time: Table & Graph



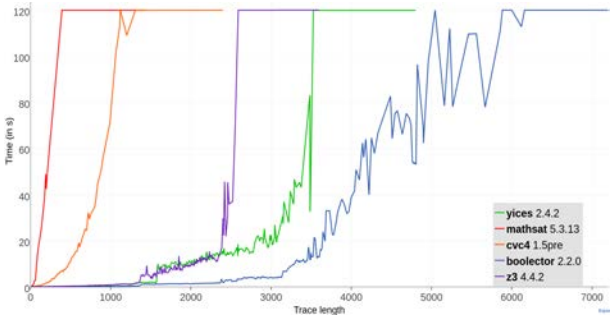
CP	Z3	Btr	CVC4	Yices	Mtsat
500	0.35	0.2	10.25	1.54	X
1000	0.66	0.26	30.4	3.76	X
10,000	X	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	2581	3456	1260	1280	174

(b) CP solving time: Table & Graph



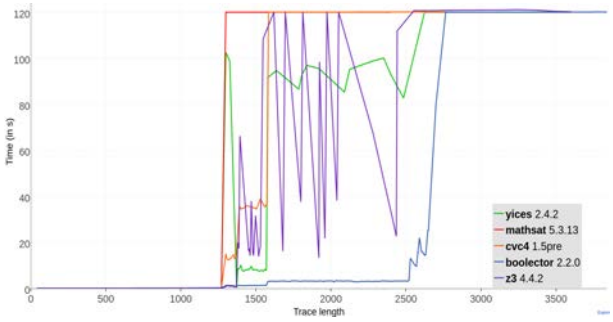
PC	Z3	Btr	CVC4	Yices	Mtsat
500	0.34	0.1	11.17	0.19	X
1000	0.83	0.16	74.5	0.75	X
10,000	X	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	2553	6160	1200	3492	210

(c) PC solving time: Table & Graph



PP*	Z3	Btr	CVC4	Yices	Mtsat
500	0.03	0.08	0.06	0.02	82.1
1000	0.03	0.14	0.11	0.03	X
10,000	X	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	2442	2700	1575	2484	987

(d) PP* solving time: Table & Graph



PP	Z3	Btr	CVC4	Yices	Mtsat
500	0.92	0.46	9.54	0.17	82.1
1000	1.03	0.93	7.42	0.2	X
10,000	X	X	X	X	X
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	1353	2550	1280	1525	987

(e) PP solving time: Table & Graph

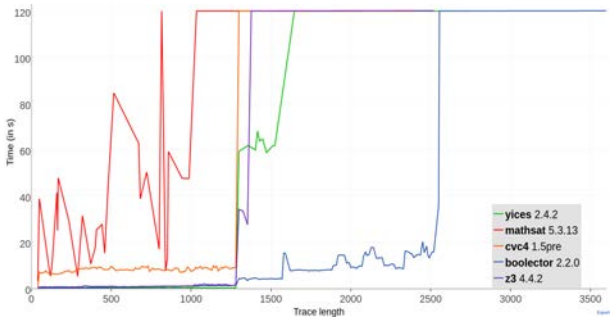


Figure 4.6: C/S policies benchmark by policy

handle each policies and gives the trace length expected to be solved with a 2 minutes timeout. Once again being able to adapt the policy appeared to be crucial depending on the solver at hand.

4.6 Toward extending CSml to syscall/libcalls and more

4.6.1 Definitions

As extension of the policy mechanism formalized and developped for expressions, we extended this mechnanism to library calls, syscalls and undecoded instructions. These are the three main vectors of non-determinism inputs in the DSE execution. Extending C/S for this, would allow to decide in a flexible way whether to concretize some parameters/return values whether to symbolize them. Using this method, irrelevant functions calls can simply be ignored or concretized while really interesting functions can be symbolized. Thus we proposed *a stub mechanism integrating C/S actions*. This mechanism is based on the *C/S injection extension* described earlier in the Chapter. Intuitively, a stub is defined as a sequence of assignments in which each lhs is assigned a value based on an action $\rho*$ defined by $\rho* \triangleq \{\mathcal{S}c, \mathcal{S}e, \mathcal{S}\}$ selected from a policy. We denote $\mathcal{S}c$ the unsound concretization where the lhs is replaced by an arbitrary constant value. We also denote $\mathcal{S}e$ the replacement of a lhs by a logical expression. Let us recall that a lhs is defined by $lhs ::= v \mid v\{i,j\} \mid @[\ e\ ]$. Thus a stub is a sequence of re-assignement of variables (registers) or memory cells. In addition a stub should be shipped with a policy indicating what action to perform on each lhs and a possibly a function providing a specific propagation expression for that lhs in the case the associated action is $\mathcal{S}e$.

Formally. A stub $\mathbb{S}t$, is defined by a tuple $\mathbb{S}t \triangleq (\mathbb{N}_{\mathbb{S}t}, \mathbb{C}_{\mathbb{S}t}, \mathbb{P}_{\mathbb{S}t})$ where $\mathbb{N}_{\mathbb{S}t} \triangleq \{Lhs\}$ defines the set of lhs modified by the stub, $\mathbb{C}_{\mathbb{S}t} \triangleq Lhs \times \Sigma^* \rightarrow \rho*$ defines the actions to perform on a given lhs (as $lhs \subset \mathbb{E}xpr$). Finally, $\mathbb{P}_{\mathbb{S}t} \triangleq \mathbb{E}xpr \times \Sigma^* \rightarrow \mathbb{E}xpr$ defines the function that performs the propagation for the given lhs and return an expression $\mathbb{E}xpr$.

We propose a new evaluation operation *stub_exec* for stub Lhs:

$$stub\_exec(\Sigma^*, \varphi, lhs) \triangleq \left\{ \begin{array}{ll} \Sigma^*(lhs) \leftarrow fresh, \varphi \wedge lhs = fresh & \text{if } \rho = \mathcal{S} \\ \Sigma^*(lhs) \leftarrow \varphi_e, \varphi \wedge \phi' \wedge lhs = \varphi_e & \text{if } \rho = \mathcal{S}e, \Sigma^*(lhs) \leftarrow \varphi_e, e \vdash_{cs\bullet} \varphi_e, \phi' \\ \Sigma^*(lhs) \leftarrow C, \varphi \wedge lhs = C & \text{if } \rho = \mathcal{S}c,\ C \triangleq eval_\Sigma(lhs) \end{array} \right\} \rho \triangleq \mathbb{C}_{\mathbb{S}t}(\Sigma, lhs) \tag{4.1}$$

Finally, a stub can be defined as the execution of *stub_exec* on all the lhs of $\mathbb{N}_{\mathbb{S}t}$. We thus formalize the whole stub execution as defined below:

$$stub\ \frac{\Sigma^*_n, \varphi_n \triangleq stub\_exec(\Sigma^*, \varphi, \mathcal{N}_0) \circ ... \circ stub\_exec(\Sigma^*_n - 1, \varphi_{n-1}, \mathcal{N}_n)}{l, \Sigma^*, \varphi, stub(\mathcal{N}, \mathcal{C}, \mathcal{P}) \rightsquigarrow l+1, \Sigma^*_n, \varphi_n, \Delta(l+1)}$$

Stub kind While doing DSE we need to consider three major kind of stubs. First, external library stubs in the case we do not execute them symbolically, then syscall stubs (dependant of the system and kernel version) and finally not decoded instruction which, even though we can not give them a precise semantic we can approximate their side-effects with stubs. These three kinds of stubs implemented are encompassed by the formalization hereabove.

4.6.2 Library call stubs

Libcalls are any functions (most notably shared library functions) that are intentionally not symbolically executed to lighten the DSE computation. However handling side-effects of this functions is essential to preserve the execution soundness. Thus various stubs are implemented for multiples functions from the `libc` or from the `Windows API`.

Call convention Libcalls stubs have to deal with the call convention used by programs. The call convention defines the way parameters are sent to the *callee* and also defines how the stack should be cleaned-up. There are typically two modes, *caller-based* and *callee-based* where the responsability to restore the stack is given respectively to the *caller* or the *callee*. Secondarily it also defines parameter order on the stack, in other words either they are pushed left-to-right (LTR) or right-to-left (RTL). Known call conventions are:

- `cdecl` used in almost all Unix-based systems. All parameters are transmitted through the stack and the caller is responsible for stack restoration.
- `stdcall` used in Windows. All parameters are sent using the stack, but is callee-based so the function called restore the stack
- `fastcall` The first two parameters are put respectively in `ecx` and `edx`. All the others are pushed onto the stack. This convention is callee-based.

All this convention are RTL conventions. Table 4.13 summaries the properties of the different call conventions. All these considerations have an impact on the way stubs are handled.

Table 4.13: Call convention summary

Name	Parameters order	Order on stack	Stack clean-up
`cdecl`	stack	RTL	Caller
`stdcall`	stack	RTL	Callee
`fastcall`	`ecx`,`edx`,stack	RTL	Callee

Formally, let's consider a call convention $cvt \triangleq \{\texttt{cdecl}, \texttt{stdcall}, \texttt{fastcall}\}$ and a function $get\_args(cvt, i^{th}) \rightarrow \mathbb{E}xpr$ which for a given call convention and parameter number returns the corresponding expression. As a matter of exemple $get\_args(\texttt{cdecl}, 0) = @[esp - 4]$ or $get\_args(\texttt{fastcall}, 1) = edx$. Similarly, we define *epilog* functions performed after a stub to perform a gentle clean-up of the stack. For this, we consider a function $card(\mathbb{S}t) \rightarrow \mathbb{N}$ the function which for a given stub returns its cardinal (i.e its number of parameters). Figures 4.7 gives the semantic of the epilog functions.

$$stdcall - epilog\ \frac{\Sigma^*_{new} \triangleq \Sigma^*[esp \leftarrow fresh] \quad \varphi_{new} \triangleq \varphi \wedge fresh = (\Sigma^*(esp) + (card(\mathbb{S}t) \times (addr\_size/8)))}{\Sigma^*, \varphi, stub\ \mathbb{S}t \rightsquigarrow \Sigma^*_{new}, \varphi_{new}}$$

$$cdecl - epilog\ \frac{}{\Sigma^*, \varphi, stub\ \mathbb{S}t \rightsquigarrow \Sigma^*, \varphi}$$

Figure 4.7: Epilog function semantic

Example To make it clearer we present the stub definition of `void *memcpy(void *dst, void *src, size_t size)`. We have $card(memcpy) = 3$ and $\mathbb{N}_{\mathbb{S}t} \triangleq \{@[dst], @[src], size, eax\}$ with eax the returned value, which according to the specification is equal to dst at return. We want to propagate variables $@[dst]$ and $@[src]$, in addition we also want to concretize $size$ which usually has an undetermined number of iterations. Thus, we have $\mathbb{C}_{\mathbb{S}t} \triangleq \{(@[dst] \mapsto \mathcal{P}), (@[src] \mapsto \mathcal{P}), (size \mapsto \mathcal{C}), (eax \mapsto \mathcal{P})\}$. Lastly, we define the propagation function $\mathbb{P}_{\mathbb{S}t}$ given in figure 4.8.

4.6.3 System call stubs

Syscalls, are dependant on the OS Linux,Windows and dependent on the Application Binary Interface (ABI) on Linux. Syscalls are distinguished by the number given in `eax`. Handling syscall is highly kernel dependent, however, easier than libcalls as there is not calling convention neither the need of epilog. Yet, unless having a dynamic instrumenter working at kernel level, there is no other way than making stubs for syscalls.

$$\mathbb{P}_{\mathbb{S}t}(e) \triangleq \begin{cases} dst & \text{if } e = eax \\ @[src] & \text{if } e = @[dst] \\ \epsilon & \text{if } e = @[src] \\ \epsilon & \text{if } e = size \end{cases} \tag{4.2}$$

Figure 4.8: memcpy(dst, src, size) propagation function stub

4.6.4 Instruction stubs

The symbolic aspect of DSE allows to emancipate from the strict IR decoding. Indeed, due to the variety of `x86` instructions, it is difficult to model all of them precisely. That is why instruction stubs comes handy to give an approximated modeling of an instruction, either over-approximated by symbolizing either under-approximated by concretizing. The purpose is to keep the soundness of execution. Let us take the example of `rdtsc`.

RDTSC is an instruction returning in both `edx` and `eax` the numbers of tick since the last reset of the CPU read in the TSC register. Modeling precisely the content of the TSC content is not simple (and thus not decoded in DBA). Hence, we have $\mathbb{N}_{\mathbb{S}t} \triangleq \{edx, eax\}$, and the propagation function can be modelled either by calling the real `rdtsc` and putting the result in the two registers or more lazily by symbolizing the content `edx,eax`. The second approach consists in choosing $\mathcal{S}$ and a relative fair approach to over-approximating the execution while keeping a satisfiable path predicate.

4.6.5 Implementation

The formalization hereabove, and the implementation made in BINSEC/SE only allows one stub per function or syscall. As a consequence, it forbids to select a different stub for the same function depending on the context.

Each new stub should be implemented at three levels:

- making the policy specification with parameters and returned values;
- implementing the data gathering in PINSEC (used by $eval_{\Sigma}(e)$);
- implementing in BINSEC/SE the propagation function $\mathbb{P}_{\mathbb{S}t}$ modeling the behavior for each parameters or returned values.

Figure 4.9 provides an overview of the global integration of the stub mechanism in the DSE engine.

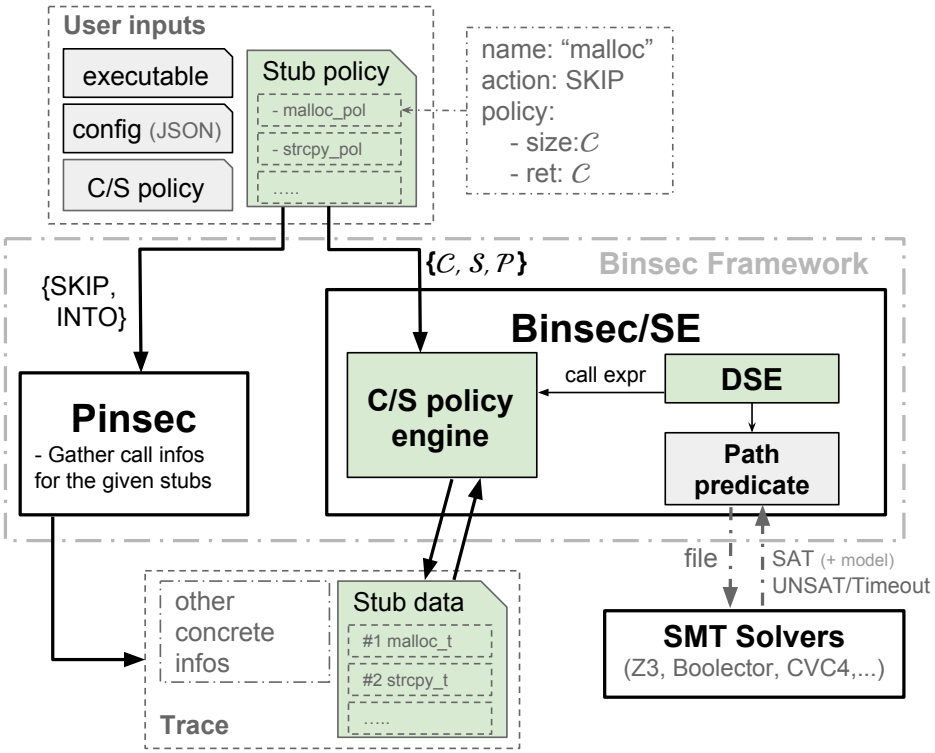


Figure 4.9: Stubs C/S architecture

Chapter 5

Backward-Bounded DSE

This chapter presents a scalable and robust method for solving *infeasibility* queries, a standard problem encountered in reverse-engineering. BB-DSE does not supersede existing DSE algorithm but rather complement them by addressing infeasibility queries in a precise manner.

Problem Dynamic methods only address reachability issues, namely *feasibility questions*, i.e. verifiying that certain events or setting can occur, e.g. that an instruction in the code is indeed reachable. Contrariwise, many questions encountered during reversing tasks are *infeasibility questions*, i.e. checking that certain events or settings cannot occur. It can be used either for detecting obfuscation schemes, e.g. detecting that a branch is dead (i.e. it cannot be taken) or to prove their absence, e.g. proving that a computed jump cannot lead to an improper address.

These *infeasibility issues are currently a blind spot of both standard and advanced disassembly methods*. Dynamic analysis and DSE do not answer the question because they only consider a *finite number of paths* while infeasibility is about considering *all paths*. Also, (standard) syntactic static analysis is too easily fooled by unknown patterns. Finally, while recent semantic static analysis approaches [BHV11; BR10; KV10; SMS11] can in principle address infeasibility questions, they are currently neither scalable nor robust enough.

At first sight infeasibility is a simple mirror of feasibility, however from an algorithmic point of view they are not the same problem. Indeed, since solving feasibility questions on general programs is undecidable, practical approaches have to be one-sided, favoring either feasibility (i.e. answering *"feasible"* or *"I don't know"*) or infeasibility (i.e. answering *"I don't know"* or *"infeasible"*). While there currently exist robust methods for answering feasibility questions on heavily obfuscated codes, no such method exist for infeasibility questions.

Motivation Let us consider the obfuscated pseudo-code given in Figure 5.1. The function `<main>` contains an opaque predicate in (1) and a call stack tampering in (2) (cf. 8.1 and 8.2).

Getting the information related to the opaque predicate and the call stack tampering would allow to:

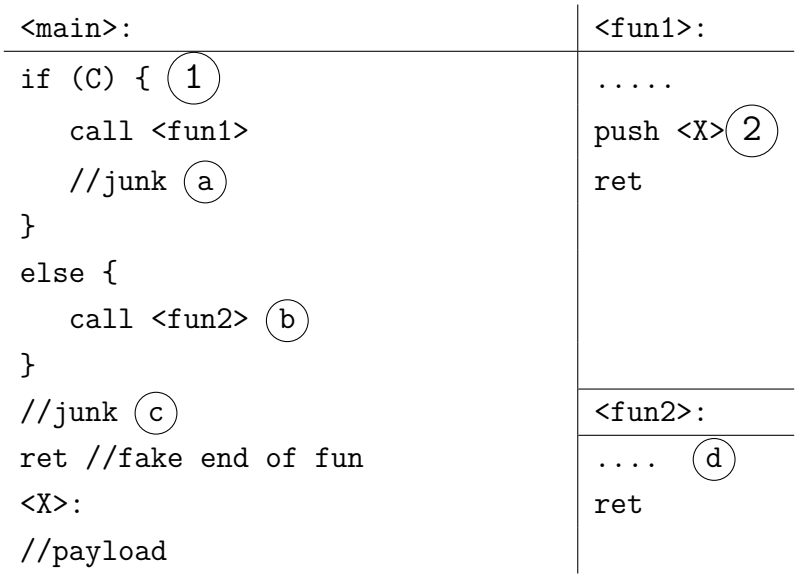


Figure 5.1: Motivating example

- (1) to know that `<fun1>` is always called and reciprocally that `<fun2>` is never called. As consequence (b) and (d) are dead instructions;
- (2) to know that the `ret` of `<fun1>` is tampered and never return to the caller. As a consequence (a) and (c) are dead instructions. Such trick would also allow to hide the real payload located at `<X>`.

Hence the main motivation is not to be fooled by such infeasibility-based tricks that slow-down the program reverse-engineering and its global understanding.

Goal and challenges In this chapter, we are interested in solving automatically infeasibility questions occurring during the reversing of (heavily) obfuscated programs. The intended approach must be *precise* (low rates of false positives and false negatives) and able to *scale* on realistic codes both in terms of size (*efficient*) and protection – including self-modification (*robustness*), and *generic* enough for addressing a large panel of infeasibility issues. Achieving all these goals at the same time is particularly challenging.

Proposition We present *Backward-Bounded Dynamic Symbolic Execution* (BB-DSE), the first precise, efficient, robust and generic method for solving infeasibility questions. To obtain such a result, we have combined in an original and fruitful way, several state-of-the-art key features of formal software verification methods, such as deductive verification [Lei05], bounded model checking [Bie+99] or DSE. Especially, the technique is *goal-oriented* for precision, *bounded* for efficiency and combines *dynamic information and formal reasoning* for robustness.

Impact Backward-Bounded DSE does not supersede existing DSE approaches, it complements them by addressing infeasibility questions. Altogether, this work paves the way

for robust, precise and efficient disassembly tools for obfuscated binaries, through the careful combination of static/dynamic and forward/backward approaches.

Table 5.1: Disassembly methods for obfuscated codes

	feasibility query	infeasibility query	efficiency	robustness
dynamic analysis	✓/×(†)	×	✓	✓
DSE	✓	×	×	✓
static analysis (syntactic)	✓	✓/×(††)	✓	×
static analysis (semantic)	×	✓	×	×
BB-DSE	×	✓(‡)	✓	✓

(†): follow only a few traces
(††): very limited reasoning abilities
(‡): can have false positive and false negative, yet very low in practice

5.1 Formalization

Preliminaries We consider a binary-level program P with a given initial code address a_0. A state $s \triangleq (a, \sigma)$ of the program is defined by a code address a and a memory state σ, which is a mapping from registers and memory to actual values (bitvectors, typically of size 8, 32 or 64). By convention, s_0 represents an initial state, i.e. s_0 is of the form (a_0, σ). The transition from one state to another is performed by the *post* function that executes the current instruction. An execution π is a sequence $\pi \triangleq (s_0 \cdot s_1 \cdot \ldots \cdot s_n)$, where s_{j+1} is obtained by applying the *post* function to s_j (s_{j+1} is the successor of s_j). Similarly, we call s_j the predecessor of s_{j+1} if s_{j+1} is obtainted by application of the *post* function on s_j.

Let us consider a predicate φ over memory states. We call *reachability condition* a pair $c \triangleq (a, \varphi)$, with a a code address. Such a condition c is *feasible* if there exists a state $s \triangleq (a, \sigma)$ and an execution $\pi_s \triangleq (s_0 \cdot s_1 \cdot \ldots \cdot s)$ such that σ satisfies φ, denoted $\sigma \models \varphi$. It is said *infeasible* otherwise. An *feasibility (resp. infeasibility) question* consists precisely in trying to solve the feasibility (resp. infeasibility) of such a reachability condition.

These definitions do not take self-modification into account. They can be extended to such a setting by considering code addresses plus waves or phases [Bon+15].

Principles We build on and combine 3 key ingredients from popular software verification methods:

- backward reasoning from deductive verification, for *precise goal-oriented reasoning*;

- combination of dynamic analysis and formal methods (from DSE), for *robustness*;
- bounded reasoning from bounded model checking, for *scalability* and the ability *to perform infeasibility proofs*.

The initial idea of BB-DSE is to perform a **backward reasoning**, similar to the one of DSE but going from successors to predecessors (instead of the other way). Formally, DSE is based on the *post* operation while BB-DSE is based on its inverse *pre*. Perfect backward reasoning pre^* (i.e. fixpoint iterations of relation *pre*, collecting all predecessors of a given state or predicate) can be used to check feasibility and infeasibility questions. But this relation is not computable.

Hence, we rely on computable **bounded reasoning**, namely pre^k, i.e. collecting all the "predecessors in k steps" (k-predecessors) of a given state (or predicate). Now symmetry does not hold anymore: while pre^k can answer *positively to infeasibility queries* (if a predicate has no k-predecessor, it has no k'-predecessor for any $k' > k$ and cannot be reached), but *cannot falsify them* (because it could happen that a predicate is infeasible, for a reason beyond the bound k). Moreover, it is efficient as the computation does not depend on the program size or trace length, but on the user-chosen bound k.

In practice, checking whether $pre^k = \emptyset$ can be done in a symbolic way, like it is done in DSE: the set pre^k is computed implicitly as a logical formula (typically, a quantifier-free first-order formula over bitvectors and arrays), which is unsatisfiable iff the set if empty. This formula is then passed to an automatic solver, typically a SMT solver [12] such as Z3.

Yet, backward reasoning is still very fragile at binary-level, since computing *pre* in a perfect way may be highly complex because of dynamic jumps or self-modification. The last trick is to combine this pre^k reasoning with **dynamic traces**, so that the whole approach benefits from the robustness of dynamic analysis. Actually, the pre^k is now computed w.r.t. the control-flow graph *induced by a given trace* π – in a dynamic disassembly manner. We denote this *sliced* pre^k by pre^k_π.

Hence we get *robustness*, yet since some real parts of pre^k may be missing from pre^k_π, we *now lose correctness*: we may have false positive FP (because pre^k_π will be incomplete w.r.t pre^k), additionally to the false negative FN due to "boundedness" (because of too small k). A picture of the approach is given in Figure 5.2.

Algorithm Considering a reachability condition (a, φ), BB-DSE starts with a dynamic execution π and iterates it in order to solve φ at the given location. The algorithm is given in 2.

As a summary, this algorithm enjoys the following good properties: it is efficient (depends on k, not on the trace or program length) and as robust as dynamic analysis. On the other hand, the technique may report both false negative (bound k too short) and false positive (dynamic CFG recovery not complete enough). *Yet, in practice, our experiments demonstrate that the approach performs very well, with very low rates of FP and FN.* Experiments are presented in Sections 8.1.4, 8.2.4, 10.1 and 10.2.

By convenience, we will not distinguish anymore between the predicate φ and the reachability condition (a, φ) if a is clear from context.

Algorithm 2: BB-DSE algorithm

Input: a dynamic execution trace π, a reachability condition (a, φ)
Output: TS the infeasibility status of φ at location a
$TS := \emptyset$;
switch $pre^k_\pi((a, \varphi))$ **do**
  case *UNSAT:* **do** $TS :=$ INFEASIBLE;
  case *SAT:* **do** $TS :=$ UNKNOWN;
  case *TIMEOUT:* **do** $TS :=$ TIMEOUT;
end
return TS;

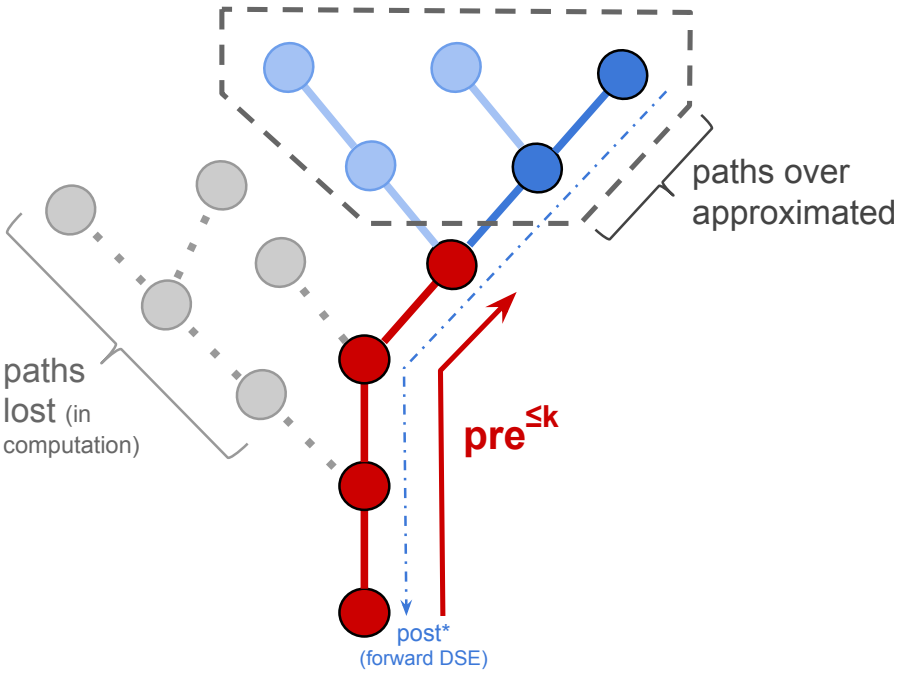


Figure 5.2: pre^k schema

5.2 Solving Infeasibility Questions with BB-DSE

We show in this section how several natural problems encountered during deobfuscation and disassembly can be thought of as infeasibility questions, and solved with BB-DSE.

Opaque predicates (OP) is a predicate always evaluating to the same value. Intuitively, to detect an opaque predicate the idea is to backtrack all its data dependencies and gather enough constraints to conclude to the infeasibility of the predicate. Considering $p = (a, \varphi)$ the pair address-predicate for which we want to check for opacity and π the execution trace under attention we have:

- if p is dynamically covered by π, then returns FEASIBLE;
- otherwise, returns BB-DSE (p), where INFEASIBLE is interpreted as "opaque".

This obfuscation is addressed in detail in Chapter 8.

Call stack tampering Call stack tampering consists in altering the standard compilation scheme switching from function to function by associating a `call` and a `ret` and making the `ret` to return to the call next instruction. The `ret` is tampered (a.k.a violated) if it does not return to the expected return site pushed on the stack at the call. This infeasibility problem can be addressed with the predicate $@[esp_{\{call\}}] = @[esp_{\{ret\}}]$. It compare the content of the value pushed at call $@[esp_{\{call\}}]$ with the one used to return $@[esp_{\{ret\}}]$. If it evaluates to UNSAT, a violation necessarily occurs. A complete taxonomy and detection algorithm is discussed in Chapter 8.

Opaque constant Similar to opaque predicates, opaque constants are expressions always evaluating to a single value. Let us consider the expression e and a value v observed at runtime for e. Then, the opaqueness of e reduces to the infeasibility of $e \neq v$.

Dynamic jump closure When dealing with dynamic jumps, switch, etc., we might be interested in knowing if all the targets have been found. Let us consider a dynamic jump `jump eax` for which 3 values v_1, v_2, v_3 have been observed so far. Checking the jump closure can be done through checking the infeasibility of $eax \neq v_1 \wedge eax \neq v_2 \wedge eax \neq v_3$.

Virtual Machine & CFG flattening Both VM obfuscation and CFG flattening usually use a custom instruction pointer aiming at preserving the flow of the program after obfuscation. In the case of CFG flattening, after execution of a basic block the virtual instruction pointer will be updated so that the dispatcher will know where to jump next. As such, we can check that all observed values for the virtual instruction pointer have been found for each flattened basic block. Thus, if for each basic block we know the possible value for the virtual instruction pointer and have proved it cannot take other values, we can ultimately get rid of the dispatcher.

A glimpse of conditional self-modification Self-modification is a killer technique for blurring static analysis, since the real code is only revealed at execution time. The method is commonly found in malware and packers, either in simple forms (unpack the whole payload at once) or more advanced ones (unpack on-demand, shifting-decode schemes [Uga+15]).

The example in Figure 8.1 (page 106) taken from `ASPack` combines an opaque predicate together with a self-modification trick turning the predicate to true in order to fool the reverser. Other examples from existing malwares have been detailed in previous studies (NetSky.aa [YD15]).

Dynamic analysis allows to overcome the self-modification as the new modified code will be executed as such. Yet, BB-DSE can be used as well, to *prove interesting facts about self-modification schemes*. For example, given an instruction known to perform a self-modification, we can take advantage of BB-DSE to know whether another kind of modification by the same instruction is possible or not (conditional self-modification). Let us consider an instruction $mov\ [addr],\ eax$ identified by dynamic analysis to generate some new code with value $eax = v$. Checking whether the self modification is conditional reduces to the infeasibility of predicate $eax \neq v$.

5.3 Benchmark against forward DSE

We compare BB-DSE with standard forward DSE, as well as with (unbounded) backward DSE. We are interested in comparing their efficiencies and their adequacy to infeasibility questions – through the distribution of their results, between SAT, UNSAT and timeout. The experiment is performed on a trace of 115000 instructions and we check at each conditional jump if the branch not taken is infeasible (UNSAT) or not (SAT), which is equal to checking if the branch is dead. For BB-DSE, we take the algorithm for opaque predicate detection described in Section 5.2, with bound values $k = 100$ and $k = 20$. We argue in latter experiments (Section 8.1) that $k = 20$ is a reasonable bound. We use the forward DSE of BINSEC/SE, and backward DSE is obtained from BB-DSE with a bound set to ∞.

Results are presented in Table 5.2. While forward and backward DSE provide similar results, BB-DSE clearly surpasses them in terms of efficiency, spending less than a second for every predicate without any timeout ($\geq$ 2000 with DSE). From a result point of view, BB-DSE with k=16 returns very few UNSAT answers compared to the other methods (54 vs $\geq$ 7000). Actually, this was expected since DSE takes the whole path into account, and while dead branches are rare in normal code, dead paths are very common.

Conclusion This preliminary experiment gives a clear demonstration on the advantages of BB-DSE over DSE on infeasibility questions. Indeed, besides the dramatic gap in efficiency (which was of course expected since DSE depends on the whole size trace), DSE reports far more infeasible branches – which would lead in practice to too many false positives. These results were expected, as they are direct consequences of the design choices behind DSE and BB-DSE. On the opposite, BB-DSE is not suitable for feasibility questions.

Table 5.2: Benchmark DSE versus BB-DSE

	bound k	Cond. branch			Total time
		#SAT	#UNSAT	#Timeout	
forward DSE	/	575	**7749**	**2460**	**17h43m**
backward DSE	∞	575	7748	2461	17h48m
BB-DSE	100	3378	7406	0	18m78s
BB-DSE	20	10730	**54**	**0**	**4m14s**

As such, we provide throughout benchmarks of BB-DSE in Chapter 8 and real world application in Chapter 10.

Part III

Implementation & Optimizations

Chapter 6

Dynamic Symbolic Execution Platform

6.1 Overview

Our DSE algorithm has been implemented in the multipurpose Binsec framework [DB15]. Far from being DSE centric, Binsec provides low-level API to decode and manipulate instructions in DBA. At the top of it, static analyses, simulation features have been implemented with some interaction in between them. Figure 6.1 shows the overall landscape of functionalities.

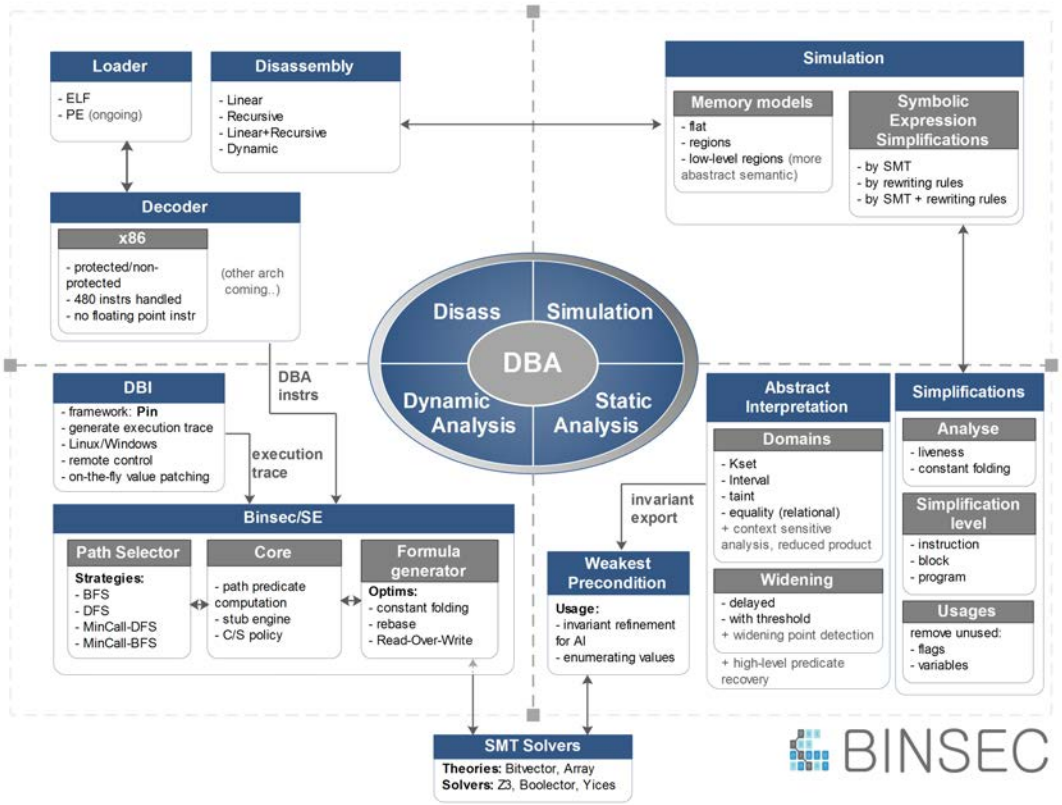


Figure 6.1: Binsec features overview

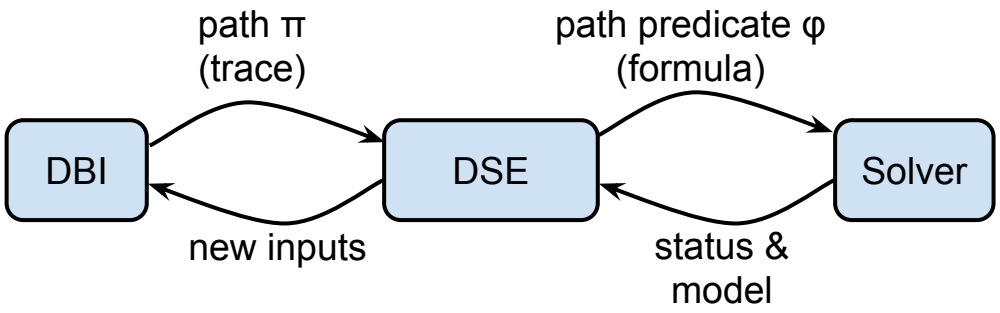


Figure 6.2: Classic DSE analysis workflow

The DSE is usually performed by three components described in Figure 6.2. First, the Dynamic Binary Instrumentation (DBI) generates an execution trace transmitted to the symbolic execution engine which computes the path predicate and generates a formula sent to an automatic solver. As a response, the solver returns a status and a model which can optionally be used to generate new inputs. These inputs can subsequently be sent back to the DBI to generate a new trace.

6.2 Dynamic Instrumentation

6.2.1 Introduction

Several different instrumentation systems are being used for DSE. Probably the most famous is Pin [Luk+05] developped by Intel. It allows to instrument x86 programs both 32 and 64 bits by embedding at runtime a shared library that will hijack the normal control flow of the program to instrument it. No debug feature is used by Pin which helps bypassing basic anti-debugging tricks. In the same vein, DynamoRIO provides similar functionalities [BZA12]. A possible alternative is the use of the virtualization engine Qemu [Bel16] (developped by the awesome Fabrice Bellard). Last, the promising Panda engine [Dol+15] based on TCG, LLVM and Qemu which unlike other DBI allows repeatable instrumentation and backward execution.

6.2.2 Pinsec

Pinsec is the instrumenter developped for this thesis. Based on Pin, and developped in C++ it accounts for approximately 2000 loc and is able to instrument both Linux and Windows binaries. Execution traces are generated in protobuf[3] to be parsable and used in any language. Among advanced functionalities, it allows to gather function parameters for some libc, and Windows API functions; it also allows to retrieve arbitrary values in memory and registers. It eventually provides a mechanism for on-the-fly value patching (mostly used to force a given path). Finally, via a messaging API, it allows some remote

[3] https://developers.google.com/protocol-buffers/

control actions for dynamic interaction with the symbolic execution engine (see Section 6.6).

Configuration. Basic parameters given to the pintool are, *-start-entrypoint* to start at the entry point of the program or more generically *-start* to begin at a specific address. That is especially useful if we want to skip all the initializations performed at the begining of the program. As a remplacement *-function* allows to provides a function name to instrument which requires symbols to be present. The *-out* option is used to specify the output trace file. To limitate the trace length or the instrumentation time *-maxlength* or *-timeout* are available. Lastly, it is possible to activate the tracing of self-modification layers with *-trace-waves* that will tag all addresses written in memory. If at some point of the execution the program jumps on a tagged address a new wave information is recorded in the trace.

In addition to parameters given via the command line, an instrumentation can be configured via a JSON file, providing more advanced features (*-config* command line argument). The configuration permits to define in a granular manner which function to trace and which function to "skip". It also allows to define the policy to adopt for all library calls or syscalls, namely either to trace it or not, or to retrieve theirs values etc.

Input injection A key feature, for path selection and iterating paths is the ability to inject inputs, or values during the execution, forcing the value of a certain register or memory cell at some point of the execution. We define $T \triangleq \{R, M, I\}$ the kind of inputs with R a register, M a memory address, I an indirect pointer. The idea is to alterate the dynamic execution of the program to patch values or to gather some value for later usage (concretization etc). Indeed, these values can be later used for C/S injection in the dynamic symbolic engine. We define $A \triangleq \{\mathcal{S}, \mathcal{C}, (\mathcal{P}t \times Bv)\}$ the set of possible actions to perform for that input with $\mathcal{S}$ the pintool ignores it, $\mathcal{C}$ the value is retrieved and $\mathcal{P}t$ the value is patched with the given value. An input is as a rule of the form $guard \Rightarrow A$ of which *guard* defines the predicate where to apply the input action. This predicate is defined by an adress, an occurence and $W \triangleq \{\texttt{Be}, \texttt{Af}\}$ defining respectively before and after the given address. An input rule is defined by:

$$\begin{array}{ccccccccc} type & :: & address & :: & when & :: & iter & \Rightarrow & action \\ T & :: & Bv & :: & W & :: & \mathbb{N} & \Rightarrow & A \quad \Leftrightarrow \end{array}$$

As a matter of example `eax :: 0x801040 :: Be :: 0` $\Rightarrow$ `(`$\mathcal{P}t$`, 0x0)` indicates to patch eax with value 0x0 at the first occurence of location 0x801040 before the instruction is being executed.

Performances. This pintool was not especially design to run fast and not especially designed to be compact (because of protobuf). For instance, at load-time all instructions are cached for a quick posterior access. Yet, performing this task slow-down the effective starting of the program. Additionally, we need to instrument the program at the instruction level while Pin also allows to instrument at the basic-block or function level which is far more efficient. As a matter of appreciation the table 6.1 shows empirical statistics

about instrumentation performances. The table shows the trace length, native execution time compared to instrumentation time and the average execution time per instruction. The benchmark was performed on a Windows virtual machine so all the results are relative to the performances of the VM. The initialization phase is quite slow, as consequence longer trace have a better ratio of instruction/sec. The benchmark is composed of 6 packers later used for large scale benchmarks. It should be taken into account that a pintool by design usually induces a slow-down from 7.8x to 20x [Luk+05] compared to native execution when instrumenting at basic-block level. Thus, a significant time is spent in the DBI itself independently from the pintool implemented. In addition, we perform analysis at the instruction level which is very costly in terms of time. *All the packers executions take less than 3 minutes which is efficient-enough for automated analysis of large binaries data-set.*

Table 6.1: PINSEC execution time overhead comparison

	#instrs	native in (s)	pinsec in (s)	instrs per sec
Aspack	377,310	0.015	75.69	4985
Expressor	635,285	0.995	169.84	3740
MoleBox	5,257,032	0.735	231.39	22719
Mystic	4,531,677	0.047	116.50	38899
PELock	2,389,016	0.165	189.68	12595
WinUpack	657,406	0.016	77.75	8455
Avg	2,307,954	0.329	143.47	16086.67

6.3 Symbolic Execution Engine

6.3.1 Introduction

As mentioned earlier, the symbolic execution engine is supposed to perform both the path selection ($\mathbb{S}el$) and path predicate computation ($\mathbb{C}$). Among existing DSE working at binary level, we can notice Mayhem [Cha+12], Fuzzball [Cas+13], S2E [CKC12], Angr [Sho+16] and the promising recent Triton [SS15]. What differentiates them, is mostly architectures supported, theories used for formulas, solvers supported, path covering strategies and additional analyses like abstract interpretation, tainting or data dependencies.

6.3.2 Binsec/SE

BINSEC/SE is the symbolic engine developped in the BINSEC platform. As the DBI engine it is parameterized with JSON configuration files. At the time of writing it represents more than 10K OCaml lines of code. Like most engines an analysis can be implemented

using a callback mechanism (see Figure 6.3). It supports `Z3`, `boolector`, `CVC4` and `Yices`. Nonetheless, what differentiates it from the existing DSE engines are the modular path predicate computation (i.e concretization/symbolization), all the optimizations implemented at the formula level and the different stub mechanisms for library and system calls. All these differences are elaborated in Part II and Chapter 7.

Path predicate computation

Path predicate computation is the cornerstone of computation. It is in charge to evaluate the semantic of each instruction and update the path predicate accordingly. The computation in BINSEC/SE also integrates the C/S policy mechanism. The path predicate computation time is generally negligible compared to solving time [Dav+16a]. Meanwhile for the sake of evaluation we measured the path predicate computation time for different C/S policies (cf. 4).

Table 6.2: Path predicate computation time and instruction/seconds

	trace len	CC	CP	PC	PP*	PP
Aspack	377,310	22.87	23.02	18.67	19.19	17.34
Expressor	635,285	37.30	37.94	30.38	31.40	28.84
Crypter	1,170,108	69.79	71.01	61.04	61.51	57.73
PELock	2,389,016	136.79	141.81	122.30	114.82	109.32
WinUpack	657,406	51.45	49.83	43.16	47.58	41.15
Yoda's Crypter	240,900	18.52	19.29	14.54	15.96	15.22
avg i/s		16245.02	15952.25	18856.30	18832.28	20289.41

The benchmark given in table 6.2 aims at evaluating the impact of C/S policies on the path predicate computation. We compute both the execution time and the number of instructions processed per seconds. The idea is to have a rough apreciation of the throughput. For that, we selected few random binaries and performed the computation on them. *This includes the parsing of instructions in the protobuf trace, which represents a significant amount of time.* PP correspond roughly to the case where no C/S policies is applied as everything is propagated like in standard DSE. The difference with CC is only 4044 instructions per seconds which represent a gain of 19%. Considering CC to be an heavy policy we can conclude that the C/S policies computation takes at most 20% of the path predicate computation (still negligible compared to solving time).

Path selection

Beside a fine-grained control of a single execution trace, automatic multiple paths exploration is also a desired feature of a DSE tool. Ideally, the path exploration engine should allow either to fulfill some standard coverage requirements, or to focus on specific parts of the code through dedicated (user-defined) search heuristics. It relies on a simple API allowing to easily implement various exploration strategies. In particular, this API offers a function `select(S)` returning the "best" trace from a set of traces `S`, w.r.t. a

user-defined `score` function. Scores are built from several (predefined or dynamically computed) traces characteristics, such as length, last instruction call-depth, distance to a given target, etc. This approach allows to precisely define a wide-range of exploration strategies. Several strategies are already implemented such as DFS, BFS, random path, MinCall-DFS and MinCall-BFS [Bar+13]. The Depth-First-Search (DFS) strategy might get stuck in narrow parts of the code and Breath-First-Search might get stuck in loops. Thus, it is worth considering alternate strategies. For instance MinCall-DFS selects the longest path having the lowest call depth. This strategy ensure to hit all the functions while keeping the longest path possible to have a maximum coverage. MinCall-BFS works similarly but consider the shortest path and the lowest call depth.

Analysis development

Internally, an analysis is represented as an OCaml class, which each analysis should inherit from. This class provides appropriate callbacks and data structures for implementing any DSE-based analysis. Main callbacks are:

- `pre_execution`, `post_execution` triggered respectively once at the begining and once at the end of the analysis;
- `visit_instr_before`, `visit_instr_after` triggered before (resp. after) every assembly instruction of the trace;
- `visit_dbainstr_before`, `visit_dbainstr_after` triggered before (resp. after) every DBA instructions of an `x86` instruction;

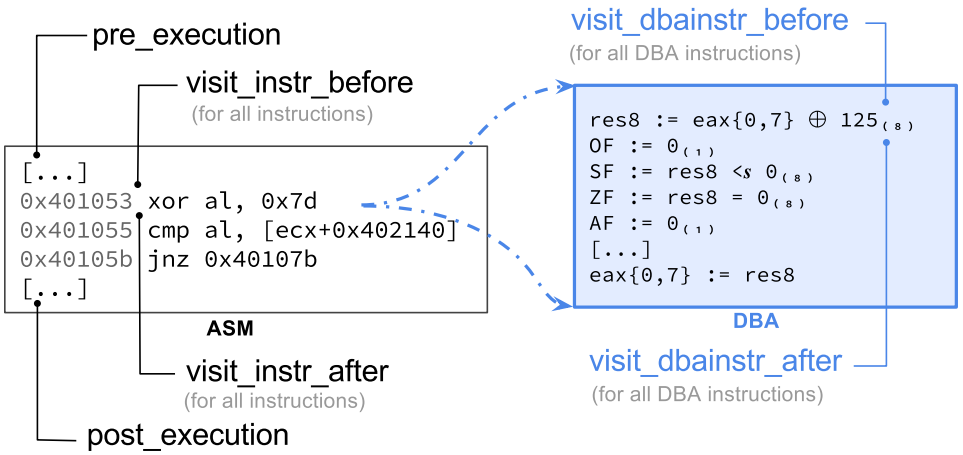


Figure 6.3: BINSEC/SE analysis callbacks

A summary of all callback location calls is given in Figure 6.3. These callbacks allow to apply specific actions at a specific step and/or location along the execution in a highly configurable manner. They help developing new analyses without a deep understanding of the whole inner-working of the DSE.

6.3.3 Memory Management & Initial State

In standard DSE we usually assume `argv` and the environment variable `env` to be inputs in the program memory. Later in the execution, allocated memory can also become inputs. In practice, all the other memory addresses in memory are either not mapped, or not accessible for writing due to theirs belonging to the kernel. This differs from the closed quantifier-free array theory over the bitvectors used QF_ABV. Indeed, by default an array is totally unconstrainted and the bounds are limited to the size of bitvector indexes. Then, the solver is free to give a valuation for any unconstrainted memory cells which can lead to meaningless results. To address this issue the main solutions remaining to be implemented are:

- constraining the *load* operations by enforcing possible addresses,
- dumping the initial state,
- potentially using, taint analysis to force load operation to be performed at tainted addresses (also called "controlable").

6.4 Automatic Solving

6.4.1 Introduction

There are two kinds of automatic solvers used. First constraint-programming based solvers like or-tools [Goo16b] from Google or also Colibri [Che+14]. Second, Satisfiability Modulo Theories(SMT) solvers based on DPLL. Binary-level DSE requires `bitvectors` theory and, in most cases, the `array` theory. Known solvers supporting these two theories are `Z3` [MB08], `CVC4` [Det+14], `boolector` [NPB15] and `Yices` [Dut14].

Nevertheless, their difference tends to be very negligible as most of them now handle the smtlib2 format [BFT16]. What differenciate solvers are the theories supported and the usability features included like interactive or incremental modes.

6.4.2 Solving in Binsec/se

Binsec/se embeds an internal representation of formulas, subset of the SMTLIB2 acting only on bitvectors and arrays. This representation can be exported to SMTLIB2 as a common format with external solvers. Binsec/se does not provide direct binding with solvers. The two ways to solve formulas are:

- files in SMTLIB2 format,
- interactive mode, for solvers supporting this functionality. This allows to use the two commands `push` and `pop` to remove a posteriori, some terms of the formula.

Although we use a common format, some solvers may require specific processing. For instance, some of them do not support timeout as command line argument or generated models are not necessarily in the same format. Main differences lay in the support of

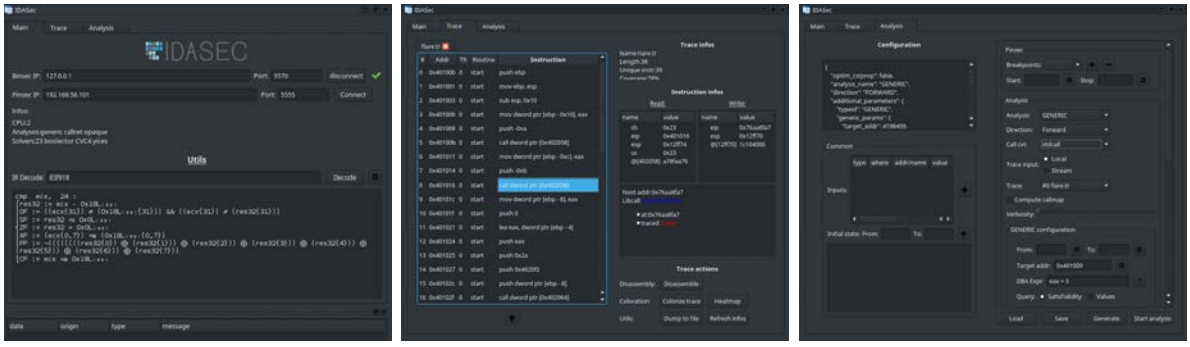
arrays as function parameters. Indeed, BINSEC/SE uses two functions as syntactic sugar for reading, or writing 32 bits values in an array. One of them is the `load32_at` function given in listing 6.1. It is invoked the same way as `select` but returns a 32bits value. This greatly improves the formula readability. Unfortunately, `boolector` does not support arrays as function parameters, as a consequence it should be inlined.

```
(define-fun load32_at ((memory (Array (_ BitVec 32) (_ BitVec 8))) ; arg1
                       (addr (_ BitVec 32)))                       ; arg2
                       (_ BitVec 32)                               ; return
        (concat (concat (concat
        (select memory (bvadd #x00000003 addr))
        (select memory (bvadd #x00000002 addr)))
        (select memory (bvadd #x00000001 addr)))
        (select memory addr))
)
```

Listing 6.1: load32_at SMT function

6.5 IDASec: DSE analysis vizualisation

As fullfilment of one of the contributions aiming at lifting analysis results to make them usable for the reverse-engineer. I developed an IDA plugin called IDASEC facilitating the triggering of an analysis and the exploitation of results. This plugin represents more than Python 7000 lines of code.



(a) Decoding (b) Trace view (c) Analysis configuration

Figure 6.4: IDASec main UIs

Basic features allow to request the DBA semantic for an instruction in order to understand its side-effect, as shown in Figure 6.4a. But the main purpose of IDASEC is to manipulate dynamic traces. Apart from being able to load a trace and vizualize it (as shown in Figure 6.4b) it allows to highlight instructions executed, creates heatmap of instructions and some dynamic disassembly features. In the later case, the idea is to follow the trace and disassemble instruction that would have been missed or wrongly disassembled by IDA.

Thank's to an internal DBA representation and a tiny SMTLIB2 representation it can perform some computation on data and values in order to generate reports or charts. From an analysis perspective, some obfuscation specific features are implemented to highlight dead code, annotate code or compute various statistics. Global aim is to integrate DSE analysis in the main tool which is used in reverse-engineering so as to facilitate data exploitation in an user-friendly manner.

6.6 Components Interaction

The communication between the three entities is being made using a message queueing protocol base on ZeroMQ[4]. This framework has been chosen as it provides great bindings in the three languages involved C++, Python and OCaml. In this context, Binsec/se works as a server handling analysis requests over the network. A request is formed by a configuration file serialized in protobuf, followed by the execution trace splitted in chunks. A chunk is a buffer of instructions, transmitted when filled. The size of a chunk can be parametrized with the argument `-chunk-size` in Pinsec. This changes from the classical workflow where arguments are sent on the command line and the trace read from a file. When running as a server Binsec runs multiple worker to receives different requests simultaneously. The architecture is modular, thus various configurations are possible:

- Pinsec ↔ Binsec for direct streaming, of the execution trace to Binsec (see Section 6.7 for a concrete application),
- Idasec ↔ Binsec for static symbolic execution. Paths can easily be created using IDA. (see Section 10.2 for an application),
- Pinsec ↔ Idasec ↔ Binsec for a complete interaction between the three entities. This configuration allows a dynamic and simultaneous interaction with the program instrumented and the lifting of resulting informations in IDA, all in the same time.

All messages transmitted over the three entities are of the form `COMMAND`, `DATA` where `COMMAND` indicates the `DATA` kind or a control message. From the Binsec/se side, at the time of writing three commands are supported:

1. `DECODE_INSTR`: to decode the semantic of an instruction into DBA,
2. `REQUEST_INFOS`: to have analysis available, solver available etc,
3. `START_ANALYSIS`: that takes a configuration as `DATA` and triggers the appropriate analysis accordingly.

Pinsec, support few commands to interact with the instrumentation[5]. Supported commands are:

[4] http://zeromq.org/

[5] It does not used the standard GDB remote debugging interface for now

- `RESUME` resuming the instrumentation after a breakpoint hit. When a breakpoint is triggered PINSEC send a `BP_REACHED` to the symbolic executor,
- `PATCH_ZF` allowing to patch ZF forcing a specific value. This is being used to force a program to take certain branches.

API. Within an analysis in BINSEC/SE two additional callbacks are available to interact with the different entities:

- `send_message` sending a command, to any of the other components. Note, that by default when running a remote symbolic execution, all log messages are sent to the remote entity,
- `input_message_received` triggered when a message is received from an external component

6.7 In practice: A crackme example

A `crackme` is a challenge simulating a real world situation where binary programs are designed to make the analysis difficult. The goal is usually to find a string key (the flag) allowing to validate the challenge.

Flare-On is a reverse engineering challenge organized since 2014 by the FLARE team of FireEye Security[6]. This section analyses the first challenge of 2015 since it is straightforward enough to be discussed in detail and will put in action all features described hereabove. This crackme is a 32 bits Windows program which asks for a password and prints "You are success" or "You are failure" depending on the keyboard input. Each byte of the input (`input_buffer`) is xored with `0x7d`, the result is then checked against a key stored in the data section (`data_str`), cf. Figure 6.5.

Solving the crackme by symbolic execution aims at checking that the predicate `ZF=1` is true at location `0x40105b`; and this, for every character of the key. If the generated formula is satisfiable, then the right character can simply be retrieved in the formula solution at `input_buffer[ecx]`. One last remaining problem is the initial state. If none is specified, the solver is allowed to give any valuation to the content of `data_str`, while it is read-only data. Hence, an initial state constraining the value of the `data_str` bytes is required in order to get meaningful results for `input_buffer` (cf. Section 6.3.3). Finally, iterating over `input_buffer` can be done in two ways: a manual offline way, and a fully automated way (more complex).

The simple (painful) way From a trace taking the fail branch, we should solve `ZF=1` at `0x40105b` to get the right char and inject it back as input via the configuration file (cf: 6.2.2) for generating a new trace that will go for a second loop iteration, repeating until the whole key is found.

[6] `http://www.flare-on.com/`

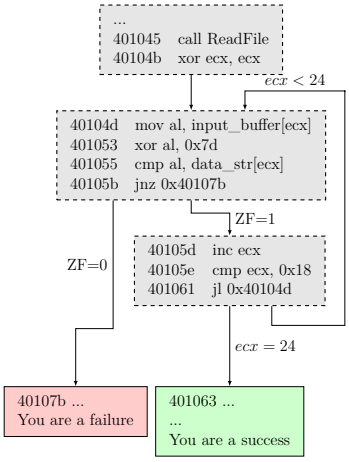


Figure 6.5: Flare-On #1 CFG decoding loop

The automated way Configures a breakpoint at `0x40105b` (the address of `jnz`), compute the right character value, send a command to patch the `ZF` flag when the command `BP_REACHED` is received and resumes instrumentation to force the tracer to take the right branch and to loop again. This fully automatic method makes usage of both the remote control features and the incremental solving (cf. Section 6.4.2). This method allows to solve all the character values in $\mathcal{O}(n)$ w.r.t trace length.
Finally, we obtain the right key solving the challenge, `bunny_sl0pe@flare-on.com`[7].

[7] A complete demo is available at `https://youtu.be/0xUc2jbpjQo`

Chapter 7

Path predicate optimizations

Optimizing the path predicate at the formula level is a crucial task as solving it is the most costly part of DSE. Furthermore, prior knowledge of the analysis and program allows to perform some relevant *context-sensitive* pre-processing optimizations. As an example, some optimizations will greatly take advantage of the way memory accesses are performed. First, pre-processing optimizations implemented are aiming at reducing the complexity in space/time, but also provide normalized representation of formulas. This normalized representation is used for optimizations to benefit from one another. Second we propose array optimizations aiming at reducing constraints on the array representing the memory. Benchmarks preformed shows in best cases, a tremedous speed-up in solving time.

7.1 Pre-processing optimisations

This section describes four different pre-processing optimizations: (1) *backward pruning* aiming at removing all unused terms in the formula, (2) *constant propagation* which performs several syntactic simplification, (3) *rebase*, an original optimization normalizing some expression and (4) *high-level predicate* aiming at lifting low-level `x86` comparison operations into natural conditions. Table 7.1 summarizes the optimizations and which solver benefits from each of them.

Table 7.1: Summary pre-processing optimizations

pruning	constant propagation	rebase	high-level predicates
∼`mathsat`	∗	∗	∼`Yices`

∗: enables other optimizations

7.1.1 Backward-pruning

The main pre-processing operation performed by most DSE engines is the *backward-pruning* phase. The objective is to remove all unused or irrelevant terms in the formula

by doing a backward pass starting from the assert property to check. All variable terms used in any assert of the formula are then kept. The listing 7.1 gives an example where the value of `esp3` is checked. Recursively, `eax, esp1` and `esp0` are then used.

```
esp1 := esp0
esp2 := esp1 - 4
esp3 := eax + esp1
(assert esp3 == #x00000004)
```

Listing 7.1: Backward-pruning example

Experiments Regarding benefits measurement, we compare the number of terms in formulas with and without pruning. Table 7.2 provides the formulas statistics with and without the pruning phase. This benchmark has been performed on a 377300 instructions path predicate. The gain in space is significant, it varies from a 129Mo formula file with more than 1M terms to a 22Mo formula file with 239916 terms. The gain in solving time is mitigated. For example, as shown in Figure 7.1 `mathsat` can solve a 72k not-pruned trace, it can solve a 82k pruned trace with a 2min timeout. Solving times, appeared not to be significantly better if not worse for some solvers (`Z3`, `CVC4`). Thus this optimisation is more valuable for formula compactness and readability rather than any solving time improvements.

Table 7.2: Backward-Pruning statistics

	#inputs	#variables	#constraints	#terms (total)
No pruning	619	1,173,269	218,436	1,392,324
With pruning	**61**	**155,593**	**84,262**	**239,916**
Gain	x10	x7.5	x2.5	x5.8

As of now, unless explicitely mentioned all the other optimizations and benchmarks use this generic post-processing pass on the formula.

7.1.2 Constant propagation & Local Rewriting

Constant propagation is also a common optimization in DSE engine solvers or compilers. While it is usually performed by the solver, it is interesting to make it during the path predicate computation as it simplifies data structures hold internally and also unleash the power of others optimizations that require a canonical representation of terms. Figure 7.2 shows a trivial example of constant folding on the expression `esp2`. Propagation is being performed by classic rewriting rules which the more important are given in Figure 7.3. We define by $\Downarrow$ the concrete evaluation operator, that simulate an expression according to its semantic or fail otherwise. Thus, this operator works only on constant variables.

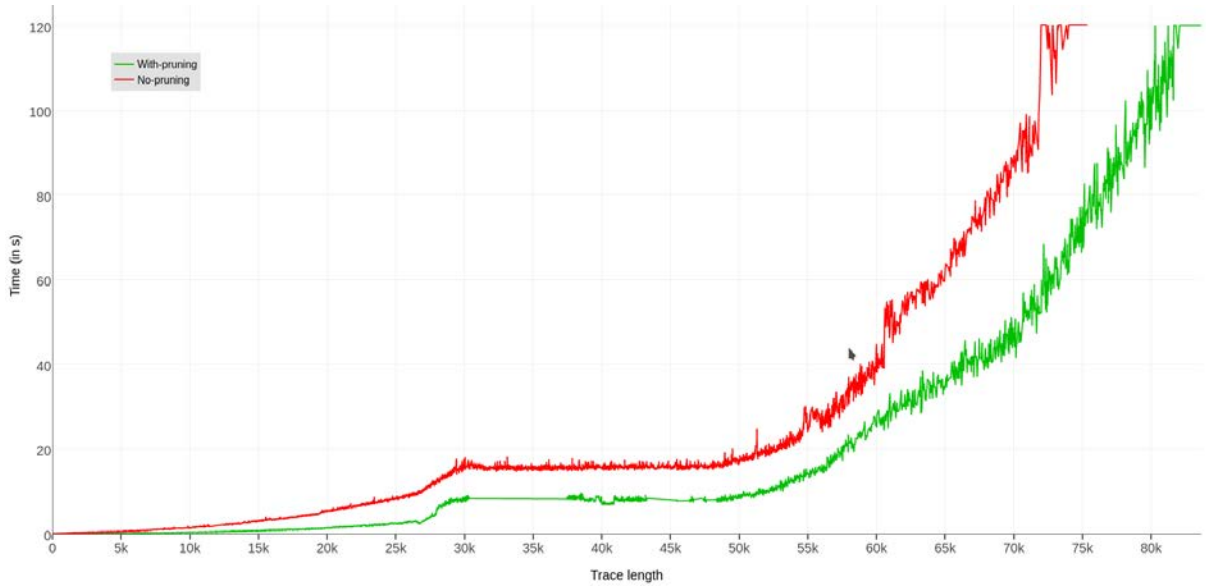


Figure 7.1: Solving time `mathsat` with/without pruning

Before constant folding		After constant folding
esp1 := 0xfffffff0	→	esp1 := 0xfffffff0
esp2 := esp1 + 0x00000004		esp2 := 0xffffffec

Figure 7.2: Constant propagation example

$$\frac{e_1, e_2 \in Bv \qquad e_1 \Diamond_b e_2 \Downarrow e'}{e_1 \Diamond_b e_2 \quad \rightsquigarrow \quad e'} \qquad \frac{e_1 \in Bv \qquad \Diamond_u e_1 \Downarrow e'}{\Diamond_u e_1 \quad \rightsquigarrow \quad e'}$$

$$\frac{e' \triangleq \begin{cases} e_1 & if\ c = true \\ e_2 & if\ c = false \\ if\ c\ ?\ e_1 :\ e_2 & otherwise \end{cases}}{if\ c ? e_1 :\ e_2 \quad \rightsquigarrow \quad e'} \qquad \frac{e_1 = e_2}{e_1 \ \| \ e_2 \quad \rightsquigarrow \quad e_1}$$

$$\frac{e_1 = e_2 \qquad sz \triangleq size(e_1)}{e_1 \ \oplus \ e_2 \quad \rightsquigarrow \quad 0^{\{sz\}}} \qquad \frac{e_1 = true \quad \vee \quad e_2 = true}{e_1 \ \| \ e_2 \quad \rightsquigarrow \quad true}$$

$$\frac{e_1 \Downarrow false \qquad e_2 \Downarrow false}{e_1 \ \| \ e_2 \quad \rightsquigarrow \quad false} \qquad \frac{e_1 \Downarrow false \quad \vee \quad e_2 \Downarrow false}{e_1 \ \&\& \ e_2 \quad \rightsquigarrow \quad false}$$

$$\frac{e_1 \Downarrow true \qquad e_2 \Downarrow true}{e_1 \ \&\& \ e_2 \quad \rightsquigarrow \quad true} \qquad \frac{e_1 = e_2 \qquad sz \triangleq size(e_1)}{e_1 - e_2 \quad \rightsquigarrow \quad 0^{\{sz\}}} \qquad \frac{e_2 \Downarrow 0 \qquad sz \triangleq size(e_1)}{e_1 \times e_2 \quad \rightsquigarrow \quad 0^{\{sz\}}}$$

$$\frac{e_2 \Downarrow 0}{e_1 + e_2 \quad \rightsquigarrow \quad e_1} \qquad \frac{e_2 \Downarrow 1}{e_1 \times e_2 \quad \rightsquigarrow \quad e_1}$$

$$\frac{e_1 \in Var \qquad e_2 \in Bv \qquad e_1' \triangleq \Sigma^*(e_1) \qquad e_1' \in Bv \qquad e_1' \neq e_2}{\Sigma^*, e_1 = e_2 \quad \rightsquigarrow \quad false}$$

$\Downarrow$ the concrete evaluation operator

Figure 7.3: Some propagation rules & rewriting

7.1.3 Rebase

The rebase optimization is a new auxiliary optimization aiming at reducing the number of variable definitions in the path predicate so as to improve the efficiency of both the *backward-pruning* pass, and the *read-over-write* optimization described in Section 7.2 by providing a common basis of logical comparison. This optimization works by replacing a new variable definition with an older definition by commiting all the arithmetic changes performed on it. It works for basic arithmetic operators $\{+, -\}$. Example in figure 7.4 shows the two advantages of applying such transformation combined with the backward-pruning pass. In the example `esp3` can be rebased on `esp1` which makes the `esp2` definition useless. Then the backward-pruning pass remove the term from the formula.

path predicate		After rebase (+pruning)
esp1 := eax	→	esp1 := eax
esp2 := esp1 - 0x04		~~esp2 := esp1 - 0x04~~
esp3 := esp2 - 0x04		esp3 := esp1 - 0x08

Figure 7.4: Constant propagation example

Figure 7.5 defines the two propagation rules used by the rebase optimization.

$$\frac{e \in Var \wedge \Sigma^*(e) \in Var \qquad e' \triangleq \Sigma^*(e)}{\Sigma^*, e \quad \rightsquigarrow \quad e'}$$

$$\frac{e_0 \ \Diamond_b' \ e_0' \triangleq \Sigma^*(e1) \qquad e_1 = e_0 \in Var \qquad e_0', e_2 \in Bv \qquad (\Diamond_b' \ e_0') + (\Diamond_b \ e_2) \Downarrow C_{new}}{\Sigma^*, e_1 \ \Diamond_b \ e_2 \quad \rightsquigarrow \quad e_0 + C_{new}}$$

Figure 7.5: Rebase propagation rules

Impact In addition to reducing the number of terms in the formula, this optimization also lays some basis for *making logical comparisons more easier*. For instance, in the example it is clear that `esp2` and `esp3` are not aliased (strictly disjoint) because they are both based on `esp1` but with different offsets. This property will be of a great help for the next section.

7.1.4 High level predicate encoding

The semantic of assembly instructions tends to be modeled with low level operations using for instance bit-masks, `xor` or `extract`. This is especially the case for conditions. At the source level the condition $if(x = 0)$, while certainly be assembled with both a

cmp and a jz instruction that will communicate with flags update. Table 7.3 shows an example of binary semantic for the simple $ecx < 24$ predicate.

Table 7.3: $ecx < 24$ binary semantic

Mnemonic	DBA semantic
cmp ecx, 24	$res32 := ecx - 24$ $OF := (ecx^{\{31\}} \neq 24^{\{31\}})\&\&(ecx^{\{31\}} \neq res32^{\{31\}})$ $SF := res32 <_s 0$ [...]
jl 0xfffffec	$if(SF \neq OF)$ goto 0xfffffec else [...]

$$((ecx - 24) <_s 0) \neq (ecx^{\{31\}} \neq 24^{\{31\}})\&\&(ecx^{\{31\}} \neq res32^{\{31\}})$$

Based on some recent researches [DB16b], the idea is to lift low-level operators to higher-level operators that once symbolically executed will be hopefully simpler to solve. Indeed, it depends on the inner working of the solver, some might be more efficient on low-level encoding while some others might be better on higher-level operators. Based on Table 7.4 we perform a syntactic processing on instructions to transform low-level predicates to their natural condition equivalent. Then the symbolic execution is performed as usual. This optimization is highly pervasive as it will modify all the path constraints in the formula. The transformation is *sound* as it only replaces a `cmp, sub, test` immediatly followed by conditional instruction based on pattern matching rules. Doing a generic transformation would require more in-depth and expensive formal analysis [DBG16]. Also, this optimizations is pattern-matching based so it works solely on `x86` and would require different rules to be adapted to other architecures.

Experiments Using the same execution trace (377000 instrs), the benchmark highlight the computation time with and without the high-level predicate optimization. Experiments carried put in evidence a small gain for certain solvers. Figure 7.6 shows the results for `Yices`. While it fluctuates the high-level predicate encoding is on average more efficient than without optimization (58% of the time). As a conclusion, the gain is rather small but such pre-processing worth being activated if the solver used takes advantage of higher-level predicates encoding.

7.2 Array optimizations (load/store)

The theory of arrays is used in DSE to represent the memory. Therefore, it is important to handle it carefuly and to optimize operations performed on it, they are used to be highly constrainted. We consider three types: arrays A, index I and elements E in any theory (in our case bitvectors). As a recall, the array theory [McC62] $\mathcal{T}_A$ provides two

Table 7.4: High-level predicates mapping [BR10]

Comparison instruction	Conditional instruction	High-level predicate
cmp x, y	ja, jnbe	$a >_u b$
	jne, jnb, jnc	$a \geq_u b$
	jb, jnae, jc	$a <_u b$
	jbe, jna	$a \leq_u b$
	je, jz	$a = b$
	jne, jnz	$a \neq b$
	jg, jnle	$a >_s b$
	jge, jnl	$a \geq_s b$
	jl, jnge	$a <_s b$
	jle, jng	$a \leq_s b$
sub x, y	ja, jnbe, jb, jnae, jc	$x' \neq 0$
	jae, jnb, jnc, jbe, jna, jge, jnl, jle, jng	$true$
	je, jz	$x' = 0$
	jg, jnle	$x' >_s 0$
	jl, jnge	$x' <_s 0$
test x, y	ja, jnbe, jne, jnz	$(a\&\&b) \neq 0$
	jae, jnb, jnc	$true$
	jb, jnae, jc	$false$
	jbe, jna, je, jz	$(a\&\&b) = 0$
	jg, jnle	$((a\&\&b) \neq 0) \wedge ((a\&\&b) \geq_s 0)$
	jge, jnl	$(a \geq_s 0) \vee (y \geq_s 0)$
	jl, jnge	$(a <_s 0) \wedge (y <_s 0)$
	jle, jng	$((a\&\&b) = 0) \wedge ((a <_s 0) \wedge (y <_s 0))$

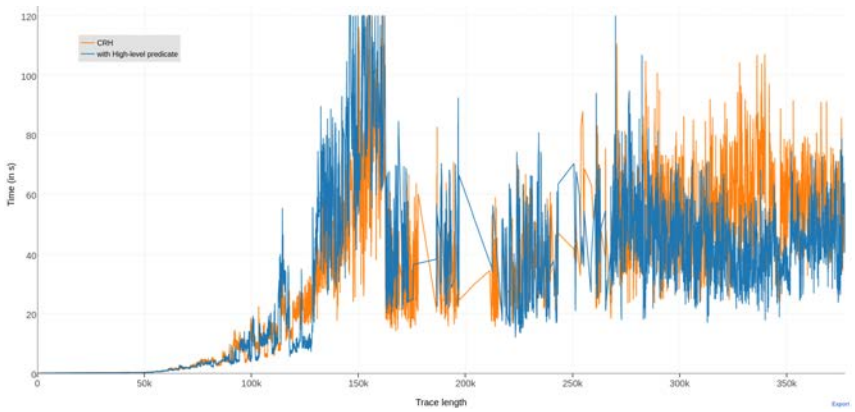


Figure 7.6: `Yices` solving time w.r.t trace length

operations $\Sigma_A = \{select, store\}$. also named *load* and *store* or *read*, *write*. From now we indifferently use them equally. The signature of these two operations are defined by:

- $read : A \times I \to E$
- $store : A \times I \times E \to A$

So, while $read(A, x)$ returns the logical value at index x in the array A, $write(A, x, v)$ returns a new array updated with the value v stored at x. The first property of array is the *functionality* (cf. **FC**) which states that given an array a, if two indexes i, j are equals the *read* operation on whichever index are equals. Given the signature of the *store*, a memory is either an empty memory or the outcome of a sequence of store. The read-over-write (RoW) is instinctively knowing whether a *read* index has previously been written or not. For a given memory, this can be done by iterating all its store operations. The Read-over-Write axiom consider two cases:

$$\textbf{FC} : \text{if } i = j, read(a, i) = read(a, j) \tag{7.1}$$
$$\textbf{RW1} : \text{if } i = j, read(write(a, i, v), j) = v \tag{7.2}$$
$$\textbf{RW2} : \text{if } i \neq j, read(write(a, i, v), j) = read(a, j) \tag{7.3}$$

Intuitively, the two cases can be described as:

- **RW1** we can replace a *read* operation by the value written on the previous *write* if it is performed on the same logical indexes,
- **RW2** if both indexes are different, we can replace the *read* in the result of the *store* by a *read* in a (directly).

Backwarding simplification These two properties can be applied in order to perform simplification on arrays. The idea is to perform such simplification on every *load* of the execution to either substitute it by the written value (Figure 7.7) or to "rebase" it to an older memory (Figure 7.8). The later case works similarly to the *rebase* optimization. For a given *load* the simplification algorithm needs to work recursively as long as it can compare the load and store indices.

Before Read-over-Write		After Read-over-Write
memory1 := write(memory0, 0xffc351fc, 0x04) eax := read(memory1, 0xffc351fc)	→	memory1 := write(memory0, 0xffc351fcn 0x04) eax := 0x04

Figure 7.7: RoW case #1

Before Read-over-Write		After Read-over-Write
memory1 := write(memory0, 0xffc351fc, 0x04)	→	memory1 := write(memory0, 0xffc351fc, 0x04)
memory2 := write(memory1, 0x80105200, 0xfc)		memory2 := write(memory1, 0x80105200, 0xfc)
eax := read(memory2, 0x7f000080)		eax := read(memory0, 0x7f000080)

Figure 7.8: RoW case #2

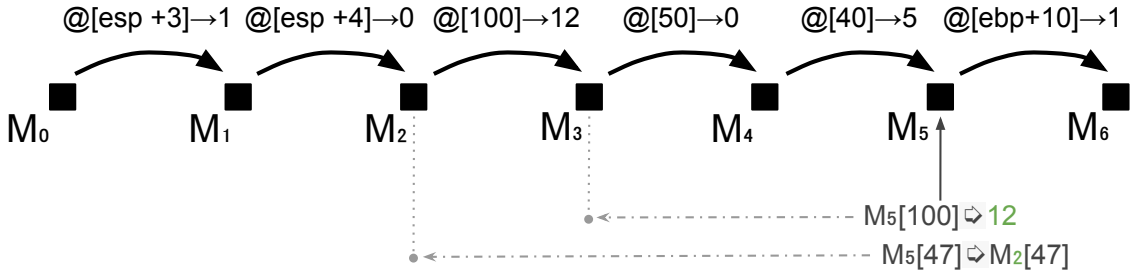


Figure 7.9: Example STC-RoW

7.2.1 Standard Read-over-Write pre-processing

To simplify a *load* operation a *backtracking* step on the whole store chain must be performed. The "standard" Read-over-Write is a linear approach where the RoW complexity for a single *load* is linear $\mathcal{O}(n)$ with n the number of *store* operations in the path predicate. Depending on the number of *load* the worst case complexity of the whole simplification process is roughly quadratic. Hence the backtracking phase along the store-chain is usually bounded by a constant k to ensure only a linear overhead. Figure 7.9 shows an example of such chain of *store*. The first load $M_5[100]$ can be fully substituted with a RoW backtracking 2 times. The second load $M_5[47]$ can be rebased on M_2 by backtracking 3 times. It cannot be backtracked further as 47 and $esp + 4$ cannot be logically compared.

Formally, we denote $St(m, i, x)$ a store operation in the memory m at index i and the store-chain $\overrightarrow{St} \triangleq < St_0, St_1 \dots St_n >$. Then we consider $3 - cmp$ a three-way comparison function allowing to compare expressions. This function should return *true* and respectively $false$ if both expressions are comparable and respectively equal or not. If two expressions are not comparable the function should return ?. More formally we have:

$$\text{3-cmp}(x, y) \begin{cases} true & \text{if } comparable(x, y) \wedge x = y \\ false & \text{if } comparable(x, y) \wedge x \neq y \\ ? & \text{if } \neg comparable(x, y) \end{cases}$$

Implementation 3-cmp *comparable* returns true only if both expression use the same logic variables (in this case the bitvector theory $\mathcal{T}_{Bv}$). As an example $3 - cmp(esp_1 - 4, esp_1 - 4) = true$, $3 - cmp(esp_1 - 4, esp_1 - 8) = false$ and $3 - cmp(esp_1, eax_0) =?$ as esp_1 and eax_0 cannot be syntactically compared. We consider by convenience that memories

(aka array) are named strictely incrementally, thus M_{x+1} is the memory created after a store in M_x.

Backtracking algorithm We can now formalize the RoW backtracking algorithm as a recursive function iterating the store-chain bounded by k. We denote by STC-RoW the unbounded algorithm and kSTC-RoW the algorithm bounded by k. The overall kSTC-RoW complexity is $\mathcal{O}(\#load \times k)$ thus bounded by $\mathcal{O}(n^2)$ and the algorithm is defined by:

$$\texttt{kSTC-RoW}(load(M_n, e_{ld}), k) \triangleq \begin{cases} v & if\ C = true \wedge k \neq 0 \\ \texttt{kSTC-RoW}(load(M_{n-1}, e_{ld}), k-1) & if\ C = false \wedge k \neq 0 \\ load(M_n, e_{ld}) & if\ C = ? \vee k = 0 \end{cases}$$

$$\text{with: } M_{n-1}, e_{st}, v \triangleq \mathbb{S}ts.(n-1) \text{ and } C \triangleq \text{3-cmp}(e_{ld}, e_{st})$$

7.2.2 Read-over-Write "Concrete map"

In the case of constant indexes (eg. policy CC) we can in principle remove all *load*. Yet, kSTC-RoW cannot achieve it. A common solution is using a map of all indices written, then we can perform in $\mathcal{O}(ln(n))$ the backtracking step of RoW. This provides the best complexity possible for the RoW simplification. We denote CM-RoW this algorithm. Under these settings, all *load* can either be replaced by the value previously written either being rebased on the initial memory. As a consequence, the chain of *stores* is not used and can be pruned. Unfortunately, having constant values for indices is a really specific situation and in DSE, indices are more often arbitrary logical expression on bitevctors.

7.2.3 Read-over-Write "Store-Map Chain"

The main issues of the previous approaches are:

- STC-RoW does not scale on huge path length we have to deal with,
- kSTC-RoW is not optimal,
- CM-RoW only applies under very specific settings.

We propose a more scalable and optimal algorithm which complexity is oscillating between $\mathcal{O}(ln(n))$ and $\mathcal{O}(n^2)$ depending on the settings and environment. We can use a different internal data-structure to improve the complexity of the RoW. The idea behind this encoding is that multiple stores having the same base will update the same map thus reducing the cost of lookup when performing the RoW on a *load*. If a new store is performed but with a different base, a new map (with its base) is appended to the chain. Figure 7.10 shows the same store chain as the standard approach (see Fig.7.9) but using our data-structure. For one *load*, the STC-RoW complexity is $\mathcal{O}(n)$ while it is $y \times log(x)$ for the STMC-RoW where y is the number of store-groups and x the size of the maps. In the worst case scenario we have $y = n$ and thus $x = 1$ bringing us back to the same complexity as STC-RoW.

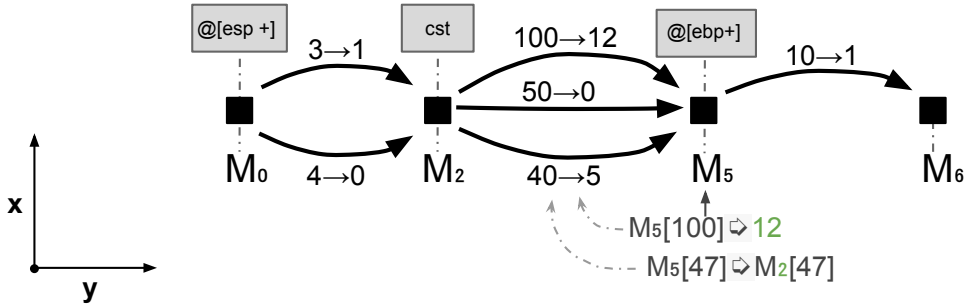


Figure 7.10: Architecture STMC-RoW

Formalization Similarly to the standard RoW we defines by $\mathbb{S}ts^\sharp \triangleq < St_1^\sharp, St_2^\sharp \dots St_n^\sharp >$, the new store chain where $St_n^\sharp$ is an element of the chain. Thus, we defines $St_n^\sharp \triangleq (M_n, b_n, map_n)$ the tuple where $M_n \in \mathcal{T}_A$ is the memory, $b_n \in Base$ a base expression and $map_n \in (\mathbb{N} \times \mathbb{E}xpr)$. We denote $Base \triangleq \{\mathbb{E}xpr \cup Csts\})$ formed of either an expression either a constant value. Based on these definitions we define b_of_expr : $\mathbb{E}xpr \rightarrow (Base \times \mathbb{N}$ an auxiliary function that returns the base and offset of any expression. For instance b_of_expr$(esp + 4) = esp, 4$ or b_of_expr$(43) = (Csts, 43)$. The update function and the STMC-RoW algorithms are formalized below:

$$update - store(M_{n+1}, e_{st}, v) \triangleq \begin{cases} map_n[\text{off}_{st}] := v & if\ \neg(\text{3-cmp}(b_n, b_{st}) \vDash ?) \\ \mathbb{S}ts^\sharp :: (M_{n+1}, b_{st}, (\text{off}_{st} \mapsto v)) & if\ \text{3-cmp}(b_n, b_{st}) \vDash ? \end{cases}$$

$$\text{with: } b_n, map_n \triangleq \mathbb{S}ts^\sharp.(n) \text{ and } b_{st}, \text{off}_{st} \triangleq \text{b\_of\_expr}(e_{st})$$

$$STMC{-}RoW(load(M_n, e_{ld}) \triangleq \begin{cases} map_N.(\text{off}_{ld}) & if\ \text{3-cmp}(b_{ld}, b_n) \vDash true \wedge \text{off}_{ld} \in dom(map_n) \\ load(M_{n-1}, e_{ld}) & if\ \text{3-cmp}(b_{ld}, b_n) \vDash false \\ load(M_n, e_{ld}) & otherwise \end{cases}$$

$$with : b_{ld}, \text{off}_{ld} = \text{b\_of\_expr}(e_{ld}) \text{ and } M_n, b_n, map_n = \mathbb{S}ts^\sharp.(n)$$

Such a construction enhances significantly the complexity of the RoW algorithm. Table 7.5 summarizes the complexity gain between the different alternatives. The best construct is to keep a concrete mapping from constant addresses to theirs value but it only works if all read/write are performed to constant addresses. So, while with constant indices STMC-RoW have the best complexity (the same as a direct mapping CM), it also gain on more symbolic modes knowing that in the worst case it provides the same complexity as kSTC-RoW. As a consequence this algorithm adapts and scales to depending on the sort of *loads* and *store* performed and provides the optimal complexity in all cases. We denote respectively *perfect* techniques that perform the optimal substitution and we denote them *partial* instead.

This algorithm works especially well on C/S policies that keep concrete memory addresses (cf. Table 7.5) in that it will always be possible to backtrack to the first memory thus opening the way to syntactical optimizations as the *memory flattening* presented hereafter.

Table 7.5: Complexity comparison of RoW simplifications

index	concrete map CM-RoW	store-chain STC-RoW	k-store-chain kSTC-RoW	store-map chain STMC-RoW
constant values	$n \cdot ln(m)$ (perfect) +WoW	$n.m$ (perfect)	$n \cdot k$ (partial)	$n \cdot ln(m)$ (perfect) +WoW
symbolic	(cannot handle)	$n.m$ (perfect)	$n \cdot k$ (partial)	$n \cdot y \cdot ln(x)$ (partial) +WoW

. k backward bound, . x maximum number of value within a map, . n number of load, . m number of store, . y number of map, . (partial) do not simply all possible *load*, . $perfect$ simplify all possible *load*, . WoW Write-Over-Write gained wihtout additional computation (writing over another write on the same index)

Write-over-Write The WoW is the property indicating that two consecutive *store*, at the same logical index are equivalent to the second store in the array of the first. Formally:

$$store(store(A_n, i, v_1), j, v_2) \wedge i = j \implies store(A_n, j, v_2)$$

Performing a WoW simplification implies to backtrack on every *store* in memory which is very costly. Moreover, such simplifications should also handle potential *load* operation performed on the memory of a *store* before simplifying it. As such, STMC-RoW with its map-based chain provides the Write-over-Write (WoW) for free. Indeed, two store using the same base, and writing at the same address will update the same value in the map (overwriting the previous one).

7.2.4 Benchmarks

The following two benchmarks emphasize the difference and the effect of the kSTC-RoW and STMC-RoW techniques on a generated formula w.r.t compactness and efficiency of the optimization. For that a 370000 instructions long trace was used. For the kSTC a bound of $k = 150$ was arbitrarily used. The first benchmarks shows formula simplification statistics by in applying kSTC-RoW or STMC-RoW. The second benchmark, provides some insight on the internal statistics of optimizations themselves like the chain lengths, map sizes etc.

Benchmark #1 This benchmark provides formula statistics namely number of inputs, variables, load and store in memory and the number of constraints. These attributes are compared without RoW and with kSTC-RoW or STMC-RoW under different levels of concretization on the memory. The idea is to assess the potency of optimizations on formulas addressing the memory accesses differently. For that, we re-use policies throughoutly discussed in Chapter 4, namely CC, PC, PP* and PP. For clarity in the results the policy CP is not shown as results are very close from PP* and PP. Results are given in Table 7.6.

Table 7.6: Simplification statistics kSTC Vs STMC RoW

	CC			PC			PP*			PP		
	∅	kSTC	STMC	∅	kSTC	STMC	∅	kSTC	STMC	∅	kSTC	STMC
inputs	119	96	61	119	114	114	119	85	85	119	116	116
variables	**482357**	335926	**155593**	482357	353234	353234	482357	354922	353074	482357	340340	331898
load	**107574**	52353	**3381**	107574	109755	109744	107574	100087	99203	107574	104942	100956
store	**58005**	57855	**0**	58005	57997	57997	58005	57992	57992	58005	58005	58005
constraints	218778	196691	84262	115357	110269	110269	150928	108131	108126	57361	52280	51217

Results for CC are outstanding. While kSTC-RoW removes half of *loads*, STMC-RoW substitute all of them. The 3381 remaining are *loads* on the initial memory and thus allow to prune the whole *store* chain. This shows that kSTC-RoW is not optimal and could have replaced 48972 more *loads*. On more symbolic modes and especially SS kSTC-RoW managed to replace 2632 *loads* by their value while STMC-RoW managed to replace 6618 of them, thus more than twice more. It worths noting that these statistics does shows *load* which where rebased to preceding memories. These data are shown in the next benchmark. Timing benchmarks are given in Table 7.8 and also validates the gain of STMC over the STC approach.

Benchmark #2 The goal of this benchmark is to highlight the inner efficiency of kSTC and STMC optimizations. It shows the number of *load* fully *replaced* (**ROW 1**), the number of *load rebased* (**ROW 2**) on a prior memory and the number of *load* where nothing could have been performed (∅). It also shows for kSTC the min, max, moy of the backtracking k (so bounded by 150). For STMC-RoW it takes another meaning, it represents min, max, mean the size of the different maps in the store chain. Finally it also shows for STMC-RoW the y value representing the size of the custom store chain. The benchmark re-use the same settings as the first benchmark and results are given in Table 7.7.

Table 7.7: Internal optimization statistics kSTC Vs STMC RoW

		CC		PC		PP*		PP	
		kSTC	STMC	kSTC	STMC	kSTC	STMC	kSTC	STMC
replace		73209	**129644**	4	28	10139	10936	4960	**8959**
rebase		63185	2	25	0	111	1091	0	**1105**
∅		17	4243	138672	138673	128099	140637	133629	141174
kSTC: **k**	min	0	36	0	36	0	1	0	1
STMC: **x**	max	150	15506	115	15506	149	757	7	**264**
	avg	5	7506	0	7488	0	15	0	**12**
STMC:**y**		/	1	/	1	/	20581	/	24770

Results are insightful to understand the potency of algorithms. In STMC-RoW in CC *load* but two are replaced by theirs value. The two remaining have been rebase on the initial memory while the others were already *load* on initial memory. On average kSTC

backtrack on five *store* to perform the replacement which means many of RoW are rather local. As expected in CC the store-chain length is 1 so all the RoW simplification where performed in $ln(x)$. For more symbolic modes, optimizations are still working despite the reasoning difficulty. For instance, in PP, STMC manages to replace 8959 *load* and to rebase 1105 of them but more surprinsingly the maximum size of the map is 264 with an average of 12. This results shows that storing multiple *store* with the same base is working and efficient in practice even one more symbolic modes. This benchmark highlight another takeway on PP. In STMC, the store-chain is reduced to 24770 while the whole store chain is still 58005 and kSTC have to backtrack on such size.

Conclusion Results are clear, while kSTC slightly reduce the number of load and the number of variables compared to no optimization, STMC significantly peforms better and especially on CC (as expected). Indeed, for STMC in CC the store chain is only one map containing all the element. Notably, PC is particulary weak as only 28 load were replaced. This indicates that keeping either load or store symbolic tends to weaken the optimization. Further, solving time benchmarks are performed in the next section. As a conclusion, STMC performance in CC are optimal as expected. Yet, in more symbolic modes the STMC-RoW is still potent as it managed to replace 8959 load and to rebase 1105 of them which is twice better than a standard RoW. It is worth noting that in this mode the average map was 12 with a maximum value of 264. These results validates the potency of the optimization at any level of "symbolicness".

7.2.5 Memory flattening

This syntactical optimization is complementary with the RoW and aims at removing the memory array when all the *load* and *store* are performed at concrete indexes. This is made possible by the RoW that will either replace a *load* by its value either rebase it in the initial memory denoted M_0. Then, all *load* replaced by a $fresh$ bitvector representing the memory cell. Removing all the *load* operations turn a Qantifier-Free Array and Bitvectors (QF_ABV) formula into Qantifier-Free Bitvectors (QF_BV). In this way, the formula can be solved by solvers not supporting the array theory $\mathcal{T}_A$. Jointly, it helps solvers like `Z3` which are more efficient on bitvector-only formulas. For this optimization, we consider M_0 the initial memory, and $map \in (Bv \to \varphi)$ the map from constant addresses to theirs symbolic value. The optimization algorithm is formalized as:

$$\text{mem\_flat}(map, M_i, e) \triangleq \begin{cases} map, map(e) & if\ e \in dom(map) \wedge M_i = M_0 \wedge e \in Bv \\ map[e \leftarrow fresh], fresh & if\ e \notin dom(map) \wedge M_i = M_0 \wedge e \in Bv \\ map, load(M_i, e) & otherwise \end{cases}$$

The first output is the map (update or not), that will be held during the whole symbolic execution. The second output is the expression simplified (or not). Let us detail the three cases: (1) the index is already in the map so we substitute the expression by the content of the map, (2) the index is not in the map so return a fresh symbol and update accordingly and (3) the memory where the load is performed is not the initial memory so nothing is performed.

Experiments To evaluate the gain we solve the previously used path predicate of 377300 instructions containing 3381 read in memory. We performed the experiments in CC-CRH so that the optimization will remove all the memory array. Figure 7.11 shows the solving time results for Z3 that really benefits from the flattening. While the solving time did not appeared to change significantly on other solvers, it was never worse. Thus, it is worth activating this optimization that will opportunistically remove the array theory on concrete modes like CC-CRH.

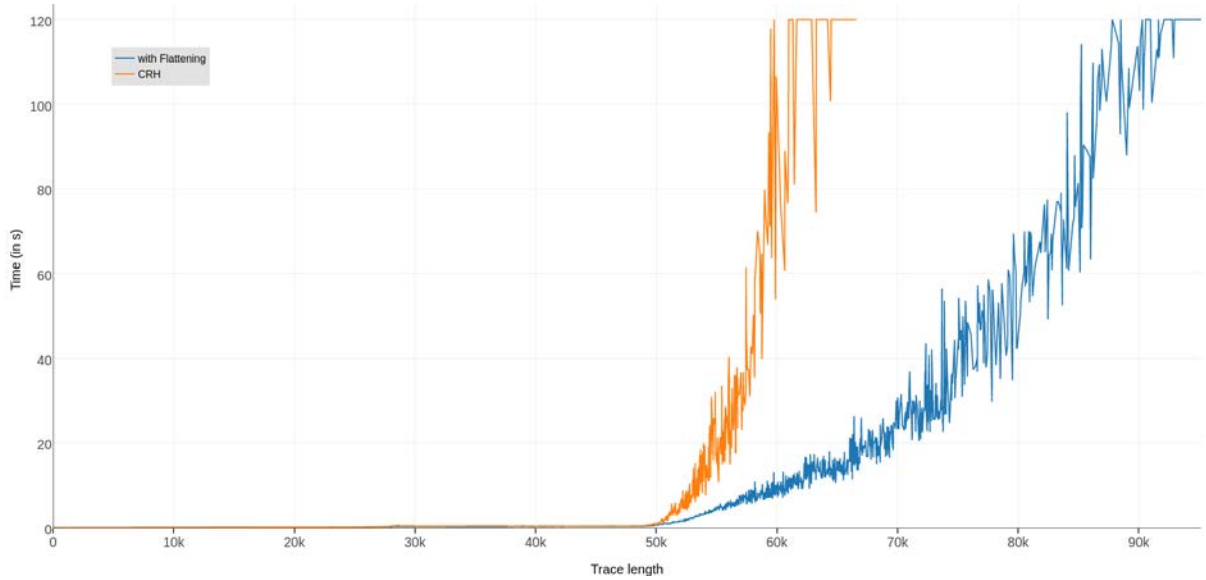


Figure 7.11: Z3 solving time w.r.t trace length

7.3 Experiments: Optimization combination

All experiments are complementary one another. This section evaluates the gain of the different RoW modes on the criteria of formula size and solving time. All that is made in different environment of execution namely different concretization and symbolization settings. Benchmarks are performed on a 377300 instructions long trace. Three benchmarks will be performed. **Bck#1** evaluates the computation time impact of the optimisations on the path predicate. **Bck#2** evaluates the impact of optimizations w.r.t C/S policies and the impact on the path predicate. **Bck#3** evaluates the different solvers w.r.t optimizations given a fixed C/S policy).We notate "C" the constant propagation, "R" the rebase, "W" the kSTC-RoW and "H" the STMC-RoW. Thus, the code CRH, indicates the combination of constant propagation, rebase and STMC performed in this order. As expected, basic pre-processing depends on the efficiency of the solver, while array optimizations demonstrate a huge speed-up for all solvers.

7.3.1 Benchmark #1: Impact on path predicate computation

Table 7.8 shows the benchmark results for five different C/S policies (see. Chapter 4) and five different levels of optimizations. Results show that not performing optimization increases the instruction throughput which tends to discourage using these optimizations. However, they drastically decreases the solving time as shown in **Bck#3**. As a conclusion, we might be interested in finding the right balance in selecting the optimizations although it will everytime be in favor of using all optimizations.

Table 7.8: Path predicate computation time and instruction/seconds

	CC	CP	PC	PP*	PP	*avg*
∅	22.09s 17081 i/s	22.04s 17119 i/s	17.69s 21329 i/s	18.56s 20329 i/s	16.90s 22325 i/s	19.46s 19392 i/s
C	27.81s 13567 i/s	27.38s 13780 i/s	23.89s 15794 i/s	24.72s 15263 i/s	22.64s 16665 i/s	25.29s 14920 i/s
CR	29.41s 12829 i/s	28.60s 13193 i/s	25.14s 15008 i/s	25.78s 14636 i/s	24.28s 15540 i/s	26.64s 14162 i/s
CRW	79.32s 4756 i/s	30.67s 12302 i/s	26.46s 14260 i/s	26.61s 14179 i/s	25.91s 14562 i/s	37.79s 9983 i/s
CRH	40.84s 9238 i/s	30.17s 12506 i/s	37.34s 10105 i/s	26.97s 13990 i/s	26.29s 14352 i/s	32.32s 11673 i/s
avg	39.89s 9458 i/s	27.77s 13586 i/s	26.10s 14454 i/s	24.53s 15383 i/s	23.20s 16261 i/s	

performed on a 377000 instructions path predicate

7.3.2 Benchmark #2: Optimizations

This benchmark assesses the potency of optimizations on different C/S policies in terms of formula size and statistics. Table 7.9 shows the statistics for each level of optimizations.

Unsurprisingly, the CRH optim in CC manages to remove all the *store* operations and drastically reduce the *load* operations to the ones reading in the initial memory. The gain for loads is x31. The STMC is by design less potent on more symbolic policies. However optimisations provide good constraints reduction, for instance in PP* constraints are reduced from 150928 to 108126 between no optimisations and CRH, thus providing a gain of 28%. As conclusion, from the statistical point of view, the combination CC-CRH removes all stores, and almost all load but more symbolic modes also provides significant improvments on the solving time as shown in **Bck#3**.

7.3.3 Benchmark #3: Solving time

With a fixed, policy CC this benchmark evaluates the solving time of a path predicate under different level of optimisations. As performed in previous Chapter 4 we evaluate

Table 7.9: Optimizations formula statistics

		#inputs	#variables	#load	#store	#constraints
CC	∅	119	482,357	107,574	58,005	218,778
	C	116	448,919	107,478	58,005	213,602
	CR	116	350,173	106,596	58,005	207,204
	CRH	61	155,593	3,381	0	84,262
CP	∅	117	482,357	107,574	58,005	160,782
	C	114	448,904	107,478	58,005	155,607
	CR	114	339,972	106,596	58,005	149,209
	CRH	114	334,344	106,585	58,005	149,204
PC	∅	119	482,357	107,574	58,005	115,357
	C	116	444,082	107,478	58,005	110,277
	CR	116	353,257	109,759	58,005	110,277
	CRH	114	353,234	109,744	57,997	110,269
PP*	∅	119	482,357	107,574	58,005	150,928
	C	86	438,605	107,465	58,005	112,124
	CR	86	357,549	109,651	58,005	111,092
	CRH	85	353,074	99,203	57,992	108,126
PP	∅	119	482,357	107,574	58,005	57,361
	C	116	444,070	107,478	58,005	52,282
	CR	116	340,396	109,759	58,005	52,282
	CRH	116	331,898	100,956	58,005	51,217

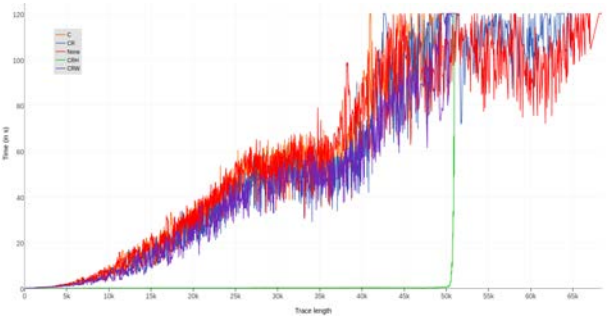
the solving time of different path predicate lengths. We evaluate 5 solvers under these settings with a 2 minutes timeout (TO=120s).

Figure 7.12 shows the solving time results obtained and the gain provided in terms of performences. Results shows, that without optimisations none of the solvers succeed in solving the whole execution trace. In CRH `Yices` manages to solve the whole predicate. The difference between no optimisation, just constant propagation and rebase is rather low as they are more pre-processing tasks aiming at improving the RoW. Results put in evidence the gap between CRW and CRH. `CVC4`, `mathsat` and `Yices` are the best examples.

While the performances of solvers are very disparate, optimizations provide better results than no optimizations and most notably with the CRH. Results show that CRW misses lots of cases and is as consequence not optimal.

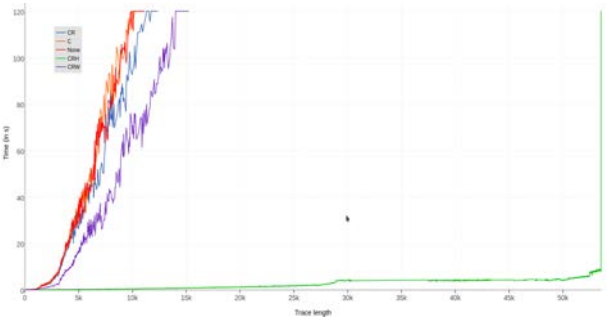
boolector	∅	C	CR	CRW	CRH
500	0.05	0.05	0.06	0.09	0.02
1000	0.08	0.12	0.15	0.12	0.02
10,000	6.07	9.73	8.03	6.28	0.05
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	68450	51000	64842	51120	50970

(a) `boolector` solving: Time & Graph



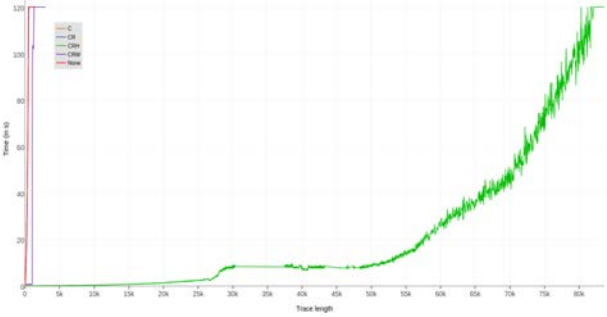
CVC4	∅	C	CR	CRW	CRH
500	0.13	0.16	0.14	0.08	0.03
1000	0.34	0.37	0.29	0.11	0.04
10,000	112.08	111.24	92.68	66.96	0.44
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	10057	10101	11640	13936	53540

(b) `CVC4` solving: Time & Graph



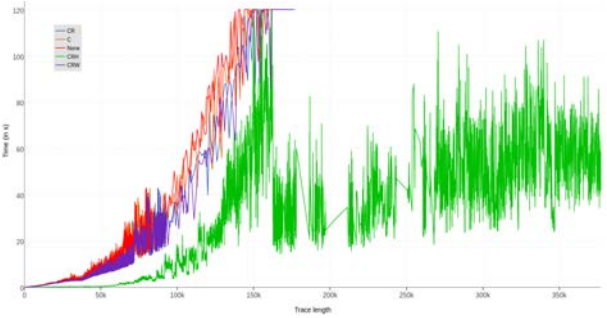
mathsat	∅	C	CR	CRW	CRH
500	X	115.91	X	0.72	0.03
1000	X	X	X	0.66	0.03
10,000	X	X	X	X	0.32
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	476	517	464	1236	81960

(c) `mathsat` solving: Time & Graph



Yices	∅	C	CR	CRW	CRH
500	0.02	0.02	0.02	0.02	0.02
1000	0.05	0.05	0.04	0.04	0.02
10,000	0.8	0.75	0.63	0.52	0.06
100,000	34.75	48.62	33.42	34.77	10.7
377309	X	X	X	X	54.37
max	155000	159000	159000	161000	377309

(d) `Yices` solving: Time & Graph



Z3	∅	C	CR	CRW	CRH
500	0.37	0.39	0.47	0.05	0.02
1000	0.86	0.89	1.13	0.05	0.02
10,000	75.50	74.94	64.75	86.92	0.07
100,000	X	X	X	X	X
377309	X	X	X	X	X
max	22320	21120	20760	15196	62540

(e) Z3 solving: Time & Graph

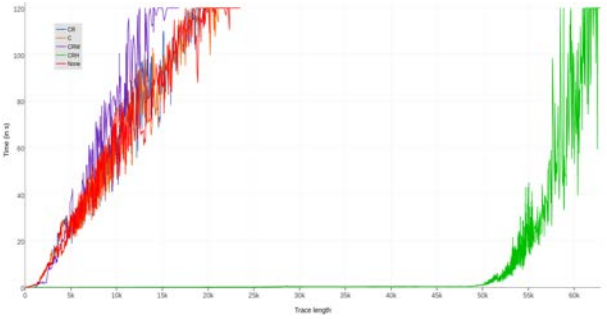


Figure 7.12: Optimizations benchmark by solvers

Part IV

Experimental Validation

Chapter 8

Deobfuscation

This chapter presents two case-study obfuscations that were used as applications of our algorithms in order to detect and thwart them. We considered opaque predicates and call/stack tampering that can be encoded as infeasibility queries. Moreover, these two obfuscations are rather resilient to static approaches which justifies focusing on them as they are commonly encountered. Other obfuscations, like self-modification, are indirectly handled by the dynamic aspect of our DSE approach. On opaque predicates, experiments will be performed on the whole *coreutils* obfuscated using O-LLVM. For call/stack tampering, experiments are performed on samples programs obfuscated with Tigress along with some packers. This Chapter lays the basic taxonomy and detection methodology used in Chapter 10 for real-world case-studies.

8.1 Opaque predicates detection

8.1.1 Definitions

An opaque predicate is a predicate always evaluating to true (resp. false) for which this property is ideally difficult to deduce. The infeasible branch will typically lead the reverser (or disassembler) to large and complex portions of useless junk code. The next section shows various concrete examples of such predicates. The first definition was given by Collberg[CTL98]:

Definition 2. *A variable V is* opaque *at point p in a program, if V has a property q at p which is known at obfuscation time written V_p^q. A predicate P is opaque at p if its outcome is known at obfuscation time. We write P_p^F / P_p^T if P always evaluates to* `False` *respectively* `True` *at p, and $P_p^?$ if P may sometimes evaluate to* `True` *and sometimes to* `False`*.*

8.1.2 Taxonomy & Example

From now we denotes opaque predicates by Opaque Predicates (OP) for concision. Collberg first proposed a taxonomy and classification of opaque predicates [CTL97] using

either arithmetic or data structure invariant properties to build opaque predicates. The different kinds of opaque predicates:

Arithmetic invariant aims at encoding invariants into arithmetic properties on numbers. An efficient method is to use modulo arithmetic. Further distinction between arithmetic opaque predicates is given below. As an example Figure 8.1 shows the x86 encoding of the $7y^2 - 1 \neq x^2$ predicate (as generated by O-LLVM [Jun+15]). This condition is always false for any values of `ds:x`, `ds:y`, so the conditional jump `jz <addr_trap>` is never going to be taken. Table shows many other OP used in various tools such as Sandmark [Col03].

```
mov   eax, ds:x
mov   ecx, ds:y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <addr_trap> //false jump to junk
....  ........    //real code
```

Listing 8.1: opaque predicate: $7y^2 - 1 \neq x^2$

Data-structure invariant such category aims at hiding invariant into data-structure, crafted on purpose which have specific opaque properties. Collberg proposed the example of an array given in listing 8.2 which hold the following properties: (1) all cells at indexes modulo 3 starting from 0 are 1 *mod* 5, (2) from index 1 all cells at indexes modulo 3 are 2 *mod* 7, (2) cells at indexes 2,5,8,11 are respectively 1,5, 2 and 7. Then `g[3] % g[5] == g[2]` is always true because g[3] holds the property 1 *mod* 5 and g[5],g[2] are statically 5 and 1.

```
void main() {
    //index:   0   1   2  3   4   5  6   7   8  9   10 11 12 13  14
        15  16  17 18  19
    int g[] = {36, 58, 1, 46, 23, 5, 16, 65, 2, 41, 2, 7, 1, 37, 0,
        11, 16, 2, 21, 16};

    g[14] = rand(); //manipulable because unconstrainted (no
       invariant)

    if(g[3] % g[5] == g[2]) {
        //Always get into
    }
}
```

Listing 8.2: Array invariant data-structure

Pointer based invariant Aliasing is known to be a hard problem. Thus, encoding invariant pointer aliasing on specific data-structure is difficult for any automated algorithm. Collberg suggested an example based on circular list. Let's consider

Table 8.1: Arithmetic opaque predicates list (non-exhaustive)

$\forall x, y \in \mathcal{I}$	$7y^2 - 1 \neq x^2$
$\forall x \in \mathcal{I}$	$2 \mid (x + x^2)$
$\forall x \in \mathcal{I}$	$2 \mid x(x + 1)$
$\forall x \in \mathcal{I}$	$3 \mid (x^3 - x)$
$\forall x \in \mathcal{I}$	$x^2 \geq 0$
$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}$	$(x - y) \mid (x^n - y^n)$
$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}$	$2 \mid n \vee (x + y) \mid (x^n + y^n)$
$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}$	$2 \nmid n \vee (x + y) \mid (x^n - y^n)$
$\forall x \in \mathcal{I}^+$	$9 \mid (10^x + 3 \cdot 4^{x+2} + 5)$
$\forall x \in \mathcal{I}$	$3 \mid (7x - 5) \Rightarrow 9 \mid (28x^2 - 13x - 5)$
$\forall x \in \mathcal{I}$	$5 \mid (2x - 1) \Rightarrow 25 \mid (14x^2 - 19x - 19)$
$\forall x, y, z \in \mathcal{I}$	$(2 \nmid x \wedge 2 \nmid y) \Rightarrow x^2 + y^2 \neq z^2$
$\forall x \in \mathcal{I}^+$	$14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$
$\forall x \in \mathcal{I}$	$2 \mid x \vee 8 \mid (x^2 - 1)$
$\forall x \in \mathcal{I}^+$	$64 \mid (7^{2x} + 16x - 1)$
$\forall x \in \mathcal{I}^+$	$24 \mid (2 \cdot 7^x + 3 \cdot 4^x - 5)$
$\forall x \in \mathcal{I}$	$\sum_{i=1,2 \mid i}^{2x+1} i = x^2$
$\forall x \in \mathcal{I}^+$	$8 \mid (7^{2x+1} + 17^x)$
$\forall x \in \mathcal{I}^+$	$2 \mid \lfloor \frac{x^2}{2} \rfloor$
$\forall x \in \mathcal{I}^+$	$3 \mid x(x + 1)(x + 2)$
$\forall x \in \mathcal{I}^+$	$7 \nmid x^2 + 1$
$\forall x \in \mathcal{I}^+$	$81 \nmid x^2 + x + 7$
$\forall x \in \mathcal{I}^+$	$19 \nmid 4x^2 + 4$
$\forall x \in \mathcal{I}^+$	$4 \mid x^2(x + 1)(x + 1)$

the example given in listing 8.3, `create_circular_list` creates a list of a random size and returns a pointer to it. Then `move` shifts the current pointer of a given number of items and `insert` add a new item in the list and returns a pointer to it. As a consequence `g` cannot alias with `h`. By moving both pointers in the data structure with the same value we ensure they cannot alias. This property si then used to create opaque predicates. This is an example but any other data-structure embedding such invariant are good candidates to create such predicates.

```
void main() {
        item * g = create_circular_list(rand());
        g = move(g, rand());
        item * h = insert(g);
        int x = rand();
        g = move(g, x);
        h = move(h, x);

        if (h == g) {
                //never taken
        }
}
```

Listing 8.3: Pointer aliasing invariant data-structure

Concurrence based invariant The idea is to encode invariant in race condition properties. Both static and dynamic analyses are known to be difficult and unreliable for proving properties on such concurrent codes. Thus, using such kind of OP is particularly efficient although difficult to craft reliably. Collberg proposed an example where two threads update global variables intentionally delayed the threads to encore opaque predicates. Listing 8.4 shows in example where two threads computes the variables `x` and `y` of an opaque predicates. The trick is that `x` is either `R*R` if `threadS` come seconds, either `R*R * R*R threadT` come second. In both cases the OP invariant will hold.

```
int X, Y, C;

void* threadS (void* name) {
        int R = rand()%C;
        sleep(rand()%10);
        X = R*R;
}

void* threadT(void* name) {
        int R = rand()%C;
        Y = 7*R*R;
        sleep(rand()%10);
        X *= X;
}

int main (void) {
        pthread_t S, T;
        C = sqrt(INT_MAX)/10;
```

```
        pthread_create(&S, NULL, threadS, "S");
        pthread_create(&T, NULL, threadT, "T");

        sleep(rand()%50);

        if((Y - 1) == X) { // 7y^2 - 1 != x^2
                //always taken
        }
}
```

Listing 8.4: Concurrence based invariant

Environment based invariant is the mean of using any invariant from the system or libraries to create opaque predicates. For instance, as the stack is aligned on `x86` machines, *esp* % 4 is always true. As a matter of example listings 8.5 and 8.6 shows respectively to opaque predicates based on library invariants. Indeed, `SetErrorMode` on Windows always returns the last error mode code thus the condition will always be true. In the same vein, `strcpy` always returns the pointer to the destination string thus we can easily craft an opaque predicate based on this invariant.

```
SetErrorMode(1024);
// potentially far away
if(SetErrorMode(12)) {
        //always taken because return 1024
}
```

Listing 8.5: SetErrorMode opaque predicate

```
char s1[14] = "Hello world";
char s2[15] = "Say hi there";
if(strcpy(s2,s1) == s2) {
        //always taken as strcpy returns s2
}
```

Listing 8.6: strcpy opaque predicate

This work solely focuses on arithmetic opaque predicates as they are the most opaques predicates encountered. Recent work proposed to share apart the taxonomy of arithmetic opaque predicates in three categories [Min+15]:

- *invariant*: always true/false regardless of inputs values
- *contextual*: opaque thanks to constraint on input domain values (environment)
- *dynamic*: similar to contextual, but the opaqueness is made by comparing two dynamic configurations that are made to be disjoint thanks to dynamically computed values (eg: two disjoint pointer values etc..)

8.1.3 Detecting opaque predicates

Intuitively, to detect an opaque predicate, the idea is to get back on all its data dependencies. If the predicate is local, the distance from the predicate and its inputs instanciation will be short and the predicate will be relatively easy to break. Otherwise the distance is at most linear with the trace length which hardly scales in forward DSE. Interestingly, for invariant opaque predicates, it is not required to get back on all its input definition to break it, constraints on them might be enough. Consequently, a BB-DSE with a $pre^{\leq k}$ approach might be very handy.

Furthermore, it allows to bypass some dynamic tricks that would mislead a reverser into flagging a predicate as opaque. A good example is a predicate found in the packer ASPack. The predicate seemingly opaque, is not to be opaque due to self-modifications (Figure. 8.1). Statically, the predicate is opaque since BL is necessarily 0 but it turns out that the second bytes of the instruction MOV BL, 0X0 is being patched to 1 during the execution in order to take the other branch when looping back later on.

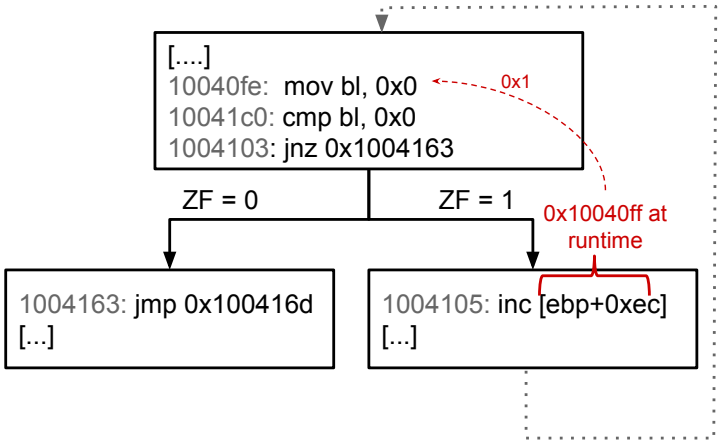


Figure 8.1: ASPack opaque predicate decoy

The key element of the approach is the aggregation of paths for a given bound k and a given predicate φ_p at the specified location l_p. Hence, the $pre^{\leq k}$ is computed for every new k-path to that predicate found in the trace. If all of the formulas generated evaluate to UNSAT the predicate is flagged OPAQUE, UNKNOWN otherwise. If both branches of a predicate are covered by the trace the predicate is flagged NOT OPAQUE. Table 8.2 summarizes the different possible status for a predicate.

Table 8.2: OP status w.r.t left/right branch status

Left	Right	Status
Covered	Covered	NOT OPAQUE
Covered	$\exists \pi,\ pre^{\leq k}_{\pi}(l, \varphi_p) \vdash SAT$	UNKNOWN
Covered	$\forall \pi,\ pre^{\leq k}_{\pi}(l, \varphi_p) \vdash UNSAT$	OPAQUE
Covered	$\forall \pi,\ pre^{\leq k}_{\pi}(l, \varphi_p) \vdash TIMEOUT$	TIMEOUT

8.1.4 Evaluation

Goal We are interested in the following two questions: ***Q1:*** *what is the detection capacity of our approach, and are the ratio of false positive, false negative acceptable?* The second question is ***Q2:*** *is our approach efficient and scalable?*

Benchmark and protocol In benchmarks, we have chosen the O-LLVM [Jun+15] in which we implemented various opaque predicates (cf. Table 8.3). O-LLVM has been selected because its core engine is open-source and it already implemented a mechanism for opaque predicates. Other obfuscation engines were not open-source [Col+12] or outdated. In order to evaluate the BB-DSE, two sets of benchmarks were selected:

- 5 simple programs taken from the State-of-the-Art in DSE for obfuscation [YD15]. Each of the sample was obfuscated 20 times (to obtain the same number of samples than *coreutils*). Programs are: *simple-if, bin-search, bubble-sort, huffman and matrix-multiply.*
- all 100 *coreutils* as a genuine dataset reference

Table 8.3: OP implemented in O-LLVM

Formulas		Comment
$\forall x, y \in \mathcal{Z}$	$y < 10 \| 2 \| (x \times (x-1))$	(provided with O-LLVM)
$\forall x, y \in \mathcal{Z}$	$7y^2 - 1 \neq x^2$	
$\forall x \in \mathcal{Z}$	$2 \| (x + x^2)$	
$\forall x \in \mathcal{Z}$	$2 \| \lfloor \frac{x^2}{2} \rfloor$	(2^{nd} bit of square always 0)
$\forall x \in \mathcal{Z}$	$4 \| (x^2 + (x+1)^2)$	
$\forall x \in \mathcal{Z}$	$2 \| (x \times (x+1))$	

In total 200 binary programs were used. For each of them a dynamic execution trace was generated with a maximum length of 20.000 instructions. By tracking where opaque predicates were added in the obfuscated files we are able a priori to know if a given predicate is opaque or not giving in consequence a set of ground truth values. Note that all predicates in *coreutils* are considered to be genuine (and then false positive if flagged as opaque). This large but controlled experiment allowed to test the accuracy of results. The 200 samples sums up a total of 1,091,986 instructions trace length and 11,725 conditionnal jumps with 6,170 genuine predicates and 5,556 opaques among them. Experiments were carried using different values for the bound k in order to find which one fits best at finding all opaque predicates without yielding too much false positives. As a matter of estimation a predicate marked opaque with $k = 2$ is more likely to be opaque than a predicate marked opaque with $k = 30$ since the path constraint might make the predicate unsatisfiable. Benchmarks were carried with a 5 second timeout per queries (k-path + predicate).

Results Table 8.4 and Figure 8.2 show respectively the graph and the table of the relations between the number of opaque (OP), genuine (OK), false positive (FP)/negative (FN) depending of the bound value k. The two last columns show the percentage of false positive among the number of all predicates and the number of false postives opaque predicates. These results represent predicates that only had one branch covered. Among the 11,725 predicates, 987 were fully covered by the trace and were excluded from these results, keeping 10,739 predicates (and 5,183 genuine predicates). The experiment shows a tremendous peak of opaque detection with an approximate $k = 10$. Alongside, the number steadily decreases as the number of false positive grows. All opaque predicates are rightly flagged opaque for $k = 16$ and luckily no predicate raised any TIMEOUT. The number of false positive 293 represent 6.28% of all predicates marked opaque and only 3.17% of all predicates for approximately 1.46 false positive per sample on average.

Note, that the path aggregation discussed in Section 8.1.3 provides a substantial gain since it reduces of 4.62% the number of false positive for k_{16} (see Table 8.5).

Table 8.4: Opaque predicate detection results

k	OP (5556)		Genuine (5183)		TO	FP/Tot (%)	FP/OP (%)
	ok	miss (FN)	ok	miss (FP)			
2	0	5556	5182	1	0	0.01%	0.02%
4	903	4653	5153	30	0	0.26%	3.22%
8	4561	995	4987	196	0	1.67%	4.12%
12	5545	11	4890	293	0	2.50%	5.02%
16	**5556**	**0**	**4811**	**372**	**0**	**3.17%**	**6.28%**
20	**5556**	**0**	**4715**	**468**	**2**	**3.99%**	**7.77%**
24	**5556**	**0**	**4658**	**525**	**7**	**4.48%**	**8.63%**
32	5552	4	4579	604	25	5.15%	9.81%
40	5548	8	4523	660	39	5.63%	10.63%
50	5544	12	4458	725	79	6.18%	11.56%

. Timeout: 5 sec
. a TO counts as an UNKNOWN result (hence, classify the predicate as genuine)
. 10,739 predicates, 5,556 opaque predicates, 5,183 genuine predicates

Resolution time We have performed another experiment about computation time of predicates solving to check if it scales on a large number of predicate (here more than 10000) with a 2.7GHz Core i7 and 16Go of RAM. Each predicate formula was solved using `Z3` with a 5 second timeout. Table 8.6 gives for each k-bound value the total time taken for solving and the average per query. For k_{16} the average time per query is 0.018s which tends to prove that this technique scales. Previous work aiming at solving invariant opaque predicates [Min+15] obtained on average, 0.49s per queries (min:0.09, max:0.79).

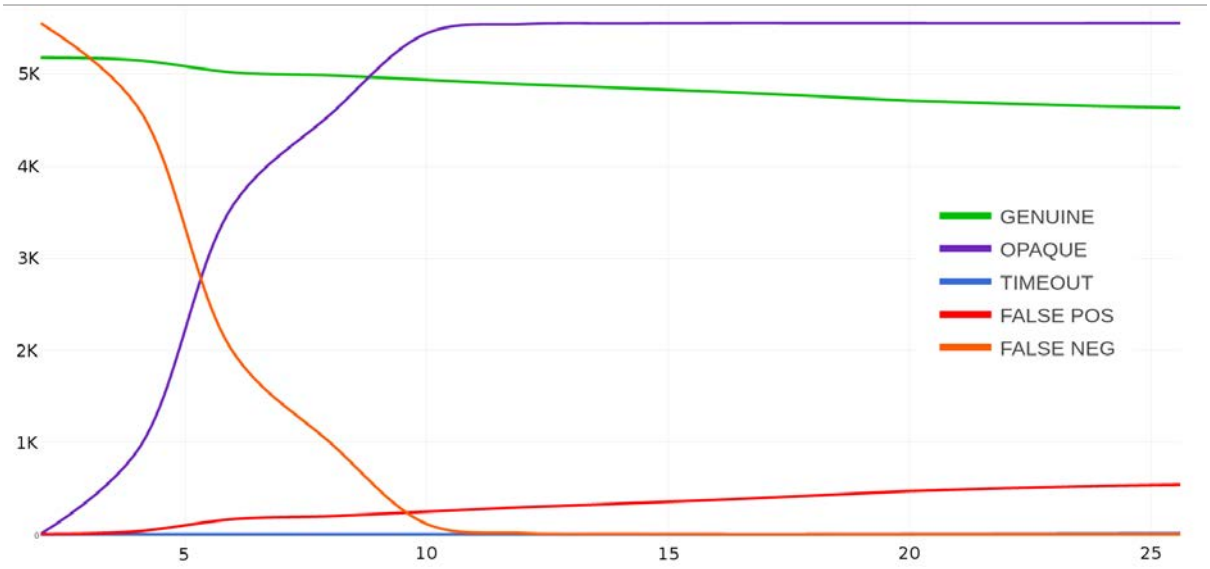


Figure 8.2: Opaque predicate detection results

Table 8.5: Difference without/with multi-path

K	OK			FP		
	∅	multip	gain(%)	∅	multip	gain(%)
2	5182	5182	0.00	1	1	0.00
4	5152	5153	0.02	31	30	3.23
8	4975	4987	0.24	208	196	5.77
12	4872	4890	0.37	311	293	5.79
16	4793	4811	0.37	390	372	4.62
20	4686	4713	0.57	495	468	5.45
24	4623	4651	0.60	553	525	5.06
32	4517	4558	0.90	640	604	5.62
40	4453	4492	0.87	696	660	5.17

. ∅ testing only one k-path for a predicate,

. *multip* testing all k-path leading to a predicate.

For this kind of opaque predicates this approach delivers a tremedous speed up (more than x27). These results validates the BB-DSE advantages compared to strictly forward DSE.

Table 8.6: Solving time

k	Total time (s) (10,739 queries)	Avg/query(s)
2	89	0.008
4	96	0.009
8	120	0.011
12	152	0.014
16	197	0.018
20	272	0.025
24	384	0.036
32	699	0.065
40	1145	0.107
50	2025	0.189

Conclusion Table 8.4 with k=16 achieves a very low false positives rate thus addressing **Q1** since there is no more than 2 false positive per samples and the number of false positive can be considered negligible compared to the number of true opaque predicates detected. The analysis is also efficient and scales thanks to its independence from the trace length which address **Q2**. As shown by Table 8.6 for k=16 the average time solving per query is 0.014s which is acceptable even with a large number of predicates to check.

8.2 Call/Stack tampering

8.2.1 Definition

A call stack tampering is the mean to alter the classical compilation scheme switching from a function to another using the `call` and `ret` instruction, where `ret` jumps on the address pushed on the stack by the `call`. The `ret` is tampered (a.k.a violated) if it does not return to the expected `returnsite` pushed on the stack by the `call`. This thesis proposes a more precise characterization and taxonomy of a call stack tampering.

A trivial example is given in Figure 8.3. From an attacker perspectives benefits are twofold:

- the reverser might be lured into exploring useless code starting from the `returnsite`,
- the real target of the `ret` instruction is hidden from the static analysis stand point.

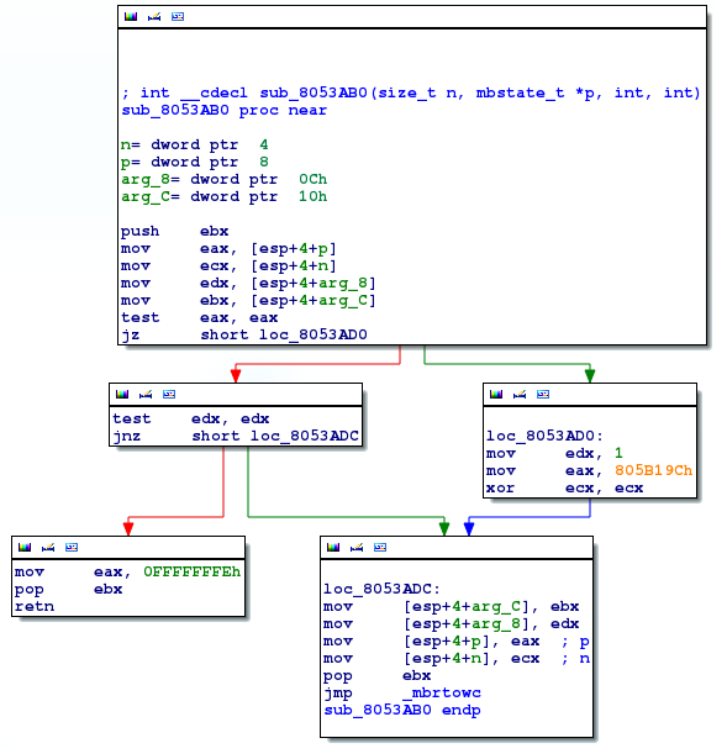
Subsequently, this obfuscation disrupt the high-level function recovery step of disassembly discussed in 2.1.3.

<main>:	<fun>:
call <fun>	[...]
..... // return site	push X
..... // junk code	ret //jump to X instead
..... // junk code	//of return site

Figure 8.3: Standard stack tampering

Tail transition Additionally, some compilation schemes make the function bound recovery harder by breaking on purpose the `call`/`ret` association. This compilation scheme called "tail transition" is aiming at reducing the function transition overhead in specific cases. It applies when a function calls another function in the last instruction of its body. The figure 8.4 shows an example taken from a program where the outcome of the function is either -1 in `eax` either calling `_mbrtowc`. In this later case the call has been compiled so that the function `_mbrtwoc` will perform the `ret` of `sub_8053AB0`. This can happen to be an issue in DSE if the function tail called (here `_mbrtwoc`) is not symbolically executed because its ret will not be visible.

Figure 8.4: Tail call to _mbrtwoc



8.2.2 Taxonomy & Examples

From a semantic point of view, a `ret` is just an indirect jump to the first value on the stack which can lead to various different scenarios of violation:

- ret performed with a dynamic jump
- no `call` but a `ret` (orphan `ret`)
- `call`, tampering of the return value and then `ret`

In this work we strengthen the violation definition to better characterize it. For reverse-engineering purposes it is important to distinguish if a `ret` does not return well, if the value has been overwritten or if the stack pointer (`esp`) has been tampered. This approach identifies the following `ret` properties:

- *integrity*: does `ret` returns to the same address as pushed by the `call`? Characterize if the tampering takes place or not. A `call`/`ret` pair is either `[genuine]` (returns to the caller) or `[violated]`;
- *alignment*: does the stack pointer (`esp`) is equal at `call` and at `ret`? If so, the stack pointer is denoted `[aligned]`, `[disaligned]` otherwise;
- *multiplicy*: in case of violation, is the current `ret` target the only possible value? This case is denoted `[single]` and `[multiple]` otherwise.

As such, a well-behaving `ret` should be marked `[genuine]` and `[aligned]`. Conversly, a tampered `ret` should be marked `[violated]` with optional tag `[single]`,`[multiple]` or `[aligned]`,`[disaligned]`.

This taxonomy helps discriminating different kinds of violations. Figure 8.5 shows an example where the `ret` does not jump back to the return site with an offset but beyond its own address with an offset. According to the taxonomy defined hereabove the violation is detected as `[violated]`, `[aligned]`, `[single]` tags as it jumps to the return site with an offset of +8.

address	mnemonic	comment
080483d1	call +5	//push 0x080483d6 as return (jump on next instruction)
080483d6	pop edx	//pop the return address
080483d7	add edx, 8	//add 8 (size of pop+add+push+ret+invalid)
080483da	push edx	//put back the return value
080483db	ret	//returns at 0x080483de (beyond invalid byte)
080483dc	.byte{invalid}	//invalid byte (cannot disassemble)
080483de	[...]	//payload

Figure 8.5: Violation “call +5”

8.2.3 Tampering detection

This study uses the BB-DSE algorithm to detect the tampering. Of the author's knowledge DSE have never been used to detect such obfuscation. The purpose is to check the infeasibility of the tampering by checking the validity of some predicates. As a recall, a predicate ϕ is VALID iff: $unsat(\neg\phi)$. To characterize the violation, the following queries are performed:

- *integrity:* $@[esp_{\{call\}}] = @[esp_{\{ret\}}]$: Compare logically the content of the value pushed at call ($@[esp_{\{call\}}]$) with the one used to return ($@[esp_{\{ret\}}]$). If it evaluates to UNSAT, a violation necessarily occured (`[violated]`), VALID the predicate returns and cannot be tampered (`[genuine]`). If SAT, the concrete values from the trace are used to characterize the `ret` integrity;
- *alignment:* $esp_{\{call\}} = esp_{\{ret\}}$: Compare the logical `esp` value at the `call` and at the `ret`. If it evaluates to VALID, the `ret` necessarily returns at the same stack offset (`[aligned]`), if it evaluates to UNSAT the `ret` is (`[disaligned]`), otherwise the alignment cannot be characterized;
- *multiplicity:* $\mathcal{R} \neq @[esp_{\{ret\}}]$: Check if the logical `ret` jump target ($@[esp_{\{ret\}}]$) can be different from the concrete value ($\mathcal{R}$) in the trace. If it evaluates to UNSAT the `ret` cannot jump elsewhere and is flagged `[single]`. Be aware that `[multiple]` cannot be found by our approach, unless several distinct (non-legit) targets are observed at runtime. Indeed, finding new targets with DSE would require forward DSE.

Table 8.7: Call stack tampering taxonomy

RT Status	integrity	alignement	multiplicity
RT Genuine `[genuine]`	RT: OK VALID: `[genuine]` (proved) SAT: / UNSAT: $\emptyset$	RT: OK/KO `[aligned]` `[disaligned]` - VALID: `[aligned]` (proved) - SAT: / - UNSAT: `[disaligned]` (proved)	
RT Tampered `[violated]`	RT: `[violated]` - VALID: $\emptyset$ - SAT: / - UNSAT: `[violated]` (proved)	(idem)	RT: 1/`[multiple]` - VALID : $\emptyset$ - SAT: / - UNSAT: `[single]`

Table 8.7 summarizes all the possible states and attached tags. This stack tampering analysis uses the same BB-DSE as opaque predicates but with slightly differents settings. In this case, the k value will be different for every `call`, `ret` pair. The trace is analysed in a forward manner each `call` encountered is pushed to a call stack. Upon `ret` encounter, the first call on the stack is poped and a $pre^{\leq k}$ is performed where k is the distance between the `call` and the `ret`. The different queries are then solved on this bounded path slice. Two problems might corrupt the `call` stack breaking the synchronicity of `call`,`ret` pairs:

- Tail call [MM16] to non-traced function: As shown above tail calls are the mean of calling functions using a jump instruction instead of call to avoid stack tear-down. Similarly to the previous case, a tail call in a non-traced instruction will hide the `ret` instruction, so the current call stack head should be poped to keep synchronicity.
- `ret` performed with dynamic jumps. As our analysis is `ret` centric any obfuscation replacing all rets by dynamic jumps would make the analysis inefficient

8.2.4 Evaluation

Two benchmarks are performed to assess the analysis effectiveness. First, a controlled benchmark is performed on samples, obfuscated on purpose. Second, a benchmark of various packers is performed to ensure the scalability on real-world obfuscated programs. The first benchmark is on the same 5 samples as opaque predicates, but obfuscated with the source-to-source obfuscator Tigress [Col+12]. Files have been obfuscated with the AntiBranchAnalysis transformation that replace all conditionnal branches with calls and rets. The two schemes used here are:

- push; call; ret; ret (the second `ret` will jump on pushed value)
- push; ret (similar to previous)

The second experiment is a free wheel experiment on few packers to find if violations were employed. Packed binaries are chosen as they are more likely to contain such protections. Indeed, they are usually the first line of protection to be broken in order to reach a malware payload. In this experiment, the payload was a legitimate program (`hostname` on Windows).

Table 8.8: Call stack tampering results

Sample	runtime genuine			runtime violation		
	`#ret` †	proved genuine	align/ disal	`#ret` †	alig/ disal	proved single
simple-if	6	6	6/0	9	0/0	8
bin-search	15	15	15/0	25	0/0	24
bubble-sort	6	6	6/0	15	0/1	13
mat-mult	31	31	31/0	69	0/0	68
huffman	19	19	19/0	23	0/3	19
ASPack	11	9	9/0	6	5/1	2
ACProtect	0	0	0/0	48	45/1	45
Crypter	125	94	94/0	78	0/30	32
PE Lock	4	3	3/0	3	0/1	0

†each `ret` is counted only once

Results Results are given in the table 8.8. The first 5 samples, totalize 218 different `ret`. Among them 77 were genuine and 141 violated. Due to the homogeneity of schemes inserted by Tigress, the diversity of tags is very low but interestingly enough results did not yield any false positive nor false negative.

The second benchmark performed yields very interesting results. For instance on ACPacker, no `ret` were legit and the fact that almost all of them were flagged `[single]` means `ret` are likely be used as static jumps. Results show that stack tampering obfuscation is far from being negligible in these packers since approximately 54% to 100% of all call/rets are tampered in these packers. As an example, three different kinds of tampering were detected in the ASPack packer which correspond to the three examples (see Figures 10.2, 10.1 and 10.3 in Chapter 10).

Chapter 9

Analysis combination

This chapter present three different kind of analyses combinations based on the algorithms and methods described previously in this thesis. The first combination is a disassembly algorithm taking advantage off both dynamic and symbolic approaches limitating the weaknesses of the purely static approaches. The second combination is a static and symbolic combination based on abstract interpretation and DSE geared for vulnerability discovery. The last combination is a forward and backward static analysis combination aiming at detecting infeasible test requirements for software testing. Based on abstract interpretation and weakest-precondition calculus, it differs from the other combination by working at the source level.

9.1 Combination principles

In general, combining different approaches is a strength from the program analysis perspective, especially when dealing with obfuscation as one might target a specific kind of analysis. As an example, Marion et al proposed a mixed static and dynamic approach for disassembly purposes [Bon+15] called *concatic disassembly*. The idea is to enrich static disassembly with dynamic analysis allowing to handle both self-modification and code-overlapping, which techniques are hindering static analysis. The disassembly obtained is more precise and efficient than a pure static disassembly. The key point of this approach is to obtain a proper segmentation of the different self-modification layers.

Forward/Backward combination Let's take the formal approach; some properties are to be checked on a given program. Various semantic analyses techniques like abstract interpretation, weakest-precondition calculus or symbolic execution exist to address this problem but each of them have theirs particularities. The two main characteristics are whether the analysis works forward or backward and whether the analysis is under or over-approximated. Table 9.1 shows the characteristics of different analyses and their abilities to detect some properties. For instance abstract interpretation computes an over-approximated invariant of the whole program, while symbolic execution computes under-approximated but more precise values for a given path. Similarly, forward DSE is able to generate new inputs to cover a given path, while backward-DSE cannot. But it

scales on any path length. It all depends on predicates needed to be checked as chown in the previous Chapter 8.

Table 9.1: Analysis techniques comparison

	Weakest-Precondition	**Abstract Interpretation**	**DSE**	BB-DSE	**Random Testing**
Approximation	Over-approximation	Over-approximation	Under-approximation	Under-approximation	Under-approximation
Type	Static	Static	Dynamic/Symbolic	Dynamic/Symbolic	Dynamic
Direction	Backward	Forward	Forward	Backward	Forward
Produce	predicate statement	Invariant on the program	Constraints on path and inputs	Constraints on sub-path	Inputs + coverable paths
Use	Function	Program	path	path	program (+tests requirements)

Combination interaction Another critical aspect when designing an analysis combination is to determine how the two analyses will communicate and what will the information exchange be. This depends on the information and results produced by the analysis. For instance, abstract interpretation can transmit information about its invariant (see Section 9.4), while fuzzing or random testing yield inputs and information about covered paths. A strict black-box integration usually weaken the potential of algorithms combination while a gray-box or a white-box combination unleash the analyses potential but is harder to do.

9.2 Sparse disassembly: Use-case binary-level

9.2.1 Introduction

Standard disassembly We call *legit* an instruction in a binary code if it is part of a real execution of the code. Two qualities expected for a disassembler are (1) *safety: does the algorithm recover only legit instructions?* and (2) *completeness: does the algorithm recover all legit instructions?* Standard disassembly approaches essentially include static recursive disassembly, static linear sweep and dynamic disassembly.

- **Recursive disassembly** consists in exploring the executable file from a given entry-point, recursively, following the possible successors of each instructions. A recursive disassembly may miss a lot of instruction, typically, because of computed jumps (`jmp eax`) or of self-modification. Like all static algorithms, the approach can easily be fooled into disassembling junk code from opaque predicates or call/stack tampering spurious bytes. The approach is neither safe nor complete.

- **Linear sweep** consists in decoding linearly all possible instructions in executable code sections. The technique aims at being more complete than recursive traversal, yet it comes at the price of many additional mis-interpreted code instructions. Like all static techniques it can still miss instructions hidden by code overlapping or self-modification. Hence the technique is unsafe and often incomplete on obfuscated codes.

- **Dynamic disassembly** retrieves only instructions and branches observed at runtime on one or several executions. That technique is safe, but possibly very incomplete. Yet, it recovers instructions masked by self-modification, code overlapping, which are out of reach for static disassembler.

For example, `objdump` is based on linear sweep, while `IDA` performs a combination of linear sweep and recursive disassembly (enhanced with severals heuristics).

Static/Dynamic combination As mentioned in previous Section (9.1), various approaches mix both static and dynamic analyses. Doing so helps to bypass the dynamic jumps, self-modification and code-overlapping issues. Nonetheless, it cannot address dead code and opaque predicates which will still be disassembled. Similarly, dead `returnsite` (never executed) due to call/stack tampering will still be disassembled. While such combination is better than static or dynamic alone, it can still be improved to circumvent aforementioned obfuscations.

9.2.2 Principles

As already explained above, standard disassembly approaches are not satisfactory. Hence, we propose a *sparse disasembly*, an algorithm based on dynamic disasssembly reinforced with a static disassembly guided with complementary information about obfuscation (computed by backward-bounded DSE). The goal is to provide a more accurate and sound disassembly. The approach takes advantage of the two analyses presented in Sections 8.1 and 8.2 in the following way:

- use dynamic values found in the trace to keep disassembling after indirect jumps instructions and disassemble the "self-modified" code;
- use a static recusive disassembly algorithm to enlarge the dynamic disassembly but the disassembly is guided by the symbolic execution. Thus it takes into account the following properties:
  - use opaque predicates found by BB-DSE to avoid disassembling dead branches (thus limiting the number of recovered non legit instructions);
  - use call/stack tampering information found by bb-DSE to disassemble `ret` targets in case of violation, as well as not to disassemble the `returnsite` of the `call` in this case.

Implementation This algorithm has been integrated in the existing recursive traversal disassembly algorithm implemented in BINSEC. The BB-DSE procedure sends OP and `ret` information to the modified disassembler which takes them into account when recursively disassembling the program.

9.2.3 Evaluation

Opaque predicates The first benchmark aims at testing the disassembly of O-LLVM obfuscated samples using the *sparse disasembly* approach. Table 9.2 gives the results of the coverage for IDA, `Objdump`, BINSEC (recursive traversal) and BINSEC*sparse disasembly* on few obfuscated samples. Once obfuscated, files are approximately 8 times their original size. IDA uses both recursive disassembly, linear sweep and some heuristics to disassemble. `Objdump` does linear-sweep only. IDA and BINSEC results are almost identical because none of the samples appear to have indirect jumps. The `Objdump` linear-sweep always over-disassemble some bytes at the end of functions. Interestingly, the *sparse disasembly* provides an average gain of 22.48% in terms of instructions coverage. The number of instructions is still far bigger than the original program. So, while it is possible to prune the dead branches, the instructions computing the opaque predicate still have to be computed.

Table 9.2: Sparse disassembly opaque predicates

sample	no obf.	Obfuscated					gain vs IDA (sparse)
		perfect	IDA	Objdump	BINSEC		
					rec.	**sparse**	
simple-if	37	**185**	240	244	240	**185**	23,23%
huffman	558	**3226**	3594	3602	3594	**3226**	10,26%
matrix_multiply	249	**854**	1075	1080	1075	**854**	20,67%
bin_search	105	**833**	1110	1115	1110	**833**	24,95%
bubble_sort	121	**1026**	1531	1537	1531	**1026**	32,98%

Call/Stack tampering Similarly, we have performed a call/stack tampering benchmark on programs obfuscated this time with Tigress. In contrast to the previous benchmark, the aim is to have the bigger (yet accurate) possible coverage. The obfuscation turns every conditional jumps into calls and rets. No any additional spurious instructions are introduced. For this reason, the approximate size of obfuscated programs is only twice bigger.

In this benchmark, IDA and `Objdump` perform well thanks to their linear-sweep. BINSEC in recursive traversal performs badly with rets and some `jump eax` added by the obfuscator. Nevertheless, the sparse disassembly performs very well with an instruction differential ranging from 12 to 63. A manual verification showed that this difference was an over-disassembly of `returnsite` that is never executed due to the stack tampering. Once again, the *sparse disasembly* provided a better and sound disassembly of the program.

Coreutils The last benchmark aims at proof-testing the *sparse disasembly* on larger obfuscated binary samples to test its efficiency. Thus, we have obfuscated the *coreutils* using O-LLVM in order to insert opaque predicates into them. Unfortunately, it was not

Table 9.3: Sparse disassembly stack tampering

sample	no obf.	Obfuscated					gain vs IDA (sparse)
		perfect	IDA	Objdump	BINSEC		
					rec.	**sparse**	
simple-if	37	**83**	95	98	30	**83**	14.45%
huffman	558	**659**	678	683	526	**659**	2.80%
matrix_multiply	249	**461**	524	533	23	**461**	12.0%
bin_search	105	**207**	231	238	47	**207**	10.39%
bubble_sort	121	**170**	182	185	89	**170**	6.6%

possible to mix it with Tigress that uses some inline assembly constructs not supported by Clang (and required by O-LLVM).

Table 9.4 shows the results on 5 *coreutils* selected randomly. While the samples are significantly bigger, the ratio between `Objdump`/IDA and BINSEC rec/sparse remains roughly similar. Obviously, the coverage of both `Objdump` and IDA is larger since it takes advantage of the linear sweep not to miss any function. Yet, they also disassemble lots of junk code.

These experiments have demonstrated that *sparse disasembly* is an effective way to *enlarge a dynamic disassembly*, in a both *significant and safe manner*. Indeed, sparse disassembly recovers between 6x and 16x more instructions than dynamic disassembly, yet it still recovers slightly less instructions than recursive disassembly (including less junk), and much less than linear sweep – partly due to the focused approach of dynamic disassembly. Hence, sparse disassembly stays close to the original trace.

Table 9.4: Sparse disassembly coreutils

sample	Tr.len	Obfuscated				
		Objdump	IDA	Dynamic disas.	BINSEC	
					rec.	**sparse**
basename	1,783	20,776	20,507	1,159	8,163	7,894
env	3,692	19,714	19,460	477	6,916	6,743
head	17,682	32,840	32,406	1,299	20,098	19,807
mkdir	1,436	57,238	56,767	1,407	10,562	10,428
mv	14,346	115,278	114,067	5,261	82,502	81,596

Conclusion The carried experiments brought very good and accurate results about controlled samples, achieving perfect disassembly. From this stand-point, *sparse disasembly* performs better than the combination of both recursive and linear like in IDA, including up to 30% less recovered instructions than IDA. The *coreutils* experiments showed that

sparse disassembly is also an effective way to enlarge a dynamic disassembly in a both significant and safe manner.

Yet, our *sparse disasembly* algorithm is currently limited by the inherent weaknesses of recursive disassembly for example the handling of all the computed jumps targets would require advanced pattern techniques.

Thus, *sparse disasembly* performs better on obfuscated samples than recursive disassembly alone and limitate the over-disassembly shortcoming of linear-sweep based approaches.

9.3 Use-After-Free detection: Use-case binary exploitation

9.3.1 Introduction

This works has been performed in collaboration with Verimag Grenoble and especially Josselin Feist to detect and exploit Use-After-Free (UaF) vulnerabilities. The technique is based on static analysis and Dynamic Symbolic Execution(DSE) and led to the discovery of various vulnerabilities and most notably CVE-2015-5221 [Mit15]. An UaF appears when a heap pointer is *used* after having been *freed*. Such memory corruptions can in some cases be leveraged by an attacker to execute arbitrary code and consequently to craft an exploit. Finding an UaF is particularly difficult as it is only triggerable after the sequence of events `malloc`, `free` and `use` for a given resource. While these three events can be located at distant area of the code. Finding such sequences requires some path coverage algorithm, which are non-linear in the general case. Moreover, it requires, at some extend, alias analysis to compare pointers on the heap. It also requires alias analysis which is known to be hard for static analyzer.

9.3.2 Combination

The combination first performs a static analysis of the code and extract a *weighted slice* containing relevant events (`malloc`,`free`,`use`). This is later used by a guided dynamic symbolic execution to trigger the bug and to generate appropriate inputs. Static analysis phase aims at reducing the coverage surface for the DSE in order to improve its efficiency. The DSE algorithm is also empowered with specific heuristics guiding the path selection and enumeration so as to maximize the chance to find a solution quickly.

- **Static Analysis** is performed by a tool GUEB [Fei15] which performs a value analysis based on abstract interpretation keeping as abstract states the set of *allocated* and *freed* memory chunks. Within the same chunk, if the sequence `malloc`, `free`, `use` is found, a slice containing all paths to the three events is extracted from the CFG. This slice is then transmitted to the DSE for UaF vulnerability discovery.

- **Guided DSE** performs a path exploration of each *slice* provided by the static analysis, to validate or invalidate each candidates. For a given slice, the DSE explores

all the possible paths until finding one satisfying the conditions to trigger the UaF. Enumerating paths can lead to the so called *path explosion* problem. Even though the analysis is performed on a *slice* it also uses its guidance heuristics to return the best candidate (path) based on a score. The score is based on the distance between the three points of interest (`malloc`, `free`, `use`).

9.3.3 Evaluation

Experiments are peformed on several versions of the DSE namely with and without guiding heuristics. Results are compared against the standard fuzzers AFL [lca16] and radamsa [Arc16]. The experimentation is performed on JasPer[8], an image conversion library. The analysis has found an UaF in the `mif_process_cmpt` function which subsequently led to the CVE-2015-5221 identifier [Mit15]. From the performance perspectives, without the guiding heuristics BINSEC/SE is unable to generate a PoC to trigger the vulnerability while with the heuristics a PoC is generated in 20 minutes.

9.3.4 Conclusion

This combination approach designed for UaF detection relies on the use of a *weighted slice* computed by abstract interpretation and path coverage with symbolic execution. It proved to be efficient on real-world applications. In addition, *guiding heuristics* proved to be useful and further work are ongoing to make them more generic. Indeed, they are for now driven by examples encountered.

9.4 [Bonus] Infeasible tests requirements: Use-case source-level

This section shows a use-case where the forward/backward combination approach has successfully been applied to a slightly different domain, namely *software testing* at source level. As the language and tools change, the approach remains similar. It emphasizes the portability of such approach accross languages and an example of the variety of domain where it can be applied. This work has been published in [Bar+15b].

9.4.1 Software Testing: Introduction

In software testing, a coverage criteria specify the requirements to be covered by a test case. These coverage criteria are normative test requirements that the tester must satisfy before delivering the software under test. Among existing coverage we have: decision coverage (DC), condition coverage (CC), multiple condition coverage (MCC), function coverage or weak mutations (WM). Figure 9.1 illustrates possible encodings for selected critera.

[8] https://www.ece.uvic.ca/ frodo/jasper

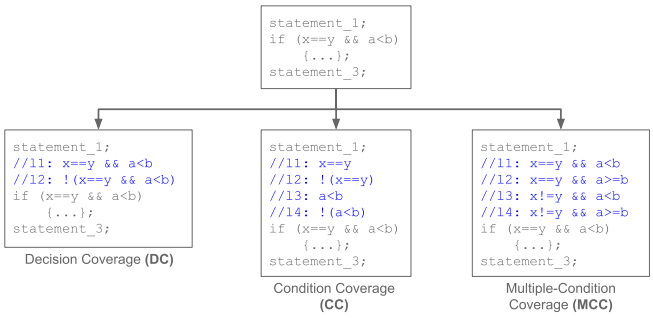


Figure 9.1: Standard coverage criterias

Labels Bardin et al use labels [Bar+14; BKC13] as a convenient formalism to specify and to encode several structural testing criterias into a unified solution making use of *existing verification tools*. Given a program P, a *label* l is a pair $\langle loc, \varphi \rangle$ where loc is a location in P and φ is a predicate over the internal state at loc. We say that a test datum t covers a label $l = \langle loc, \varphi \rangle$ if there is a state s such that t reaches $\langle loc, s \rangle$ and s satisfies φ. An *annotated program* is a pair $\langle P, L \rangle$ where P is a program and L is a set of labels in P. The main benefit of labels is to unify the treatment of test requirements belonging to different classes of coverage criteria. While Bardin et al explored the question of automatic test generation with labels [BKC14], we focus in this section on the problem of infeasibility test requirement detection.

9.4.2 Problematic

In practice, such test criterias are limited by the infeasibility problem [Wey93; WHH80; YM89], where a requirement cannot be covered by any test case due to inherent program structure. In addition to yield a worse code coverage, automated testing tools might waste time trying to cover the criteria. Identifying them is an undecidable problem [GWZ94], so it affects the decision to stop testing as it requires the knowledge of what remains to be covered. As consequence, it is difficult to accurately measure the coverage of the test suite which is usually used to stop the testing process (usually 80% to 100%).

9.4.3 Combination

The motivation of the combination is based on the fact that infeasible requirements can be transformed into assertion validity problem verifiable with various analyses. This study considers two analyses *Value Analysis* (forward) and *Weakest Precondition calculus* (backward) as basis of the combination. The idea is to compute an over-approximation of all reachable program states with *Value Analysis*(VA). Then with the *Weakest Precondition*(WP) and an assertion to check, the goal is to compute a proof obligation equivalent to the validity of the assertion. The combination improvement is to extract relevant information from the state approximation of VA for the property to check, and to submit them as *assume* to WP to enforce and improve its decision power. This communication

is performed in a Grey-box manner, by inserting assume annotations in the common annotation language used (see the next Section 9.4.4). Let P, be a program, and A a set of assertions in P which we want to check the validity.

The combination VA $\oplus$ WP depicted hereabove is divided in different steps: (1) VA analysis computes once for all a state approximation S of the program. (2) For each $a \in A$, if a is valid the method terminates. Otherwise, the greybox part starts by extracting the set R_a of variables and locations relevant to a. Next, the set R_a and the previously computed approximation S are used to deduce properties of relevant variables that will be submitted as hypotheses to the last step. (3) a WP analysis step checks if the assertion a can be proven valid using the additional hypotheses in H.

Advantages The proposed combined approach takes benefit both from the global precision of the approximation computed by VA for the whole program and the local precision of analysis for a given assertion ensured by WP. Therefore, this technique is likely to provide a better precision than the two methods used separately. Careful selection of information transferred from VA to WP tries to minimize information exchange: the amount of useless (irrelevant or redundant) data is reduced, thanks to the greybox combination. On the other hand, even if not being purely black-box, the grey-box approach remains lightweight and non-invasive: only basic knowledge of the WP technique is required to implement the validation step, and only basic knowledge of datastructures and contents of approximations computed by VA is necessary to query them and to produce hypotheses at the $\mathcal{H}_P^{VA}$ step. Neither VA nor WP requires any modification of the underlying algorithms. Moreover, the approximation S is computed only once, and then used for all assertions. These elements are the foundation of the two methods combination.

9.4.4 Implementation

This combination was developped within the LTest testing toolkit [**Bardin2014TAP**] into a component called LUncov dedicated to the detection of infeasible test requirements. This toolkit uses the DSE engine PathCrawler [Wil+05] itself included as a plugin into Frama-C [Cuo+12]. Thus, this combination takes advantage of both Abstract Interpretation (Value Analysis) and Weakest Precondition analysis provided with Frama-C. Also, Frama-C provides a common Abstract Syntax Tree (AST) and a ANSI/ISO C Specification Language (ACSL) [Cuo+12] to express annotations used by LTest to generate test requirements. Thus, hypotheses H are implemented as ACSL annotations assumed to be true.

Example The use case presented in Figure 9.1 aims at emphasis benefits of combining both Value Analysis and Weakest Precondition to prove properties that would not be possible otherwise. Note that `Frama_C_interval` is a built-in Frama-C function that return a non-deterministic value into the given range and used by the different Frama-C analyses.

```
int main() {
        int a = Frama_C_interval(0,20);
        int x = Frama_C_interval(-1000,1000);
        return g(x,a);
```

```
}

int g(int x, int a) {
        int res;
        if(x+a >= x)
        res = 1;
        else
        res = 0;
        //assert res == 1;
```

Listing 9.1: Use case source code

The assertion in function `g` cannot be proved neither by VALUE nor by WP alone on a platform like FRAMA-C. VALUE is unable to prove that `x+a` $\geq$ `x` is always true because it lacks of relational domains and does not recognize that the x on both side of the term is the same variable and having the same value[9]. The WP because it works on a per function basis, has no knowledge of `x` and `a` valuation in the function `g` and because of possible int overflows it cannot prove the assertion.

Using our combined method, we first run VALUE on the whole program, then we bring all relevant variable information into `g`, which is transformed into the code presented in Figure 9.2. With this additional information, WP gained enough information to prove the assertion.

```
int g(int x, int a) {
        //@assume a >= 0 && a <= 20;
        //@assume x >= -1000 && x <= 1000;
        int res;
        if(x+a >= x)
        res = 1;
        else
        res = 0;
        //@assert res == 1;
}
```

Listing 9.2: Use case enriched with hypothesis for WP

9.4.5 Experiments & Results

In the experiments the tools mentionned in Section 9.4.4 were used on 12 benchmarks coming from the Siemens test suite (`tcas` and `replace`), the Verisec benchmark (`get_tag` and `full_bad` from Apache source code) and MediaBench (`gd` from `libgd`). We also consider three coverage criteria: CC, MCC and WM [AO08]. Each of these coverage criteria were encoded with labels as explained in Section 9.4.1. Our overall benchmark consists in 26 pairs program–test requirements. Among the 1,270 test requirements of this benchmark, 121 were shown to be infeasible in a prior manual examination. We solely focus here on the detection efficiency. The impact on test generation (not the focus of this thesis) is detailled in the article [Bar+15b].

[9] relational domains have very recently been added into VA prior to this work

We then compared the percentage of infeasible requirements detected per program and per criterion with 3 methods. (1) abstract interpretation alone with VA, (2) weakest precondition alone with WP and (3) VA $\oplus$ WP the combination of the two. Table 9.5 gives the results in terms of detection for each pair program/criterions.

From these results it becomes evident that all the studied methods detect numerous infeasible requirements. VA detects 84 infeasibles requirements, WP 73 and the combination of the two 118. Out of the three methods, our combined method VA $\oplus$ WP performs best as it detects 98% of all the infeasible requirements. The VA and WP methods detect 69% and 60% respectively. Interestingly, VA and WP do not always detect the same infeasible labels. For instance, WP identifies all the 11 requirements in `fourballs`–WM while VA finds none. Regarding the `utf8-3-WM`, VA identifies all the 29 labels while WP finds only two. This is an indication that a possible combination of these techniques, such as the VA $\oplus$ WP method, is fruitful. Thus, VA $\oplus$ WP finds at least as much as VA and WP methods on all the cases, while in some, i.e., `replace`-WM and `full_bad-WM`, it performs even better.

These results have shown that the combination VA $\oplus$ WP performs better than VA or WP alone, and achieve a 98% rate of detection on our benchmarks. Thus, the combination of forward/backward analyses sketched at the begining of the chapter shows that it finds application beyond binary analysis.

Table 9.5: Infeasible Label Detection Power

Program	**LOC**	**Crit.**	**#Lab**	**#Inf**	**VA**		**WP**		**VA ⊕ WP**	
					#D	**%D**	**#D**	**%D**	**#D**	**%D**
trityp	50	CC	24	0	0	–	0	–	0	–
		MCC	28	0	0	–	0	–	0	–
		WM	129	4	4	100%	4	100%	4	100%
fourballs	35	WM	67	11	0	0%	11	**100%**	11	100%
utf8-3	108	WM	84	29	29	100%	2	7%	29	100%
utf8-5	108	WM	84	2	2	100%	2	100%	2	100%
utf8-7	108	WM	84	2	2	100%	2	100%	2	100%
tcas	124	CC	10	0	0	–	0	–	0	–
		MCC	12	1	0	0%	**1**	**100%**	1	100%
		WM	111	10	6	60%	6	60%	**10**	**100%**
replace	100	WM	80	10	5	50%	3	30%	10	100%
full_bad	219	CC	16	4	2	50%	4	100%	4	100%
		MCC	39	15	9	60%	15	100%	15	100%
		WM	46	12	7	58%	9	75%	11	**92%**
get_tag-5	240	CC	20	0	0	–	0	–	0	–
		MCC	26	0	0	–	0	–	0	–
		WM	47	3	2	67%	0	0%	2	**67%**
get_tag-6	240	CC	20	0	0	–	0	–	0	–
		MCC	26	0	0	–	0	–	0	–
		WM	47	3	2	67%	0	0%	2	**67%**
gd-5	319	CC	36	0	0	–	0	–	0	–
		MCC	36	7	7	100%	7	100%	7	100%
		WM	63	1	0	0%	0	0%	**1**	**100%**
gd-6	319	CC	36	0	0	–	0	–	0	–
		MCC	36	7	7	100%	7	100%	7	100%
		WM	63	0	0	–	0	–	0	–
Total			1,270	121	84	69%	73	60%	118	**98%**
Min				0	0	0%	0	0%	2	**67%**
Max				29	29	100%	15	100%	29	100%
Mean				4.7	3.2	63%	2.8	**82%**	4.5	**95%**

#D: number of detected infeasible labels %D: ratio of detected infeasible labels
–: no ratio of detected infeasible labels due to the absence of infeasible labels

Chapter 10

Real-World case studies

This chapter discusses two case studies aiming at emphasizing the efficency of detection methods designed in the previous Chapter 8. The first case study examined is a free evaluation of the detection algorithms on packers. The second, one is more targeted to a single piece of malware which particularity is to be heavily obfuscated and to have been used in severe targeted attacks.

10.1 Large scale evaluation of packers

10.1.1 Context & Dataset

Packers Packers are programs embedding another program. This can be done for compression purposes but its more usually done for protecting the embedded software of the reverse-engineering. While they can be used for legitimate purposes lots of malware are using either commercial or custom packers to hide their code. Because packers are usually the first layer of protection to be broken before any in-depth malware analysis, performing a large scale analysis on them is worthy. In addition, experience shows that packers are more likely to use obfuscation than malware payload that usually relies on such overlay of protection. We want to see if our techniques are able to detect OPs or CS tampering on real packers.

Targeted obfuscation The two obfuscations directly targeted are opaque predicates and call/stack tampering thouroughtly discussed in Section 8.1 and 8.2. Hence, performing these two analyses in a black-box manner, without prior knowledge of the code is a great way to validate at a larger scale the efficiency of the two detection analyses.

Dataset The packer set originates from previous studies performed on standard packers, especially *concatic disassembly* benchmarks [Bon+15]. All packers are Windows binaries used to pack the `hostname` utility. Choosing `hostname` as stub binary is totally arbitrary but has the advantage to be small and to provide a good oracle to know whether or not the packing succeed and whether the binary kept the same observable behavior or not. The list of packers is not exhaustive as newcomers appear regularly but still provide a

large representative panel (33). A priori, we do not know if there is any opaque predicates or call/stack tampering. The only obfuscation surely present is the self-modification.

10.1.2 Evaluation Settings

The instrumentation was limited to 10M instructions. All samples reaching this limit were not analysed because yielding a partial trace. The dynamic instrumentation appeared not to be perfect, due to the number of process, threads created and all the counter-measures used within the packers. As a sanity check, we ensure that the execution trace was successfully created without unexpected termination (tr.ok) and if the hostname of the machine appeared at the end of the execution (host) in order to analyse the trace. Also as a comparison with the previous study, the number of waves detected is put in perspective between Codiasm and Binsec. The modeling varies and thus impact the number of waves yielded. The difference lays in the way to propagate the marking of memory while doing the dynamic instrumention and is discussed in Annex B.

Analyses The bound used for the opaque predicates detection is 16 as it emerged as the best value from the benchmarks performed in Section 8.1.3. We note, OK, OP, To and Covered, the predicates which are respectively not opaque, opaque, unknown (because of timeout) and fully covered (both branches) by the dynamic instrumentation. The call/stack tampering is run without specific parameters.

10.1.3 Results

Some results of the previous study are given like the number of processes (#proc) and the number of threads(#th) spawned. Table 10.1 gives the complete results of the two analyses. Interestingly, some packers do not make use of obfuscation while others seems to use it heavily e.g `Crypter`. Discriminating true positive from false positive would require to check all of them individually. It is worth noting that we only detect OPs and CSTs and it is very likely that packers are using way more tricks. The first constatation is that both algorithms scale well on most samples and succeed in finding both OPs and CSTs.

Anomalies Among the 33 packers, three of them (`Enigma, svk, Themida`) reached the 10M instructions and were consequently not analysed. Similarly, `Molebox, Mystic` and `Setisoft` led the analysis to fail by filling the memory. Another anomaly is raised by `ACProtect` and `Crypter`. The trace was successfully retrieved but the machine hostname did not appear on the terminal during instrumentation which might indicate that the binary was not well packed. Finally, the last anomally is due to the trace size of certain packed malware which are suspiciously small like `JD Pack`(42), `Obsidium`(21) or `Yoda's Protector`(17). The two explanations for that are either the instrumentation got detected and the process stopped, either the packers kept working in a different process/thread that the instrumentation did not managed to catch up.

Table 10.1: Packer experiment OP & Stack tampering

Packers	Static	Dynamic					Obfuscation detection					
	Size					#Wave	Opaque Predicates (k_{16})				Stack tampering	
	prg	#Tr.len	(tr.ok/host)	#proc	#th	(Codisasm/Binsec)	OK	OP	To	Covered	OK(a/d)	Viol(a/d/s)
ACProtect v2.0	101K	1.813.598	(✓,×)	1	1	634/4	74	159	0	9	0 (0/0)	48 (45/1/45)
Armadillo v3.78	460K	150.014	(×,×)	2	11	164/1	1	20	0	1	2 (2/0)	0 (0/0/0)
Aspack v2.12	10K	377.349	(✓,✓)	1	1	2/2	32	24	0	136	11 (7/0)	6 (1/4/1)
BoxedApp v3.2	903K	/	(×,×)	1	15	5/-	-	-	-	-	-	-
Crypter v1.12	45K	1.170.108	(✓,×)	-	-	-/0	263	24	0	136	125 (94/0)	78 (0/30/32)
Enigma v3.1	1,1M	10.000.000	(×,×)	-	-	-/1	-	-	-	-	-	-
EP Protector v0.3	8,6K	250	(✓,✓)	1	1	1/1	10	1	0	2	4 (2/0)	0 (0/0/0)
Expressor	13K	635.356	(✓,✓)	1	1	1/1	42	8	0	39	14 (10/0)	0 (0/0/0)
FSG v2.0	3,9K	68.987	(✓,✓)	1	1	1/1	11	1	0	14	6 (4/0)	0 (0/0/0)
JD Pack v2.0	53K	42	(×,✓)	1	1	4/0	2	0	0	0	0 (0/0)	0 (0/0/0)
Mew	2,8K	59.320	(✓,✓)	-	-	-/1	11	1	0	18	6 (4/0)	1 (0/0/0)
MoleBox	70K	5.288.567	(✓,✓)	1	1	2/2	307	60	0	128	X	X
Mystic	50K	4.569.154	(✓,✓)	1	1	3/1	X	X	X	X	X	X
Neolite v2.0	14K	42.335	(✓,✓)	1	1	1/1	95	1	0	42	9 (3/0)	0 (0/0/0)
nPack v1.1.300	11K	138.231	(✓,✓)	1	1	1/1	41	2	0	34	21 (14/0)	1 (0/0/0)
Obsidium v1364	116K	21	(×,✓)	-	-	-/0	1	0	0	0	0 (0/0)	0 (0/0/0)
Packman v1.0	5,9K	130.174	(✓,✓)	1	1	1/1	12	1	0	21	7 (4/0)	0 (0/0/0)
PE Compact v2.20	7,0K	202	(✓,✓)	1	1	3/1	11	1	0	1	4 (2/0)	0 (0/0/0)
PE Lock	21K	2.389.260	(✓,✓)	1	1	14/6	53	90	0	42	4 (3/0)	3 (0/1/0)
PE Spin v1.1	26K	/	(×,×)	1	1	79/-	-	-	-	-	-	-
Petite v2.2	12K	260.025	(×,×)	1	1	2/0	60	19	0	45	4 (1/0)	0 (0/0/0)
RLPack	6,4K	941.291	(✓,✓)	1	1	1/1	21	2	0	25	14 (8/0)	0 (0/0/0)
Setisoft v2.7.1	378K	4.040.403	(×,×)	1	5	31/4	X	X	X	X	X	X
svk 1.43	137K	10.000.000	(×,✓)	-	-	-/0	-	-	-	-	-	-
TELock v0.51	12K	406.580	(×,✓)	1	1	17/5	0	2	0	5	3 (3/0)	1 (0/1/0)
Themida v1.8	1,2M	10.000.000	(×,✓)	1	28	105/0	-	-	-	-	-	-
Upack v0.39	4,1K	711.447	(✓,✓)	1	1	2/2	11	1	0	30	7 (5/0)	1 (0/0/0)
UPX v2.90	5,5K	62.091	(✓,✓)	1	1	1/1	11	1	0	26	4 (2/0)	0 (0/0/0)
VM Protect v1.50	13K	/	(×,✓)	1	1	0/0	-	-	-	-	-	-
WinUPack	4,0K	657.473	(✓,✓)	1	1	2/2	12	1	0	33	7 (5/0)	1 (0/0/0)
Yoda's Crypter v1.3	12K	240.900	(×,✓)	1	1	3/3	38	1	0	16	4 (3/0)	9 (0/1/0)
Yoda's Protector v1.02	18K	17	(×,✓)	1	1	5/0	1	0	0	0	0 (0/0)	0 (0/0/0)

#Tr.len: execution trace length – tr.ok: whether the executed trace was gathered without exception/detection – host: whether the payload was successfully executed (printing the hostname of the machine) – #proc: number of process spawned – #th: number of threads spawned – #Wave: number of self-modification waves recorded according to Codisasm and Binsec – (a/d/s): (aligned/disaligned/single)

Detection results Except for `Molebox, Mystic` and `Setisoft` the analysis succeed on all packers. Thus, it scales well on few millions instructions traces. It managed to find 149 call/stack tampering and 420 opaque predicates. Quantifying the number of false positive would require an exhaustive verification but manual checking revealed that most of them are true positive.

10.1.4 Closer look at the results

Given the amount programs and the amount of predicate tested, it is not possible to check manually all the results, but some manual inspection yield interesting scheme and patterns described below.

Opaque predicates While some packers do not seem to really use OP with a very few OP detected (probably false positive) other packers are using it heavily like `ACProtect, Molebox` or `PE Lock`. Most predicates found are strictly exclusive conditions as shown in listing 10.1.

```
1018f7a  js  0x1018f92
1018f7c  jns  0x1018f92
```

Listing 10.1: Sample of opaque predicate ACProtect

Every pair of conditions were found `js/jns, ja/jbe, jp/jnp, jo/jno` etc. This example shows that our approach successfully detect both invariant and contextual opaque predicates. Funnily `Armadillo` uses the classic `xor, jnz` pattern as shown in listing 10.2.

```
10330ae  xor  ecx,  ecx
10330b0  jnz  0x10330ca
```

Listing 10.2: Sample of opaque predicate `Armadillo`

Call/Stack tampering Only `ACProtect` and `Crypter` are heavily using call stack tampering. ASPack also do but at a lower scale. Meanwhile, patterns used are rather trivial. The most common is `push; ret`. `ACProtect` is brutal in its approach as the first 4 instructions executed are two violations (see listing 10.3). It also uses slightly more evolved patterns with in-place violation as given in listing 10.4. Last, packers seemingly using call/stack tampering are using the `ret` instruction to perform the transition to the last self-modification layer. Packers using a tampered `ret` to perform the transition to the payload are: `ACProtect, ASPack, Crypter, Mew, nPack, PE Lock, TELock, Upack, WinUpack` and `Yoda's Crypter`.

```
0 1001000 push 16793600
1 1001005 push 16781323
2 100100a ret
3 100100b ret
```

Listing 10.3: First instructions of ACProtect

```
1004328 call  0x1004318
1004318 add  [esp],  9
100431c ret
```

Listing 10.4: Sample call/stack tampering ACProtect

The taxonomy defined in Chapter 8 helped discriminating different kinds of violations found for instance in the ASPack packer. Figure 10.2, 10.1 and 10.3 show the exemple of three different kinds of call stack tampering found in the ASPack packer. In the first exemple (Figure 10.2), the tampering is detected with labels `[violated]`, `[disaligned]` since the stack pointer read the `ret` address at the wrong offset. In the second example (Figure 10.1), the return value is modified in place. The tampering is detected with the `[violated]`, `[aligned]`, `[single]` tags as it jumps to the return site with an offset of +1. The last example (Figure 10.3), takes place between the transition of two self-modification layers and the ret is used to do the tail transition with the payload of the packer. This violation is detected with `[violated]`, `[disaligned]`, `[single]` since the analysis matches an unrelated `call` far upper in the trace with a different return site. Note that the instruction push 0x10011d7 at 10043ba is originally push 0 but is patched by the instruction at 10043a9 triggering the entrance in a new auto-modification layer when executing it.

address	len	mnemonic	comment
1004a3a	5	call 0x1004c96	//push 0x1004a3f as return site
1004c96	5	call 0x1004c9c	//push 0x1004c9b as return site
1004c9c	1	pop esi	//pop return address in esi
1004c9d	5	sub esi, 4474311	
1004ca3	1	ret	//return to 0x1004a3f

Figure 10.1: ASPack violation 1/3 (trace)

address	mnemonic	comment
1004002	call 0x100400a	//push 0x1004007 as return
1004007	.byte{invalid}	//invalid byte (cannot disassemble)
1004008	[...]	//not disassembled
100400a	pop ebp	//pop return address in ebp
100400b	inc ebp	//increment ebp
100400c	push ebp	//push back the value
100400d	ret	//jump on 0x1004008

Figure 10.2: ASPack violation 2/3 (CFG)

address	mnemonic	layer	comment
10043a9	mov [ebp+0x3a8], eax	0	//Patch push value at 10043ba*
10043af	popa	0	//restore initial program context
10043b0	jnz 0x10043ba	0	//enter last SM layer (payload)
	Enter Layer 1		
10043ba	push 0x10011d7	1	//push the address of the entrypoint
10043bf	ret	0	//use ret to jump on it
10011d7	[...]	1	//start executing payload

*(at runtime eax=10011d7 and ebp+0x3a8=10043bb)

Figure 10.3: ASPack violation 3/3 (trace)

Conclusion As suggested by the manual look at the results, most artifacts detected are true positive. False positives for opaque predicates are mostly due to the program structure making the other branch of a predicate to be infeasible. Results showed that certain packers do use opaque predicates but really trivial ones unlike the ones developped in O-LLVM.

Call/stack tampering is less employed as it is more difficult to craft such obfuscation. Only, `ACProtect`, ASPack, `Crypter` and `Yoda's Crypter` seem to use it on purpose. Nevertheless, this experiment highlighted the fact that multiple packers are using `ret` as tail transition to enter the original entrypoint. Thus, call/stack tampering is a really good candidate to detect the transition to the payload. This detection is the key-point of generic unpacker algorithm, thus using call/stack tampering analysis can be of a great help for such task as the last violation will probably be the transition to the payload.

10.2 X-Tunnel: Sednit/APT28 proxy deobfuscation

10.2.1 Context & Samples

The use-case X-Tunnel is a ciphering proxy component aiming at exfiltrating data from a machine not directly reachable from internet. This malware was used as part of various targeted attack campains (APT). The group behind these attacks is named Sednit [CCD16] but also APT28, Fancy Bear, Sofacy or Pawn Storm group. This group is active since 2006 and mainly targets geopolitical entities. Among attacks alleged to this group we can mention NATO [Tre14], EU institutions [ESE15], the White House [Tre15a], the dutch team in charge of the MH17 crash investigations [Tre15b], the german parliaments [von15], the french TV channel *TV5 Monde* [Paq15] and more recently the American Democratic National Committee (DNC) [Alp16]. As an appreciation of the group sophistication, for the single 2015 year 6 zero-days (0-days) where used, 2 in Flash, 1 in Office (RCE), 2 in Java and 1 in Windows LPE. This group also uses advanced malware tools like Rootkits, USB Command-and-Control (C&C), bootkits dropped by their custom exploit-kit and recently highly pervasive Mac OS X malware named `Komplex` [CHF16].

Table 10.2: Samples infos

	Sample #1 42DEE3[...]	Sample #2 C637E0[...]	Sample #3 99B454[...]
Obfuscated	No	Yes	Yes
Size	1.1 Mo	2.1 Mo	1.8 Mo
Date creation	25/06/2015 5h15m34	02/07/2015 9h42m44	02/11/2015 8h45m54
#functions	3039	3775	3488
#instructions	231907	505008	434143
Mean instr/fun	∼76	∼134	∼125
#imports	133	139	135

Samples This use-case is based on 3 X-Tunnel samples, kindly provided by Joan Calvet formerly at ESET Montreal [CCD16] (I warmly thank him again). These three samples have apparently been compiled at few months of difference. The particularity is that the first sample dated of the 25/06/2015 is not obfuscated while the two others are. The last sample was compiled the 02/11/2015 thus these three samples are if timestamp are corrects, somewhat covering the evolution of the software on a 5 month period. The main issue raised is:

RQ: *Is there new functionalities in the obfuscated samples ?*

Answering this question requires to be able to analyse the obfuscated binaries. Table 10.2 gives the basic information about the samples. As suggested by the results, the number of instructions in obfuscated samples is almost twice as big as the original sample while the number of imported functions remains almost constant.

Analysis context A quick code gazing reveals the presence of opaque predicates in the code. Unlike previous case-studies, the analysis have to be performed statically for the following reasons:

- as the malware is a network component, it requires to connect to the C&C server (which we are definitely not eager to, with such attackers);
- as it listens on the network, all branching conditions to cover the program would be network-event based thus unreliable and more hardly reproducible (which would also require infected client to connect to xtunnel);
- X-Tunnel does not look to use any self-modification obfuscation or neaty tricks to hamper the disassembly. Thus the whole disassembled code is available.

For these reasons running the analysis statically will be easier and will provide a full coverage of the program. In addition, the opaque predicate is performed fully symbolically (does not use runtime values) thus running the analysis statically does not change the inner working of the analyse.

Goal The goal of this analysis is to detect and remove all opaque predicates along with their dead-code in order to find the potential new functionality afterward. This is a two step process (1) identifying the obfuscated code and pruning it, (2) finding the potential functionalities. We focus in this analysis on the first step which is a prerequisite for the second. Thus, we need to find a way to simplify the CFG in order to make the second step possible.

10.2.2 Approach & Analysis setup

Analysis setup This analysis will be static SE but working like a DSE analysis as opaque predicates analysis is fully symbolic. As described hereafter, all the process will be driven by IDASEC that will submit all sub-traces to BINSEC running in server mode. The obfuscation seems to have been done on a per function basis so the analysis will also be performed on a per function basis. Taking advantage of benchmarks previously done, we will use k_{16} as the bound value for the BB-DSE. The solver used is `Z3` with a 6s timeout.

BINSEC From the BINSEC perspective, the analysis is slightly different. Indeed in DSE when reaching a conditional branches, one of the two branches is necessarily taken. Thus solely the other branch need to be checked. In static SE both branches need to be checked to find out if the predicate is opaque. Also, different from DSE, if both branch are UNSAT we mark the predicate as LIKELY-OPAQUE since the unsatisfiability is necessarily due to path constraints not satisfiable which mean the predicate is unreachable.

IDASEC Most of the computation tasks will be driven by IDASEC as it has the complete view of the program and its CFG. The first thing performed is the enumeration of paths leading to every conditional jump of the program. For a conditional jump to check, the choice has been made to compute the shortest path from the function entry point as a reference path. This path is then used to create an artificial execution trace sent to BINSEC for solving. The shortest path computation is performed by a classic dijkstra algorithm performed on basic blocks. This allows to know for any instructions its depth in the function. IDASEC also performs a basic predicate synthesis to identify the predicate used and also an unobfuscated reduced CFG extraction based functionality. The two are described below. For convenience, the analysis in IDASEC allows to perform the analysis with different levels of granularity:

- on a predicate
- on a function
- one the whole program

Exploiting results Results are processed by IDASEC which performs further computation to extract relevant information. These computations require the two following components developped for that purpose:

- **Internal SMT representation** A tiny SMT representation is embedded allowing to perform computation on the generated formula. As the formula represents the logic relation of a predicate it allows to perform semantic manipulation. It is used to perform the dependancy analysis and to retrieve all the spurious instructions involved in the predicate computation. The predicate synthesis also makes use of this functionality.

- **Internal CFG representation** The CFG given by IDA is superficial and not extendable. Thus, IDASEC holds an internal representation providing all the information about paths to a specific address. It also can provide a safe path to a specific address ensuring no possible back edges within that path. Lastly, this CFG allows to tag basic blocks and instructions if user defined information.

Post-analysis processing IDASEC performs three post-analysis computations on results, (1) liveness propagation, (2) reduced CFG extraction and (3) predicate synthesis. Here are some insight on the inner working of the different algorithms:

- **Predicate "synthesis"** This algorithm allows to retrieve the instructions involved in the computation of a predicate and gives an higher level view of the predicate used which allows to quickly identify OPs. We call *spurious instructions* all instructions involved in the computation of an opaque predicate. Synthesis is improperly used here as it does not use formal synthesis algorithm usually denoted by this term. This algorithm works on the SMT representation of the predicate. For a conditional jump it looks for the two parameters used in the comparison associated and recursively replace them by their definition. Some normalization is also performed for instance $x \times x$ is replace by x^2. At the end of the recursion all register or memory cells are replaced by placeholders (x, y, z etc) to obtain the generic predicate;

- **Liveness propagation** This analysis is used to mark basic blocks and instructions with annotations namely *alive*, *dead* and *spurious*. This analysis uses OP results to propagate from the entry point the liveness of basic blocks, marking them with theirs status. For each function, it starts from the entry point and follows the CFG. When reaching a good condition, it follows both branches, otherwise it follows only the alive branch and remove the dead edge in the CFG. All unmarked blocks are consequently tagged dead. Finally, all alive instructions in basic blocks are marked alive except for *spurious* instructions which are marked SPURIOUS;

- **Reduced CFG extraction** This analysis aims at recovering a readable and unobfuscated CFG of a function. Using information computed by the liveness propagation, the CFG extraction algorithm remove all dead basic blocks of the CFG and all edges pointing to them. After this operation, it remains only basic blocks containing legitimiate and *spurious* instructions. Then, the algorithm will strip *spurious* instructions and concatenate within the same basic block all instructions having a single predecessor. This operation is performed while not encountering an instruction with multiple successors. This works well as the liveness propagation will have removed dead edges.

10.2.3 Results

Execution time The analyses were performed on the whole program of the two obfuscated samples. Table 10.3 reports the execution time. Less than two hour is clearly acceptable for samples of this size. The predicate synthesis takes a non-negligible amount of time but it was not especially designed to be efficient. The 10 predicates per second should be put in perspective as it includes sub-path creation, solving for both left and right branches and also network communications.

Table 10.3: Execution time

(in s)	#preds	DSE	Synthesis	Total	pred/sec(avg)
C637 (#1)	34505	57m36 (3456.00s)	48m33 (2913.22s)	1h46m (6369.29s)	∼10
99B4 (#2)	30147	50m59 (3059.80s)	40m54 (2454.9s)	1h31m (5514.70s)	∼10

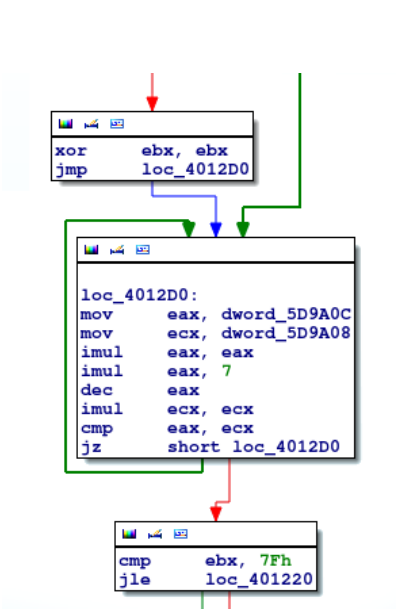
OP diversity Each sample presents a very low diversity of predicates used. Indeed solely $7x^2 - 1 \neq x^2$ and $\frac{2}{x^2+1} \neq y^2 + 3$ where found to be opaques. It is worth noting that the later was new to our knowledge and has never been seen elsewhere. Table 10.4 sums up the distribution of the different predicates. The amount of predicates and their distribution supports the idea that they were inserted automatically and chosen randomly.

Table 10.4: Opaque predicates variety

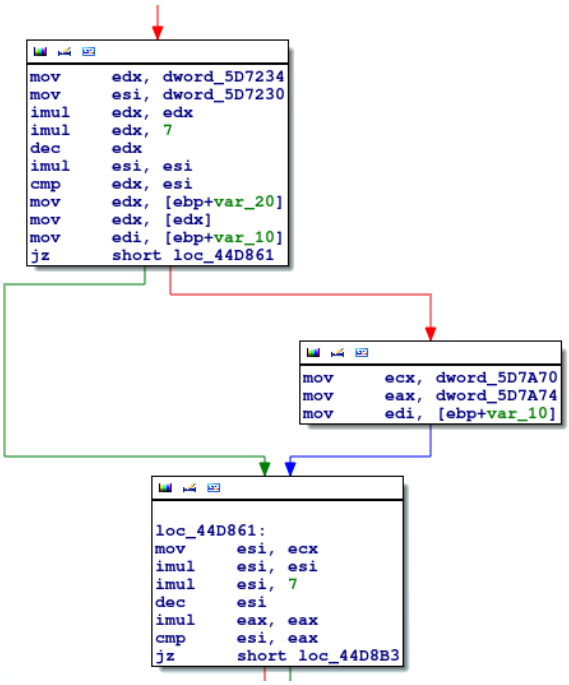
	OP #1 $7y^2 - 1 \neq x^2$	**OP #2** $\frac{2}{x^2+1} \neq y^2 + 3$
C637 (#1)	6016 (49.02%)	6257 (50.98%)
99B4 (#2)	4618 (45.37%)	5560 (54.62%)

Detection results As the diversity of opaques predicates is very low, we are able to determine, with quite a good precision, the amount of false negatives and false positives based on the predicate synthesized. If a predicate matches one of the two OP and was detected OK, then we considered it false negative (respectively false positive). Results are given in Table 10.5. From one sample to the other results are similar except that one of them yielded a timeout. The detection rate is satisfactory as false negatives only represent 2.4% of all predicates but conversely 9.7 to 10.5% of false positive are wrongly tagged opaque. We only test on path (the shortest) for each predicates. As consequence, most false positives are due to wrong path the selection. An exemple is shown in Figure 10.4a where the shortest-path selected going through the `xor ebx, ebx` which makes the

predicate `cmp ebx, 0x7f` to be always false. Also, we marked LIKELY-OPAQUE, predicates for which both branch conditions evaluate to UNSAT. This is due to the high density of opaque predicates and are mostly occuring when another OP is in the path predicate. Indeed, even though the backward bound is only 16 it is frequent that the path to a legitimate predicate goes through another opaque predicates that makes the path predicate UNSAT. Figure 10.4b illustrates this issue where the `jz` is flagged opaque because the shortest-path has select the green edge which is the dead branch of another opaque predicate.



(a) False positive in C637



(b) Likely opaque in C637

Figure 10.4: Samples of false positives / Likely predicates

Table 10.5: Opaque predicates evaluation

	#pred	OK		OP		Likely		To
		OK	FN	OP	FP	OK	OP	
C637 (#1)	34505	17196 (49.8%)	1046 (3.0%)	11973 (34.7%)	2968 (8.6%)	1156 (3.4%)	165 (0.4%)	1
99B4 (#2)	30147	16148 (53.7%)	914 (3.0%)	9790 (32.5%)	2543 (8.4%)	606 (2.0%)	146 (0.5%)	0

likely: predicates were both branches were UNSAT

Dependency evaluation As seen above, a large k bound can lead to false positive due to nested opaque predicates while in the meantime a low bound miss some predicates.

An example is given in Figure 10.5 where a bound of 24 would have been required to include all predicate dependencies. With k=16 we miss the `imul ebx,ebx` that would have enforced enough the constraints to detect the predicate. Finding the right balance is still an important issue, but results with 11311 OP detected against 808 false negative tend to confirm that such low bound is enough. Across the two samples, the maximum distance between a predicate and its variable definition where 230 for C637 and 148 for 99B4. Annexe C shows an example of predicates with a distance of 43 between variables declaration and their usage. Still, the average computed on all the OPs yield an average of 8.7. The results obtained are satisfactory in the way that they open the way to further manual triage on doubtful predicates.

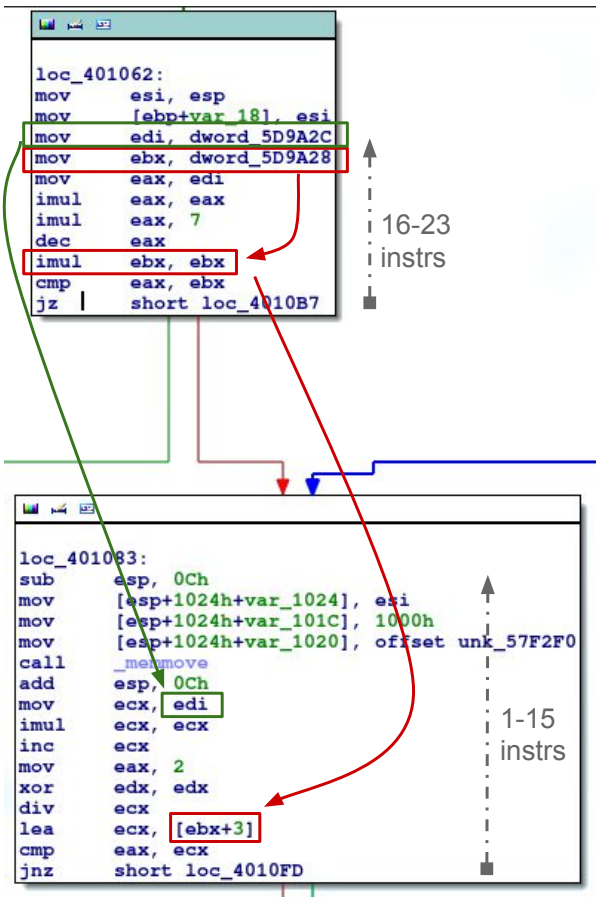


Figure 10.5: False negative in 99B4

Difference with O-LLVM The two obfuscated program structures present great similarities with O-LLVM and it has undoubtedly been obfuscated using similar technologies. Yet, interesting differences are to be emphasized. Firstly, there is more interleaving between the payload and the OPs computation. What appeared to be meaningful instruction is often encountered within the predicate computation. Secondly, while O-LLVM opaque predicates are really local to the basic blocks there are here some code sharing between predicates as shown in Figure 10.5 where `ebx` is used in two opaque predicates. As

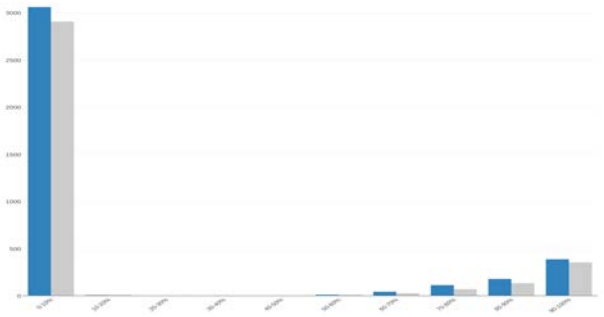
a consequence, predicates are not fully independent from one another. In the same vein, this obfuscator uses local function variables to store temporary results at the begining of the function for later usage in opaque predicates. This has in consequence to increase the depth of the dependance chain and to complexify the detection.

Obfuscation distribution Surprisingly, not all functions are obfuscated. Conversely, only few functions appeared to be obfuscated. Figure 10.6 shows the distribution of functions considering their level of obfuscation. For each function the level of obfuscation is the percentage of opaque predicates among all conditional jumps. Considering that the obfuscation is generally used to hide the critical parts of the program it can be helpful to focus on these functions. X-Tunnel include various libraries statically linked and most notably OpenSSL. Thus authors might have considered useless to obfuscated these functions.

%pred obfu	C637 (#1)	99B4 (#2)
0-10%	3060	2906
10-20%	4	5
20-50%	0	0
50-60%	5	7
60-70%	39	23
70-80%	110	67
80-90%	175	131
90-100%	384	351

distribution of the percentage of obfuscation per function

(a) Distribution of function obfuscation level

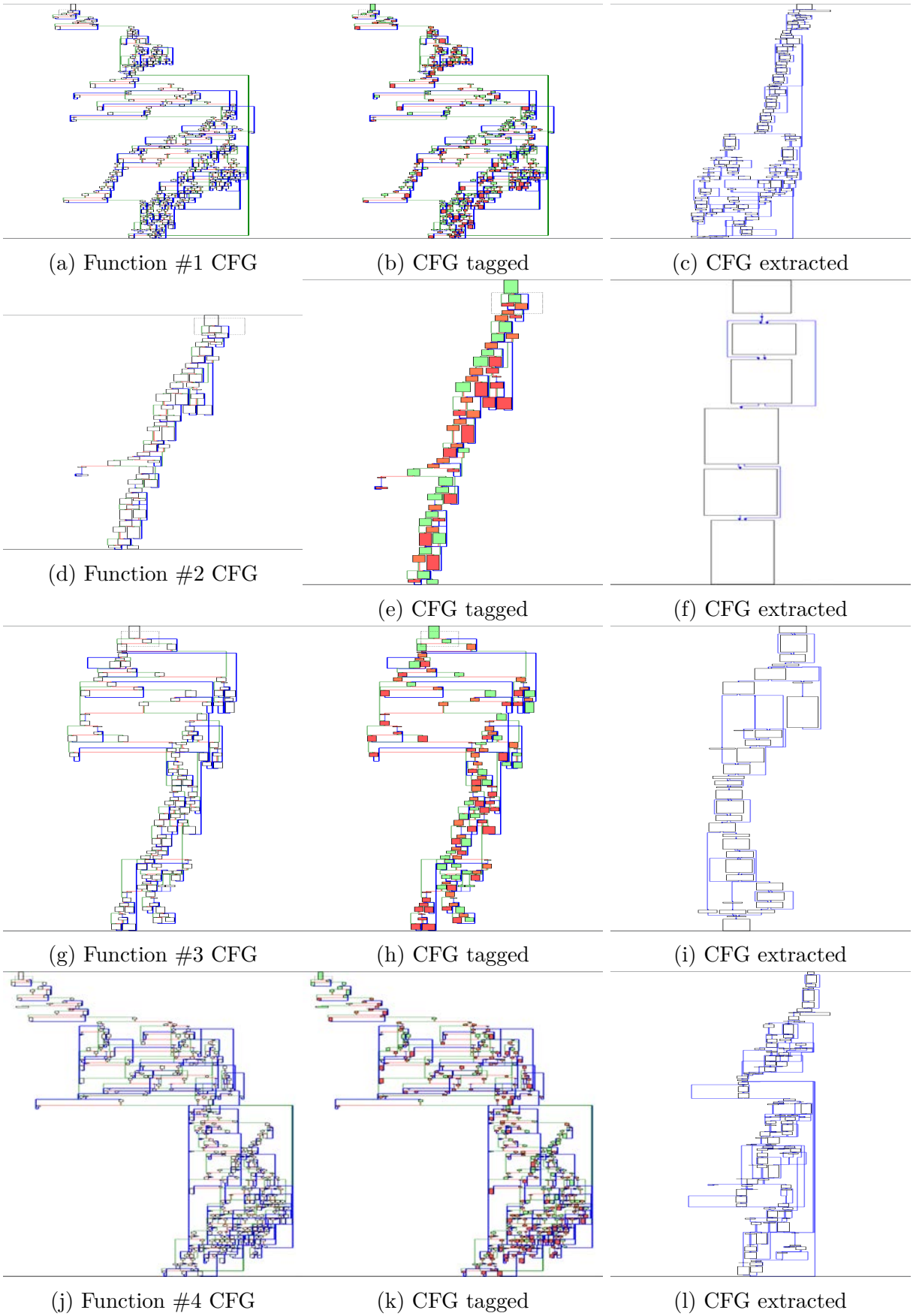


(b) Graph distribution of obfuscation

Figure 10.6: Fonction obfuscation

Code simplification After detecting OPs, the next step is to simplify the code by removing dead basic blocks and spurious instructions. Table 10.6 shows the number of instructions classified based on their status. Dead code alone, represents 1/4 of all the program instructions. Computing the difference with the original non-obfuscated program shows a very low difference. Therefore, the simplification pass allowed to retrieve a program which is roughly the size of the original one. The difference is highly likely to be due to the false negatives or missed *spurious* instructions.

Reduced CFG Extraction The extraction of the reduced CFG can be made on purpose for a chosen function. It uses the liveness data to extract the unobfuscated CFG. Figure 10.7 shows different original CFG, with their tagged version (red:dead, orange:spurious) and extracted version. The extracted versions are not executable and still contain noise and thus some oddities in the flow of the CFG. Although, it makes possible reading the CFG while it was simply not possible on the obfuscated version.



(a) Function #1 CFG (b) CFG tagged (c) CFG extracted

(d) Function #2 CFG (e) CFG tagged (f) CFG extracted

(g) Function #3 CFG (h) CFG tagged (i) CFG extracted

(j) Function #4 CFG (k) CFG tagged (l) CFG extracted

 Figure 10.7: Examples of CFG extraction performed on 4 functions

Table 10.6: Code simplification results

	#instr	#alive	#dead	#spurious	alive-(42DE insts) (231,907)
C637 (#1)	507,206	279,483 (55%)	121,794 (24%)	103,731 (20%)	47,576
99B4 (#2)	436,598	241,177 (55%)	113.764 (26%)	79,202 (18%)	9,270

10.2.4 Conclusion & Futur improvment

Results conclusion This obfuscation is relatively sophisticated compared with existing opaque predicates found ever since. It successfully manages to spread the data dependency accross a function so that some predicates cannot be solved locally at the basic block level. Hopefully, this is not a general practice across predicates so that the BB-DSE works very well in the general case. The main issue of the obfuscation is the low diversity of opaque predicates in the way that some pattern matching can come in relay of symbolic approaches to classify a posteriori false positives and false negatives. Finally, this use-case emphasizes the complexity of choosing the appropriate k bound to maximize the detection and to limitate the false positives/negatives rates.

Possible improvements Various improvements are possible in order to reduce false positives and false negatives. The first, is for a given predicate to compute the opacity with more k-paths which would reduce the risk of having selected the wrong one. The second, would be to compute on the fly the CFG simplification to remove dead edges and basic blocks as soon as possible. This would have in effect to improve the shortest path computation for a more accurate one. Last, an idea is to compute a safe path ensuring it does not go through an already detected opaque predicate or function call. Last, performing a taint on data would enable to detect missing data dependencies while keeping the scalability of the backward-bounded approach without being hampered by path constraints making formulas UNSAT for both branches.

Towarding finding new functionalities This case-study, showed how to locate highly obfuscated functions (more likely of interest) and how to extract a readable CFG version of them. In order to answer the initial question, some similarity algorithms should be computed between the non-obfuscated and the extracted version. This would help to find the differences and potential new functionalities. The initial question is *"is there new functionalities in obfuscated samples ?"*. We do not give a final answer but the low difference of instructions between the non-obfuscated and the extracted versions supports the idea that it is very unlikely that new functionalities would have been added in the obfuscated samples.

Part V

Conclusion

Chapter 11

Conclusion

We have studied in this thesis the application of different formal methods to address obfuscation problematics. The focus has been given to dynamic symbolic execution applied to two unfeasibility based obfuscation, namely opaque predicates and call/stack tampering. In the scope of this study we formalized and developped two DSE variants aiming to scale on obfuscated codes. In support we designed different optimisations. This work has been validated on two significant case studies especially packers and the X-Tunnel malware.

11.1 Contributions summary

Flexible Dynamic Symbolic Execution The main contribution of this thesis is the formalization, implementation and testing of an enhanced path predicate computation algorithm integrating concretization and symbolization in a flexible way. All that was facilitated by the conception of a meta-language CSml allowing to specify a wide range of different policies and thus behavior for concrete data. This modularity did permit to benchmark various behaviors highlighting the difference between them and emphasizing the impact of concretization and symbolization on the solving time of the path predicate. To further reduce the solving time different formula optimizations were formalized and developped of which the most notable is the STMC-RoW. Various experiments and benchmarks showed the usefulness of these optimizations to scale on larger formulas.

bb-dse Another contribution to the dynamic symbolic execution field of research is the backward-bounded DSE algorithm. This algorithm is scalable, robust and precise for solving infeasibility questions brought by obfuscation. This thesis has demonstrated the benefit of the method for several realistic kinds of obfuscations, such as opaque predicates and call stack tampering. While this algorithm does not supersede existing forward DSE approaches, it complements them by addressing infeasibility questions.

These two algorithms are the answer to *"How to make analysis algorithms scaling on real-world obfuscated malwares"* stated in the introduction.

Obfuscation detection Deobfuscation is the cornerstone of my thesis subject and was addressed by focusing on two obfuscations, opaque predicates and call/stack tampering. We proposed for each one, a complete taxonomy along with a detection method based on our DSE algorithms. This study provides a throughout study of opaque predicates and call/stack tampering and shows how the BB-DSE algorithm has successfully been used on packers and on the real-world malware X-TUNNEL. It is worth nothin that call/stack tampering obfuscation have rarely been addressed in state-of-the art research. Such analyses, also unveiled interested properties. For instance it showed that call/stack tampering was able to detect the tail transition from the packer to the payload for packers using `ret` for making this jump.

Sparse disassembly Beyond simple detection, as one of the main contribution, my thesis formalizes a new disassembly algorithm *sparse disasembly* lifting obfuscation information computed by DSE in the disassembly process. It empower a dynamic disassembly with a static disassembly guided and improved with obfuscation related data. The few benchmarks performed, yielded promising results. This work paves the way for robust, correct and almost complete semantic-aware disassembly tools for obfuscated binaries.

Experimental Validation The scalability and the robustness of these algorithms and methods have been validated on commercial packers. Moreover, these techniques found some application beyond the reverse-engineering and the binary analysis field. Indeed, a forward/backward combination approach detailed in 9.4 was successfully used for detecting infeasible test requirements at source-level for software testing purposes. The last validation of my thesis work was performed on the X-TUNNEL malware. It showed a concrete example replying to the initial question: "How to locate obfuscation, and how to take it off to make the code more readable and malleable for the reverse-engineer ?". From the localization to the CFG extraction, all steps were performed to remove the obfuscation and to make the CFG usable for the reverse-engineer. Besides that, it showed that algorithms theorized in Part II are applicable and successful on governement-grade malware.

11.2 Community contributions

A key idea of this thesis, was to make the algorithm and detection method usable in practice for the reverse-engineering community in order to bridge the gap between academic research and industrial problems. For that, all algorithms were implemented in open-source software namely PINSEC, BINSEC/SE and IDASEC. These three tools provide respectively a dynamic instrumentation tool to gather concrete behaviors of malware, a tool to perform symbolic execution and an IDA plugin allowing to use all these algorithms in a user-friendly manner.

These work and implementation were presented in non-acadamic conferences especially RMLL/Sec [BD16] and BlackHat Europe 2016 in London [DB16a]. The aim of such

initiatives was to fill in the gap between academic research and the reverse-engineering community by providing them both practical approaches and functional tools.

11.3 Perspectives

This thesis opens the way to many applications and further improvements. From a purely technical point of view we can improve dynamic analysis by adding more anti-debug or addressing more obfuscation schemes but the next step of the work made in this thesis would be to integrate the results of analyses with morphological analysis algorithms developped at the LORIA aiming at providing better and more accurate signatures for malwares.

Formal methods and more especially symbolic approaches provided very good results with dynamic symbolic execution, but another promising usage of symbolic approaches for software deobfuscation is instruction synthesis. This technique showed very good results for optimization matters and very few works have focused on using such approach for deobfuscation.

Over the past few years, the interest in automated binary analysis grew tremendously with the appearance of challenges like the Cyber Grand Challenge [DAR16] (CGC) and consequently some baseline benchmarks for binary analysis tools [Gui16]. The CGC financed by the DARPA greatly intensified the research on various aspect of binary analysis like dynamic instrumentation, binary translation, abstract interpretation and symbolic execution. My thesis humbly contributes on several of these aspects and showed that formal methods can successfully be used for obfuscation related problematics. Especially it tries to bring semantic-aware analyses to the field of malware analysis. Hopefully, this thesis will guide and incitate futur researches aiming at building more accurate malware detection algorithms.

Acronyms

ABI Application Binary Interface. i, 54, 151, *Glossary:* Application Binary Interface

ACSL ANSI/ISO C Specification Language. i, 125, 151, *Glossary:* ANSI/ISO C Specification Language

ANR Agence Nationale de la Recherche. i, 4, 151, *Glossary:* Agence Nationale de la Recherche

API Application Programming Interface. i, 151, *Glossary:* Application Programming Interface

APT Advanced Persistent Threat. i, 134, 151, *Glossary:* Advanced Persistent Threat

AST Abstract Syntax Tree. i, 125

C&C Command-and-Control. i, 134, 135, 151, *Glossary:* Command-and-Control

CFG Control Flow Graph. i, 6, 11, 12, 151, *Glossary:* Control Flow Graph

CGC Cyber Grand Challenge. i, 151, *Glossary:* Cyber Grand Challenge

CST Call/Stack Tampering. i

DARPA Defense Advanced Research Projects Agency. i, 149

DBA Dynamic Bitvector Automata. i, xxi, 151, *Glossary:* Dynamic Bitvector Automata

DBI Dynamic Binary Instrumentation. i, 68, 151, *Glossary:* Dynamic Binary Instrumentation

DNC Democratic National Committee. i, 134, 151, *Glossary:* Democratic National Committee

DPLL Davis–Putnam–Logemann–Loveland. i, 73, 151, *Glossary:* Davis–Putnam–Logemann–Loveland

DRM Digital Right Management. i, 11, 151, *Glossary:* Digital Right Management

LHS Laboratoire de Haute Sécurité. i, 4, 151, *Glossary:* Laboratoire de Haute Sécurité

LLVM Low-Level Virtual Machine. i, 68, 151, *Glossary:* Low-Level Virtual Machine

LPE Local Privilege Escalation. i, 134, 151, *Glossary:* Local Privilege Escalation

LTR left-to-right. i, 53, 151, *Glossary:* left-to-right

MBA Mixed-Boolean Arithmetic. i, 13, 151, *Glossary:* Mixed-Boolean Arithmetic

NATO North Atlantic Treaty Organization. i, 134

OP Opaque Predicates. i, 101

OTS Off-The-Shelf. i, 25, 152, *Glossary:* Off-The-Shelf

PLC Programmable Logic Controller. i, 3, 152, *Glossary:* Programmable Logic Controller

QF_ABV Qantifier-Free Array and Bitvectors. i, 92

QF_BV Qantifier-Free Bitvectors. i, 92

RCE Remote Code Execution. i, 134

ROP Return-Oriented Programming. i, 19, 152, *Glossary:* Return-Oriented Programming

RTL right-to-left. i, 53, 152, *Glossary:* right-to-left

SE Symbolic Execution. i, 21, 25

SMT Satisfiability Modulo Theories. i, 21, 73, 137, 152, *Glossary:* Satisfiability Modulo Theories

TCG Tiny Code Generator. i, 68, 152, *Glossary:* Tiny Code Generator

TSC Time Stamp Counter. i, 55, 152, *Glossary:* Time Stamp Counter

UaF Use-After-Free. i, 7, 122, 152, *Glossary:* Use-After-Free

VPC Virtual Private Counter. i, 18

Glossary

X-Tunnel Proxy component aiming at connecting machines not primarly connect on internet to access it. Used as part of targeted attacks of APT28, it is use to exfiltrate data by ciphered channels. i

Advanced Persistent Threat represent targeted attacks against certain organization or person performed as a long term process. Such attacks usually involve a whole ecosystem of malware, infrastructure and vulnerability exploits.. i, 151

Agence Nationale de la Recherche French organization for academic research. i, 4, 151

ANSI/ISO C Specification Language is a behavioral specification language for C programs. It allows to express a wide range of properties and is defined as a formal language allowing to perform various analyzes on it. i, 125, 151

anti-debug group all the techniques aiming at preventing the debugging of the software. It can be checking if debugged, attaching to itself etc. i, 14

anti-tampering group all the techniques aiming at preventing the static or dynamic modification of the software. i, 14, 18

anti-VM group all the techniques aiming at preventing the software of being run in a virtual machine (that could indicate being analyzed). i, 14

Application Binary Interface defines the interface between two modules, within the kernel or between use-space programs and kernel. i, 54, 151

Application Programming Interface represent a set of routines definitions, protocols or formats used to facilitate the interaction between two different piece of software. i, 151

APT28 Nickname given by FireEye to identify a criminal group acting since 2007. The same group is identified under other nickname like “Sofacy”, “Sednit” or “Pawn Storm”. i, 8, 134

bootkit Malicious, software which task is to infect the boot process by interleaving the transition from boot process to the system boot with its own startup. i, 3, 134

bots A bot, is a machine that runs autonomously under the control of another machine or human. It is characterized by its grouping in a network where all individual nodes perform the same taks. i, 3

Command-and-Control is a remote machine on which an infected machine connects to receive commands and instructions. i, 134, 151

Control Flow Graph Graph structure representing the program where each node is basic block and each edges are the possible flow from a block to another. i, 6, 151

Cyber Grand Challenge is a challenge created by DARPA to develop fully-automated defense reasoning systems that can discover prove and correct software flaws. Designed as a challenge it involved various teams competing against each other in a "capture the flag" game. i, 151

Davis–Putnam–Logemann–Loveland is a backtracking-based search algorithm for deciding the satisfiability of propositional logic formula. i, 151

Democratic National Committee is the formal governing body of the United States Democratic Party. i, 134, 151

Digital Right Management General term grouping all access controls techniques enforcing the access to a resource to a user. i, 11, 151

droppers A dropper, is a malicious software used to drop a file on the machine it is running on. It is usually employed as a first stage infection vector to drop another malicious executable that fits the infected machine. i, 3

Dynamic Binary Instrumentation Dynamic Analysis aiming at gather all runtimes values of the program. i, 68, 151

Dynamic Bitvector Automata intermediate representation used internally in Binsec. i, xxi, 151

exploit-kit is a software kit designed to run on web servers. Clients visiting the website would be scanned in order to find a vulnerable browser or plugin in order to infect the machine. i, 134

Frama-C "Framework d'analyse de C" is an open-source platform for C analysis. Working on the CIL representation it embeds by default a value analysis by abstract interpretation and weakest-precondition calculus algorithm. i, 4, 8, 125

Laboratoire de Haute Sécurité Inria laboratories dedicated to computer security research. i, 4, 151

left-to-right designate the way parameters are transmitted by the calling convention. i, 53, 151

Local Privilege Escalation is the mean by which a non-priviledged user gain priviledged access to a machine. i, 151

Low-Level Virtual Machine is a tool-chain providing an intermediate representation LLVM-IR allowing to develop compiler front-ends and back-ends. i, 151

Malware Software developped with the intent to harm a computer system by altering its availability, confidentiality or integrity. i, 3

Mixed-Boolean Arithmetic obfuscation used to obfuscated constant values, or simple condition into linear arithmetic operations. i, 13, 151

Off-The-Shelf idiomatic expression to represent an object (or software) ready to use. i, 25, 152

packer A packer is a program aiming at embedding a program in another. The goal is to compress the packed binary or to protect its content from analysis. Used for intellectual property such programs are also used to hide malwares. i, 8, 18

polymorphism generic term, grouping all software changing of form between two execution or during the execution. It includes self-modification. i, 15, 18

Programmable Logic Controller Industrial digital computer used to control the manufacturing or the operation of physical systems (robots, assembly lines etc). i, 3, 152

race condition specific event or output dependent on the sequencing or timing of uncontrollable events. i, xxx

ransomware specific malware ciphering files of the infected machine and requesting the payment of a ransom to decipher files. i, 3

Return-Oriented Programming exploitation technique aiming at bypass not executable stack by using ret instructions of the different functions to encode a shellcode or a specific trampoline. i, 152

right-to-left designate the way parameters are transmitted by the calling convention. i, 53, 152

rootkits Malicious, software running with the kernel permissions. i, 3

Satisfiability Modulo Theories is a decision problem for logical formulas with respect to different theories expressed in first-order logic. i, 21, 152

secure trigger protection technique, aiming at revealing or deciphering the payload of a software only under very specific settings. The setting or environment triggering the deciphering is designed to be hardly guessed and is thus secure. i, 15

Sednit Nickname given by ESET to identify the “Sofacy”, “APT28”, “Fancy Bear” criminal group. i, 8, 134

SMTLIB2 standard format used in SMT to express formulas in the any theories. Built to be modular it allows a standard input format across solvers. i, 73, 75

Time Stamp Counter is a `x86` register holding the count of cycles since the begining of the program.. i, 152

Tiny Code Generator is a code generator transforming native machine instruction into TCG instructions in order to be emulated in a portable and architecture independent way. i, 152

Use-After-Free vulnerability occuring when using a resource that has previously been freed from the memory. This improper memory access allows in some cases the control on the program. i, 7, 122, 152

watermarking mean of hiding information in another medium. It is used in software protection to hide software keys etc. i, 11

zero-day A zero-day vulnerability is a vulnerability that has never been publicly disclosed ever since. i, 3, 134

Bibliography

[12] "SMT Solvers in Software Security". In: *Presented as Part of the 6th USENIX Workshop on Offensive Technologies*. Berkeley, CA: USENIX, 2012 (cit. on p. 60).

[Alp16] D. Alperovitch. *Bears in the Midst: Intrusion into the Democratic National Committee*. https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/. June 2016 (cit. on pp. xix, xxxii, 134).

[ANR16] ANR Binsec. *BINSEC, Binary Code Analysis for Security*. http://binsec.gforge.inria.fr/. 2016 (cit. on p. 6).

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008 (cit. on p. 126).

[Arc16] B. Archer. *Radamsa*. https://github.com/aoh/radamsa. Sept. 2016 (cit. on p. 123).

[Avg+11] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. 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*. 2011 (cit. on p. 21).

[Avg+14] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing Symbolic Execution with Veritesting". en. In: ACM Press, 2014, pp. 1083–1094 (cit. on p. 21).

[Avg+16] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. "Enhancing Symbolic Execution with Veritesting". In: *Commun. ACM* 59.6 (2016), pp. 93–100 (cit. on p. 43).

[Bar+12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. "On the (Im)possibility of Obfuscating Programs". In: *J. ACM* 59.2 (May 2012), 6:1–6:48 (cit. on pp. xix, 11).

[Bar+13] S. Bardin, P. Baufreton, N. Cornuet, P. Herrmann, and S. Labbe. "Binary-Level Testing of Embedded Programs". In: IEEE, July 2013, pp. 11–20 (cit. on p. 72).

[Bar+14] S. Bardin, O. Chebaro, M. Delahaye, and N. Kosmatov. "An All-in-One Toolkit for Automated White-Box Testing". In: *TAP*. York, UK: Springer, 2014 (cit. on pp. xxviii, xxix).

[Bar+15a] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J.-Y. Marion. "Sound and Quasi-Complete Detection of Infeasible Test Requirements". In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–10 (cit. on p. 7).

[Bar+15b] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J.-Y. Marion. "Sound and Quasi-Complete Detection of Infeasible Test Requirements". In: *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. 2015, pp. 1–10 (cit. on pp. 8, 123, 126).

[BD16] S. Bardin and R. David. *Binsec: Binary-Level Semantic Analysis to the Rescue*. https://sec2016.rmll.info/program/#binsec. May 2016 (cit. on pp. xxxv, 148).

[Bel16] F. Bellard. *Qemu, Open Source Processor Emulator*. wiki.qemu.org/Manual. 2016 (cit. on p. 68).

[BFT16] C. Barrett, P. Fontaine, and C. Tinelli. "The Satisfiability Modulo Theories Library (SMT-LIB)". In: (2016) (cit. on p. 73).

[BH11] S. Bardin and P. Herrmann. "OSMOSE: Automatic Structural Testing of Executables". In: *Softw. Test., Verif. Reliab.* 21.1 (2011), pp. 29–54 (cit. on pp. 21, 40).

[BHV11] S. Bardin, P. Herrmann, and F. Védrine. "Refinement-Based CFG Reconstruction from Unstructured Programs". In: *VMCAI*. 2011, pp. 54–69 (cit. on pp. xvii, 4, 57).

[Bie+99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. "Symbolic Model Checking without BDDs". In: *TACAS 1999*. Springer, 1999 (cit. on p. 58).

[BKC13] S. Bardin, N. Kosmatov, and F. Cheynier. "Efficient Leverage of Symbolic Execution to Advanced Coverage Criteria". In: *CoRR* abs/1308.4045 (2013) (cit. on p. 124).

[BKC14] S. Bardin, N. Kosmatov, and F. Cheynier. "Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria". In: *ICST*. Cleveland, OH, USA: IEEE, 2014 (cit. on p. 124).

[Bon+15] G. Bonfante, J. M. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry. "CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. 2015, pp. 745–756 (cit. on pp. xxxi, 13, 59, 117, 129, 169).

[BR10] G. Balakrishnan and T. Reps. "WYSINWYX: What You See Is Not What You eXecute". In: *ACM Transactions on Programming Languages and Systems* 32.6 (Aug. 2010), pp. 1–84 (cit. on pp. xvii, 4, 57, 85).

[Bru+07] D. Brumley et al. *BitScope: Automatically Dissecting Malicious Binaries*. Tech. rep. In CMU-CS-07-133, 2007 (cit. on pp. xx, 19, 21).

[Bru+11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 2011, pp. 463–469 (cit. on p. 22).

[BZA12] D. Bruening, Q. Zhao, and S. P. Amarasinghe. "Transparent Dynamic Instrumentation". In: *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (Co-Located with ASPLOS 2012)*. 2012, pp. 133–144 (cit. on p. 68).

[Cad+08] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: Automatically Generating Inputs of Death". In: *ACM Trans. Inf. Syst. Secur.* 12.2 (2008) (cit. on pp. 33, 43).

[Cad+11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. "Symbolic Execution for Software Testing in Practice: Preliminary Assessment". In: ACM Press, 2011, p. 1066 (cit. on p. 21).

[Cas+13] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song. "HI-CFG: Construction by Binary Analysis and Application to Attack Polymorphism". In: *Computer Security – ESORICS 2013*. Ed. by D. Hutchison et al. Vol. 8134. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–181 (cit. on p. 70).

[CCD16] J. Calvet, J. Campos, and T. Dupuy. *Visiting The Bear Den, A Journey in the Land of (Cyber-)Espionage*. RECON 2016, Montreal, 17/06/16 (cit. on pp. xxxii, 8, 134, 135).

[CDE08] C. Cadar, D. Dunbar, and D. R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008, pp. 209–224 (cit. on p. 22).

[CFM12] J. Calvet, J. M. Fernandez, and J.-Y. Marion. "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs". en. In: ACM Press, 2012, p. 169 (cit. on p. 18).

[Cha+12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. "Unleashing Mayhem on Binary Code". In: IEEE, May 2012, pp. 380–394 (cit. on pp. 22, 33, 43, 70).

[Che+14] O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. "Behind the Scenes in SANTE: A Combination of Static and Dynamic Analyses". In: *Autom. Softw. Eng.* 21.1 (2014), pp. 107–143 (cit. on p. 73).

[CHF16] D. Creus, T. Halfpop, and R. Falcone. *Sofacy's 'Komplex' OS X Trojan*. http://researchcenter.paloaltonetworks.com/2016/09/unit42-sofacys-komplex-os-x-trojan/. Sept. 2016 (cit. on p. 134).

[CKC12] V. Chipounov, V. Kuznetsov, and G. Candea. "The S2E Platform: Design, Implementation, and Applications". en. In: *ACM Transactions on Computer Systems* 30.1 (Feb. 2012), pp. 1–49 (cit. on pp. 22, 31, 70).

[CN09] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1st. Addison-Wesley Professional, 2009 (cit. on pp. xix, 12).

[Col+12] C. Collberg, S. Martin, J. Myers, and J. Nagra. "Distributed Application Tamper Detection via Continuous Software Updates". en. In: ACM Press, 2012, p. 319 (cit. on pp. xxxi, 107, 114).

[Col03] C. Collberg. "Sandmark - A Tool for Software Protection Research". In: *IEEE Security & Privacy* 1540-7993.03 (2003), p. 10 (cit. on p. 102).

[Coo+09] K. Coogan, S. Debray, T. Kaochar, and G. Townsend. "Automatic Static Unpacking of Malware Binaries". In: *Proceedings of the 2009 16th Working Conference on Reverse Engineering*. WCRE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 167–176 (cit. on p. 18).

[CS13] C. Cadar and K. Sen. "Symbolic Execution for Software Testing: Three Decades Later". In: *Communications of the ACM* 56.2 (Feb. 2013), p. 82 (cit. on pp. xviii, 21).

[CTL97] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfuscating Transformations*. 1997 (cit. on pp. xxix, 101).

[CTL98] C. Collberg, C. Thomborson, and D. Low. "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. New York, NY, USA: ACM, 1998, pp. 184–196 (cit. on p. 101).

[Cuo+12] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. "Frama-C: A Software Analysis Perspective". In: *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*. SEFM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 233–247 (cit. on pp. xxviii, 4, 8, 125).

[Dal+06] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. "Opaque Predicates Detection by Abstract Interpretation". In: *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology*. AMAST'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 81–95 (cit. on pp. xx, 18).

[DAR16] DARPA. *DARPA | Cyber Grand Challenge*. https://www.cybergrandchallenge.com/. 2016 (cit. on p. 149).

[Dav+16a] R. David, S. Bardin, J. Feist, L. Mounier, M.-L. Potet, T. D. Ta, and J.-Y. Marion. "Specification of Concretization and Symbolization Policies in Symbolic Execution". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 2016, pp. 36–46 (cit. on pp. xxii, 8, 71).

[Dav+16b] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion. "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis". In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*. 2016, pp. 653–656 (cit. on pp. xxv, 8, 22, 32, 44, 169).

[Dav16] R. David. *IDASec - Github*. https://github.com/RobinDavid/IDASec. Jan. 2016 (cit. on p. 6).

[DB15] A. Djoudi and S. Bardin. "BINSEC: Binary Code Analysis with Low-Level Regions". In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 2015, pp. 212–217 (cit. on pp. xxi, xxv, 22, 23, 67).

[DB16a] R. David and S. Bardin. *Code Deobfuscation: Intertwining Dynamic, Static and Symbolic Approaches*. https://www.blackhat.com/eu-16/briefings/schedule/index.html#code-deobfuscation-intertwining-dynamic-static-and-symbolic-approaches-4933. 1/11/16 (cit. on pp. xxxv, 8, 148).

[DB16b] A. Djoudi and S. Bardin. "Recovering High-Level Conditions from Binary Programs". In: *(Under Review)*. 2016 (cit. on p. 84).

[DBG16] A. Djoudi, S. Bardin, and E. Goubault. "Recovering High-Level Conditions from Binary Programs". In: *21st International Symposium on Formal Methods*. 2016 (cit. on pp. 24, 84).

[DBM16] R. David, S. Bardin, and J.-Y. Marion. "Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes". In: (under review). 2016 (cit. on pp. xxiii, 8).

[Den16] Denuvo. *Denuvo Software Solutions*. http://denuvo.com/. 2016 (cit. on p. 11).

[Det+14] M. Deters, A. Reynolds, T. King, C. W. Barrett, and C. Tinelli. "A Tour of CVC4: How It Works, and How to Use It". In: *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. 2014, p. 7 (cit. on pp. xxiv, 73).

[Dol+15] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. "Repeatable Reverse Engineering with PANDA". In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. PPREW-5. New York, NY, USA: ACM, 2015, 4:1–4:11 (cit. on p. 68).

[DP09] T. Dullien and S. Porst. "REIL: A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis". In: *CanSecWest '09*. 2009 (cit. on p. 22).

[Dut14] B. Dutertre. "Yices 2.2". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 737–744 (cit. on p. 73).

[ESE15] ESET Research. *Sednit APT Group Meets Hacking Team*. http://www.welivesecurity.com/2015/07/10/sednit-apt-group-meets-hacking-team/. Oct. 2015 (cit. on p. 134).

[Fei+16] J. Feist, L. Mounier, M.-L. Potet, S. Bardin, and R. David. "Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free". In: (under review). 2016 (cit. on p. 8).

[Fei15] J. Feist. *GUEB Static Analyzer for Detecting Use-After-Free on Binary*. https://github.com/montyly/gueb. 2015 (cit. on p. 122).

[Fut+06] A. Futoransky, E. Kargieman, C. Sarraute, and A. Waissbein. "Foundations and Applications for Secure Triggers". In: *ACM Trans. Inf. Syst. Secur.* 9.1 (2006), pp. 94–112 (cit. on pp. xx, 15).

[Gar+13] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. *Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits*. Published: Cryptology ePrint Archive, Report 2013/451 http://eprint.iacr.org/. 2013 (cit. on p. 12).

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. "DART: Directed Automated Random Testing". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 213–223 (cit. on pp. 32, 42).

[GLM12] P. Godefroid, M. Y. Levin, and D. A. Molnar. "SAGE: Whitebox Fuzzing for Security Testing". In: *Commun. ACM* 55.3 (2012), pp. 40–44 (cit. on p. 21).

[GMR09] W. Guizani, J.-Y. Marion, and D. Reynaud-Plantey. "Server-Side Dynamic Code Analysis". In: *4th International Conference on Malicious and Unwanted Software, MALWARE 2009, Montréal, Quebec, Canada, October 13-14, 2009*. 2009, pp. 55–62 (cit. on p. 18).

[Goo16a] D. Goodin. *Puzzle Box: The Quest to Crack the World's Most Mysterious Malware Warhead*. http://arstechnica.com/security/2013/03/the-worlds-most-mysterious-potentially-destructive-malware-is-not-stuxnet/1/. 3/14/16 (cit. on p. 15).

[Goo16b] Google. *Google Optimization Tools*. https://developers.google.com/optimization/. 2016 (cit. on p. 73).

[Gui16] D. Guido. *Your Tool Works Better than Mine ? Prove It*. https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/. Blog. Jan. 2016 (cit. on p. 149).

[GWZ94] A. Goldberg, T. C. Wang, and D. Zimmerman. "Applications of Feasible Path Analysis to Program Testing". In: *ISSTA*. ACM, 1994 (cit. on p. 124).

[Hex16] Hex Rays. *IDA Pro - Hex-Rays*. https://www.hex-rays.com/index.shtml. Sept. 2016 (cit. on p. xxv).

[HG12] S. Heelan and A. Gianni. "Augmenting Vulnerability Analysis of Binary Code". In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 199–208 (cit. on pp. 21, 33, 42, 44).

[IOA12] IOActive. *Reversal and Analysis of Zeus and SpyEye Banking Trojans*. Tech. rep. 2012 (cit. on pp. xvii, 3).

[Jun+15] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. "Obfuscator-LLVM: Software Protection for the Masses". In: *Proceedings of the 1st International Workshop on Software Protection*. SPRO '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 3–9 (cit. on pp. xxx, 102, 107).

[Kin12] J. Kinder. "Towards Static Analysis of Virtualization-Obfuscated Binaries". In: *Proceedings of the 2012 19th Working Conference on Reverse Engineering*. WCRE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 61–70 (cit. on pp. xx, 18).

[Kin76] J. C. King. "Symbolic Execution and Program Testing". In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394 (cit. on pp. xx, 21).

[KPY07] M. G. Kang, P. Poosankam, and H. Yin. "Renovo: A Hidden Code Extractor for Packed Executables". en. In: ACM Press, 2007, p. 46 (cit. on pp. xx, 18).

[KV10] J. Kinder and H. Veith. "Precise Static Analysis of Untrusted Driver Binaries". In: *FMCAD 2010*. Springer, 2010 (cit. on p. 57).

[Lan13] R. Langner. *To Kill a Centrifuge*. Tech. rep. Nov. 2013 (cit. on pp. xvii, 3).

[lca16] lcamtuf. *American Fuzzy Loop*. http://lcamtuf.coredump.cx/afl/. Sept. 2016 (cit. on p. 123).

[Lei05] K. R. M. Leino. "Efficient Weakest Preconditions". In: *Inf. Process. Lett.* 93.6 (2005) (cit. on p. 58).

[LPG13] A. Lakhotia, M. D. Preda, and R. Giacobazzi. "Fast Location of Similar Code Fragments Using Semantic 'juice'". en. In: ACM Press, 2013, pp. 1–6 (cit. on pp. xx, 19).

[Luk+05] C.-K. Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200 (cit. on pp. xxiv, 6, 68, 70).

[MB08] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 2008, pp. 337–340 (cit. on pp. xxiv, 73).

[McC62] J. McCarthy. “Towards a Mathematical Science of Computation”. In: *IFIP Congress*. 1962, pp. 21–28 (cit. on p. 84).

[MG14] C. Mougey and F. Gabriel. *DRM Obfuscation vs Auxiliary Attacks*. Recon 2014, June 2014 (cit. on p. 11).

[Min+15] J. Ming, D. Xu, L. Wang, and D. Wu. “LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code”. en. In: ACM Press, 2015, pp. 757–768 (cit. on pp. 19, 105, 108).

[Mit15] Mitre. *Mitre CVE-2015-5221*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5221. Aug. 2015 (cit. on pp. xxviii, 7, 122, 123).

[MM16] X. Meng and B. P. Miller. “Binary Code Is Not Easy”. en. In: ACM Press, 2016, pp. 24–35 (cit. on pp. xvii, 4, 13, 114).

[NIS16] NIST. *SAMATE - Software Assurance Metrics And Tools Evaluation*. http://samate.nist.gov/. 2016 (cit. on pp. 45, 46).

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999 (cit. on p. 18).

[NPB15] A. Niemetz, M. Preiner, and A. Biere. “Boolector 2.0 System Description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), pp. 53–58 (cit. on pp. xxiv, 73).

[Paq15] E. Paquette. *Piratage de TV5 Monde: L’enquête S’oriente Vers La Piste Russe*. http://www.lexpress.fr/actualite/medias/piratage-de-tv5-monde-la-piste-russe_1687673.html. Sept. 2015 (cit. on p. 134).

[PGD15] M. D. Preda, R. Giacobazzi, and S. K. Debray. “Unveiling Metamorphism by Abstract Interpretation of Code Properties”. In: *Theor. Comput. Sci.* 577 (2015), pp. 74–97 (cit. on p. 18).

[Rol09] R. Rolles. “Unpacking Virtualization Obfuscators”. In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1 (cit. on p. 12).

[Roy+06] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware”. In: *Proceedings of the 22Nd Annual Computer Security Applications Conference*. ACSAC ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 289–300 (cit. on p. 18).

[SAB10] E. J. Schwartz, T. Avgerinos, and D. Brumley. “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: IEEE, 2010, pp. 317–331 (cit. on pp. 33, 41).

[Sha+08a] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. 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*. 2008 (cit. on p. 15).

[Sha+08b] M. I. Sharif, V. Yegneswaran, H. Säıdi, P. A. Porras, and W. Lee. "Eureka: A Framework for Enabling Static Malware Analysis". In: *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. 2008, pp. 481–500 (cit. on p. 18).

[Sho+16] Y. Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016 (cit. on pp. 22, 70).

[SMA05] K. Sen, D. Marinov, and G. Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272 (cit. on pp. 5, 21, 32, 42).

[SMS11] A. Sepp, B. Mihaila, and A. Simon. "Precise Static Analysis of Binaries by Extracting Relational Information". In: IEEE, Oct. 2011, pp. 357–366 (cit. on p. 57).

[Son+08] D. Song et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Proceedings of the 4th International Conference on Information Systems Security*. ICISS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 1–25 (cit. on p. 21).

[SS15] F. Saudel and J. Salwan. "Triton: A Dynamic Symbolic Execution Framework". In: *Symposium Sur La Sécurité Des Technologies de L'information et Des Communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 2015, pp. 31–54 (cit. on pp. 22, 70).

[SW13] A. Sahai and B. Waters. *How to Use Indistinguishability Obfuscation: Deniable Encryption, and More*. Published: Cryptology ePrint Archive, Report 2013/454 http://eprint.iacr.org/. 2013 (cit. on p. 12).

[Tor15] J. Torrey. *HARES, Hardened Anti-Reverse Engineering System*. Tech. rep. Infiltrate 2015, 2015 (cit. on p. 15).

[Tra14] Trails of Bits. *ReMASTering Applications by Obfuscating during Compilation*. https://blog.trailofbits.com/2014/08/20/remastering-applications-by-obfuscating-during-compilation/. 20/08/14 (cit. on p. 12).

[Tre14] Trend Micro. *Operation Pawn Storm, Using Decoys to Evade Detection*. Tech. rep. 2014 (cit. on pp. xxxii, 134).

[Tre15a] Trend Micro. *Operation Pawn Storm Ramps Up Its Activities; Targets NATO, White House*. http://blog.trendmicro.com/trendlabs-security-intelligence/operation-pawn-storm-ramps-up-its-activities-targets-nato-white-house/. Apr. 2015 (cit. on p. 134).

[Tre15b] Trend Micro. *Pawn Storm Targets MH17 Investigation Team*. http://blog.trendmicro.com/trendlabs-security-intelligence/pawn-storm-targets-mh17-investigation-team/. Oct. 2015 (cit. on p. 134).

[Uga+15] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers". In: *SSP 2015, IEEE Symposium on Security and Privacy, May 18-20, 2015, San Jose, CA, USA*. San Jose, UNITED STATES, May 2015 (cit. on pp. xx, 18, 63, 169, 170).

[USC14] US-CERT. *Alert (TA14-353A) Targeted Destructive Malware*. https://www.us-cert.gov/ncas/alerts/TA14-353A. 2014 (cit. on pp. xvii, 3).

[von15] von Gastbeitrag. *Digital Attack on German Parliament: Investigative Report on the Hack of the Left Party Infrastructure in Bundestag*. https://netzpolitik.org/2015/digital-attack-on-german-parliament-investigative-report-on-the-hack-of-the-left-party-infrastructure-in-bundestag/. June 2015 (cit. on pp. xxxii, 134).

[VXH16] VXHeaven. *VX Heaven*. 2016 (cit. on p. 46).

[Wan+11] Z. Wang, J. Ming, C. Jia, and D. Gao. "Linear Obfuscation to Combat Symbolic Execution". In: *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*. 2011, pp. 210–226 (cit. on p. 15).

[Wey93] E. Weyuker. "More Experience with Data Flow Testing". In: *IEEE Trans. Softw. Eng.* 19.9 (1993) (cit. on p. 124).

[WHH80] M. Woodward, D. Hedley, and M. Hennell. "Experience with Path Analysis and Testing of Programs". In: *IEEE Trans. Softw. Eng.* SE-6.3 (1980) (cit. on p. 124).

[Wil+05] N. Williams, B. Marre, P. Mouy, and M. Roger. "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In: *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*. 2005, pp. 281–292 (cit. on pp. xxix, 5, 21, 33, 125).

[Yad+15] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. "A Generic Approach to Automatic Deobfuscation of Executable Code". In: IEEE, May 2015, pp. 674–691 (cit. on p. 19).

[YD15] B. Yadegari and S. Debray. "Symbolic Execution of Obfuscated Code". en. In: ACM Press, 2015, pp. 732–744 (cit. on pp. xx, 19, 22, 63, 107).

[YM89] D. Yates and N. Malevris. "Reducing the Effects of Infeasible Paths in Branch Testing". In: *ACM SIGSOFT Softw. Eng. Notes* 14.8 (1989) (cit. on p. 124).

[Zho+07] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms". In: *Information Security Applications*. Ed. by D. Hutchison et al. Vol. 4867. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75 (cit. on pp. xix, 13).

Appendix A

Anti-Debug examples

```
int check_vbox_regkey() {
  HKEY regkey;
  string key_s = "SOFTWARE\\Oracle\\VirtualBox Guest Additions";
  LONG ret = RegOpenKeyEx(HKEY_LOCAL_MACHINE, key_s, 0, KEY_READ, &
      regkey);
  if (ret == ERROR_SUCCESS) {
    RegCloseKey(regkey);
    return true;
  }
  else
    return false;
}
```

Listing A.1: Check VirtualBox registry entry

```
int check_vbox_process() {
  int found = false;
  PROCESSENTRY32 pentry;

  HANDLE snapshot = CreateToolhelp32Snapshot( TH32CS_SNAPPROCESS, 0 );

  if(!Process32First(snapshot, &pentry)) {
    CloseHandle(snapshot);
    return found;
  }

  do {
    if (lstrcmpi(pentry.szExeFile, "vboxservice.exe") == 0) {
      found = true;
    }
  } while (Process32Next(snapshot, &pentry));
  return found;
}
```

Listing A.2: Check the presence of a VirtualBox process

```
unsigned long long rdtsc() {
  unsigned eax, edx;
  __asm__ volatile("rdtsc" : "=a" (eax), "=d" (edx));
  return ((unsigned long long)eax) | (((unsigned long long)edx) << 32);
}

int check_rdtsc() {
  int i;
  unsigned long long avg, ret = 0;
  for (i = 0; i < 10; i++) {
    ret = rdtsc();
    avg = avg + (rdtsc() - ret);
    Sleep(500);
  }
  avg = avg / 10;
  return (avg < 750 && avg > 0) ? FALSE : TRUE;
}
```

Listing A.3: Check the RDTSC average difference value

Appendix B

Wave-model difference

A throughout study of the different self-modification models has been made in 2015 [Uga+15]. This annexe aim at emphasizing the difference between Codisasm [Bon+15] and PINSEC [Dav+16b] differences. Let's consider N the current layer of self-modification. The self-modification detected is performed by tagging memory cells with a given layer number. If that memory cells is later executed the current layer level N will switch to this number if strictly superior than the previous N. The difference between the two models lays in the way to propagate the self-modification aka the marking of memory cells. On an instruction writing in memory Codisasm will use the global current layer counter N while PINSEC will use the value of the instruction itself denoted i_n where i is the instruction and n the layer associated to it. In this model all instruction are marked with layer 0 by default. Thus, an instruction is only marked with $n+1$ if it is written by an instruction of level n. The figure B.1 shows graphicaly the difference between the two models. Both modeling will update theirs global counter to $N = 1$ when entering F#1. Meanwhile the execution flow might return the original code and while PINSEC will consider F#2 as another chunk of the layer 1 Codisasm will consider it to be another layer.

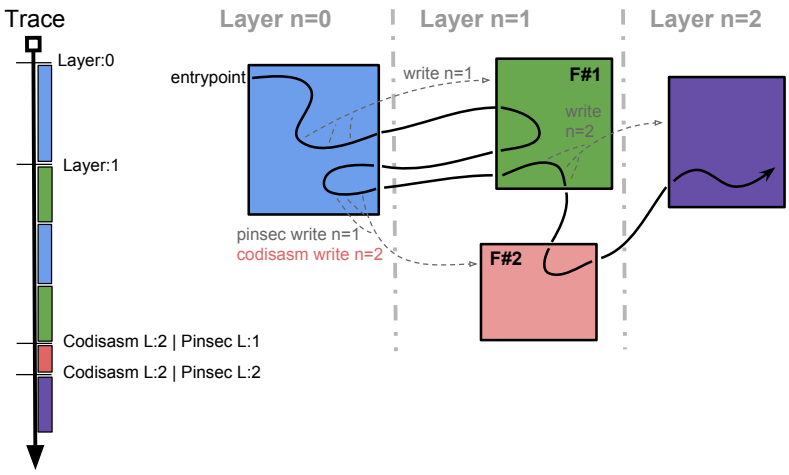


Figure B.1: Layer transition model

Reflecting to the formalization and modeling of [Uga+15] Codisasm address perfectly single frame packer, linear(forward) with tail transition in full-code unpacking mode, but consider any frame sub-division as layers.

Appendix C

X-Tunnel dependancy opaque predicate

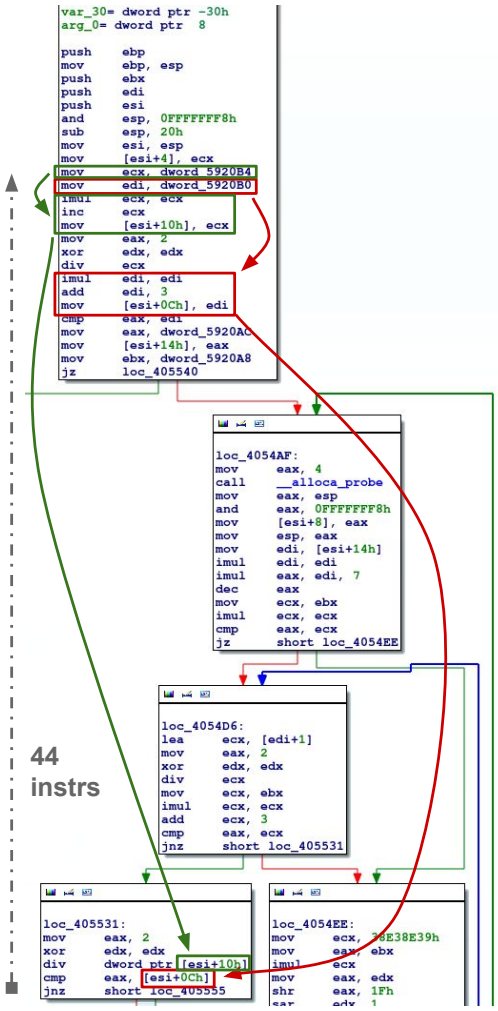


Figure C.1: 44 instructions depth, dependency

Résumé

L'analyse de codes malveillants est un domaine de recherche en pleine expansion de par la criticité des infrastructures touchées et les coûts impliqués de plus en plus élevés. Ces logiciels utilisent fréquemment différentes techniques d'évasion visant à limiter la détection et ralentir les analyses. Parmi celles-ci, l'obfuscation permet de cacher le comportement réel d'un programme. Cette thèse étudie l'utilité de l'*Exécution Symbolique Dynamique* (DSE) pour la rétro-ingénierie. Tout d'abord, nous proposons deux variantes du DSE plus adaptées aux codes protégés. La première est une redéfinition générique de la phase de calcul de prédicat de chemin basée sur une manipulation flexible des concrétisations et symbolisations tandis que la deuxième se base sur un algorithme d'exécution symbolique arrière borné. Ensuite, nous proposons différentes combinaisons avec d'autres techniques d'analyse statique afin de tirer le meilleur profit de ces algorithmes. Enfin tout ces algorithmes ont été implémentés dans différents outils, Binsec/se, Pinsec et Idasec, puis testés sur différents codes malveillants et packers. Ils ont permis de détecter et contourner avec succès les obfuscations ciblées dans des cas d'utilisations réels tel que X-Tunnel du groupe APT28/Sednit.

Mots-clés: Code malveillants, Désobfuscation, Méthodes formelles, Exécution Symbolique, Rétro-ingénierie.

Abstract

Malware analysis is a growing research field due to the criticity and variety of assets targeted as well as the increasing implied costs. These softwares frequently use evasion tricks aiming at hindering detection and analysis techniques. Among these, obfuscation intent to hide the program behavior. This thesis studies the potential of Dynamic Symbolic Execution (DSE) for reverse-engineering. First, we propose two variants of DSE algorithms adapted and designed to fit on protected codes. The first is a flexible definition of the DSE path predicate computation based on concretization and symbolization. The second is based on the definition of a backward-bounded symbolic execution algorithm. Then, we show how to combine these techniques with static analysis in order to get the best of them. Finally, these algorithms have been implemented in different tools Binsec/se, Pinsec and Idasec interacting alltogether and tested on several malicious codes and commercial packers. Especially, they have been successfully used to circumvent and remove the obfuscation targeted in real-world malwares like X-Tunnel from the famous APT28/Sednit group.

Keywords: Malware, Deobfuscation, Formal Methods, Symbolic Execution, Reverse-Engineering.