Titre: Title:	Déduction des cibles de sauts indirects pour les applications de traçage x86
Auteur: Author:	Gabriel-Andrew Pollo-Guilbert
Date:	2021
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pollo-Guilbert, G.-A. (2021). Déduction des cibles de sauts indirects pour les applications de traçage x86 [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/9990/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/9990/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Déduction des Cibles de Sauts Indirects pour les Applications de Traçage x86

GABRIEL-ANDREW POLLO-GUILBERT
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Décembre 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Déduction des Cibles de Sauts Indirects pour les Applications de Traçage x86

présenté par **Gabriel-Andrew POLLO-GUILBERT**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Guy BOIS, président
Michel DAGENAIS, membre et directeur de recherche
François-Raymond BOYER, membre

DÉDICACE

Je dédicace ce mémoire à mon père et ma mère pour le support inconditionnel qu'ils m'ont offert à travers ma vie, mes études et mes choix.

Je dédicace ce mémoire à mes frères et soeurs, mes amis et amies, et ma copine pour m'avoir toujours encouragé et cru en moi.

REMERCIEMENTS

Je remercie premièrement Michel Dagenais, mon directeur de recherche, pour l'opportunité de ma maitrise, pour ses conseils et pour son encadrement, malgré le travail à distance en période de pandémie. Je remercie aussi à ses côtés Geneviève Bastien de m'avoir offert la chance d'explorer et d'en apprendre sur le domaine dès mon entrée à l'université.

Je remercie aussi les collègues du laboratoire ou de l'industrie avec qui j'ai eu la chance de travailler et discuter, soit Clément, Misha, Mohammad, Vahid et Jason.

Je remercie aussi tous les partenaires industriels ou gouvernementaux ayant participé financièrement ou techniquement aux différents projets, soit Ericsson, Ciena, AMD, EfficiOS, Prompt et le Conseil de Recherches en Sciences Naturelles et en Génie du Canada.

Finalement, je remercie les lecteurs et surtout le Jury pour leur intérêt porté à ce mémoire.

RÉSUMÉ

Le traçage est un outil puissant utilisé dans le processus de développement d'une application. Ceci est particulièrement utile pour comprendre le comportement des applications qui interagissent en temps réel avec plusieurs composantes externes. En effet, ces applications ne peuvent généralement pas être arrêtées pour inspecter leur état interne à l'aide d'un débogueur interactif. A la place, alors que l'application roule à vitesse normale, une trace de son execution est générée, afin de suivre son chemin d'exécution et enregistrer les valeurs importantes dans la trace.

Le programmeur doit premièrement insérer des sondes dans son logiciel. Lorsqu'une sonde est lancée, un évènement est créé et enregistré. Grâce au traçage, le développeur est en mesure d'extraire un grand volume d'information critique à l'optimisation ou au débogage de certains problèmes. Contrairement à la journalisation, le traçage se doit d'avoir le plus petit surcout d'exécution. Par conséquent, différentes optimisations sont effectuées.

Dans les applications de haute disponibilité, le développeur doit pouvoir extraire de l'information concernant l'exécution. Idéalement, une sonde est déjà insérée statiquement dans le code source de l'application. Cependant, il peut être difficile de prédire quel problème pourrait survenir en production, l'insertion des sondes doit être plus flexible. Grâce à l'instrumentation dynamique, il sera possible d'insérer de telles sondes en parallèle à l'exécution d'un programme.

L'insertion dynamique d'une sonde consiste à modifier certaines parties d'un programme, au niveau des instructions, afin de rediriger le flot d'exécution vers une routine d'instrumentation qui se charge d'enregistrer les données d'intérêts. Cette modification du programme amène beaucoup de problèmes et de défis. Un des problèmes qui limite le taux de succès de l'instrumentation est la difficulté d'extraire le graphe de flot de contrôle d'une fonction, pour déterminer si deux instructions ou plus peuvent être remplacées par une redirection.

Dans ce mémoire, nous présentons une technique basée sur la correspondance de patrons afin d'extraire ce graphe. Ce problème a longtemps été étudié dans le domaine de la translation de code ou de la rétro-ingénierie, mais rarement dans le domaine du traçage. Notre technique est en mesure d'extraire avec une grande fiabilité les données nécessaires au graphe de flot de contrôle et d'ainsi augmenter le taux de succès de l'instrumentation dynamique.

ABSTRACT

Tracing is a powerful tool used in the process of application development. It is especially useful to understand the behavior of applications that interact in real time with many different external components. Indeed, such applications often cannot be stopped, to inspect their state, using an interactive debugger. Instead, while the application is running at normal speed, a trace of its execution is generated, in order to follow its execution path and record important values in the trace.

The developer must first insert tracepoints into the application. When a tracepoint is reached and executed, an event is created and saved in the trace. Thanks to tracing, the developer is able to extract a large amount of critical data that can be used for optimisation or debugging. Unlike logging, one of the requirements of tracing is that its execution must carry as low overhead as possible. To do so, multiple techniques are used by different tracers.

In high reliability environments, a common task of a developer is to be able to extract information concerning the execution of a system. Ideally, a tracepoint would already be inserted statically into the application source code. However, it is very difficult to predict what kind of bug the system will face in production. Hence, the instrumentation of a program needs to be more flexible, since we may be missing tracepoints. Thanks to dynamic instrumentation, it will now be possible to insert such dynamic tracepoints while the system is running.

The insertion of dynamic tracepoints usually relies on the modification of individual instructions in a program. It needs to do that in order to redirect the execution flow to specialised instrumentation that will save the data of interest. Modifying a program, especially when it is running, brings a great deal of problems and challenges. One of the problems that decreases the success rate of dynamic instrumentation is the difficult task of recovering the control flow graph of a function, to determine if two or more instructions can be replaced by a redirection.

In this thesis, we present a technique, based on pattern matching, that can recover the required jump targets, to obtain a control flow graph. This problem has long been studied in the domain of binary translation or reverse engineering, but rarely in the domain of tracing. Our efficient technique is able to recover with high reliability the control flow graph, hence increasing the success rate of dynamic instrumentation.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BPF	Berkeley Packet Filter
CET	Control-flow Enforcement Technology
CFG	Control Flow Graph
CTF	Common Trace Format
DWARF	Debugging With Attributed Record Formats
eBPF	extended Berkeley Packet Filter
GCC	GNU C Compiler
GFC	Graphe de Flot de Contrôle
GNU	GNU's Not Unix
LTTng	Linux Tracing Toolkit – next generation
NFA	Nondeterministic Finite Automaton
RCU	Read-Copy-Update
RISC	Reduced Instruction Set Computer
UCT	Unité Centrale de Traitement

CHAPITRE 1 INTRODUCTION

Une partie importante du processus de développement logiciel est le débogage et la mise au point, qui généralement impliquent la détection et la correction de problèmes de logique ou de performance. Ceux-ci ont pour but d'augmenter la fiabilité ou l'efficacité du logiciel, en réduisant le plus possible les problèmes d'exécution. Pratiquement aucun logiciel n'échappe à ce processus. C'est pourquoi une grande variété d'outils, comme des débogueurs, furent développés à travers les années. Dans tous les cas, il est important de recueillir des données sur un problème non trivial avant de pouvoir le réparer.

Afin d'extraire des données sur un problème logiciel, celui-ci doit être instrumenté. L'instrumentation est souvent insérée avant l'exécution du logiciel, par exemple dans le code source lui-même. Si la surcharge due à l'instrumentation est faible, il est même possible de la laisser dans un environnent de production. Cela peut être utile lorsqu'un problème survient en production, et que le problème ne peut facilement être reproduit en redémarrant le service concerné en laboratoire, avec l'instrumentation ajoutée. Cependant, il est très difficile de prévoir l'ensemble des problèmes qui peuvent être découverts en production. Il est donc peu probable que toute l'instrumentation requise ait été ajoutée statiquement, pour aider à diagnostiquer les problèmes encore inconnus appelés à se manifester en production.

Pour couvrir ces cas, il existe des techniques pouvant instrumenter un logiciel après que celui-ci soit démarré. Pour ce faire, il est nécessaire de modifier les instructions-machine elles-mêmes, souvent en parallèle avec l'exécution du programme. Ces techniques complexes doivent prendre grand soin de ne pas perturber ou interrompre l'exécution du logiciel instrumenté. Par exemple, une mauvaise modification peut mener le logiciel à exécuter une instruction invalide.

1.1 Définitions et concepts de base

1.1.1 Traçage

Le traçage est une forme de journalisation spécialisée servant à extraire de l'information à propos de l'exécution d'un logiciel. La journalisation est souvent dans un format textuel et contient de l'information générale sur le logiciel. Le volume de données, dans les journaux de format texte, est habituellement bas. Par contraste, le traçage vise habituellement de l'information plus détaillée. Une trace peut contenir beaucoup d'information, incluant des données hors de l'application, comme l'état du système d'exploitation. Son volume de données

est généralement plus élevé. Par conséquent, le traçage est souvent beaucoup plus optimisé et utilise principalement des formats binaires. Des outils additionnels aident au traitement des fichiers de trace.

1.1.2 Sonde

Une sonde est un point d'intérêt dans l'application qui a été instrumentée. Durant l'exécution du programme, la sonde est en charge de lire les informations demandées et de les enregistrer, si nécessaire. En raison des différents requis de l'industrie, différent types de sondes existent.

Sonde statique

La sonde la plus simple est la sonde statique. Pour de la journalisation de base, un simple `printf` ou `std::cout` entre dans cette catégorie. Ces sondes doivent être manuellement insérées par l'utilisateur, souvent dans le code source, et ne peuvent être totalement désactivées durant l'exécution. Lorsqu'elles sont pleinement optimisées, ce qui n'est pas le cas de `printf` ou `std::cout`, elles ont le potentiel d'être les sondes les plus efficaces.

Sonde statique avec activation dynamique

Ce type de sonde est très similaire aux sondes statiques, à l'exception qu'elles peuvent être totalement désactivées durant l'exécution. Lorsqu'elles ne sont pas actives, il n'y a pratiquement aucun surcout à l'exécution de l'application. Elles sont souvent implémentées en insérant des instructions vides, comme un NOP, dans le code machine de l'application.

Les compilateurs modernes offrent des arguments de compilation permettant d'automatiquement insérer ces sondes aux entrées et fins des fonctions. Sous x86, l'option `-mnop-mcount` de GCC ajoute 5 octets (NOP) au début de la fonction, afin d'être remplacés par un saut vers de l'instrumentation. De son côté, Clang offre `-fxray-instrument` qui alloue 11 octets à cet effet au début et à la fin des fonctions.

Sonde entièrement dynamique

Les sondes les plus flexibles sont celles qui sont entièrement dynamiques. Lorsqu'on les utilise, aucune modification n'a à être effectuée par le développeur sur le code source ou le programme produit. Il faut seulement que les symboles de débogage soient présents. Comme pour les sondes avec activation dynamique, une instruction est remplacée par un saut vers de l'instrumentation. Cependant, l'instruction remplacée n'est pas nécessairement un NOP,

mais souvent une vraie instruction qui fait partie du programme.

Remplacer des instructions qui font partie du programme génère plusieurs problèmes d'implémentation. Par exemple, l'instruction remplacée doit néanmoins se faire exécuter, ou son effet doit être émulé ailleurs dans le code, afin de conserver le comportement original du programme. Parfois, l'instruction originale est trop petite pour contenir la nouvelle instruction, auquel cas plusieurs instructions doivent être remplacées, ce qui apporte encore plus de problèmes.

En résumé, ces sondes sont très flexibles, car elles ne demandent aucune modification du programme original. Toutefois, leur implémentation est considérablement plus complexe, voire même risquée, si tous les cas problématiques ne sont pas pris en compte.

Évènement

Lorsqu'une sonde est exécutée, celle-ci génère un évènement et enregistre des valeurs d'intérêt. Un fichier de trace se résume généralement à un ou plusieurs flux d'évènements. Chaque évènement est typé, c'est-à-dire qu'il respecte la description d'une structure de données, souvent prédéfinie dans un fichier de métadonnées séparé. Afin de diminuer l'espace utilisé et d'accélérer l'exécution de la sonde, les évènements sont enregistrés directement dans un fichier binaire.

1.1.3 Architecture x86-64

L'architecture x86-64 est encore celle qui domine dans les ordinateurs de table et les serveurs. À l'origine, l'architecture x86 était une architecture 16 bits, elle a été étendue à 32 bits en 1985 par Intel. Par la suite, elle fut à nouveau étendue à 64-bits par AMD en 2000.

Taille des instructions

Une des caractéristiques importantes de cette architecture est que la taille de ses instructions est variable. Ceci est dû au fait qu'elle implémente un jeu d'instructions complexe. Au lieu d'utiliser plusieurs instructions pour faire une tâche complexe, il existe souvent une instruction effectuant directement cette tâche. Par conséquent, ceci fait en sorte que l'architecture contient beaucoup d'instructions, trop pour utiliser des instructions à taille fixe. Aujourd'hui, l'architecture compte près de 1000 instructions réparties sur plus de 3600 variantes [41].

Trappe de débogage

Lorsqu'un débogueur veut ajouter un point d'arrêt dans le programme, celui-ci doit remplacer une instruction par une autre pouvant arrêter le programme. Pour ce faire, l'architecture x86 offre une trappe de débogage avec l'instruction `int3`. La longueur de cette instruction est de seulement 1 octet. Par conséquent, il est possible remplacer le début de n'importe quelle instruction par cette trappe.

Lorsque la trappe est exécutée, une interruption est premièrement lancée au système d'exploitation. Celui-ci est en mesure de diriger le signal vers le débogueur, s'exécutant dans l'espace utilisateur. Lorsque le débogueur reçoit le signal, il peut suspendre l'exécution de l'application avec des appels système comme `ptrace` sous Linux.

Au minimum, l'exécution d'une trappe cause au moins 4 changements de contexte : un premier pour gérer l'interruption dans le noyau, un deuxième et un troisième pour entrer et quitter la routine du débogueur qui traite le signal, et un dernier pour finir la gestion de l'interruption dans le noyau et retourner le contrôle à l'application. Cette grande quantité de changements de contexte ajoute une surcharge d'exécution considérable lorsque la trappe est utilisée dans un environnent de traçage.

Saut indirect

Il existe plusieurs modes d'adressage pour les sauts sous cette architecture. Celle la plus utilisée est, sans doute, l'adressage immédiat. Dans ce mode, la cible est soit relative ou absolue, et est encodée directement dans l'instruction. En analysant ce saut, on peut déterminer sa cible sans problème.

Un saut peut aussi utiliser l'adressage indirect. Dans ce cas, la cible est contenue dans un registre ou à une adresse mémoire. Lorsque le processeur exécute ce saut, il doit premièrement lire la cible, par exemple de la mémoire, et ensuite il déplace l'exécution. Lorsqu'on analyse l'instruction, il est très difficile de déterminer la ou les cibles du saut, car ceci dépend de l'état du programme pendant son exécution.

1.2 Éléments de la problématique

Sous x86, il existe plusieurs méthodes pour implémenter des sondes entièrement dynamiques. La méthode la plus facile est d'utiliser la trappe de débogage. Cependant, celle-ci cause un surcout d'exécution significatif, surtout dans l'espace utilisateur. Une méthode souvent plus intéressante est de remplacer une instruction par un saut vers la sonde d'instrumentation.

Un saut x86 de cinq octets est souvent utilisé, car il permet d'avoir une cible se trouvant à ±2 Go du saut, ce qui permet dans la grande majorité des cas de rejoindre un endroit disponible pour y placer des sondes. Pour insérer ce saut, il faut remplacer une instruction d'une longueur de cinq octets ou plus. Dans le cas où l'instruction à remplacer est de moins de cinq octets, il est possible d'utiliser un saut relative à une adresse de un octet, mais celui-ci ne permet que de sauter de ±127 octets. En raison de cette limitation, il est souvent requis d'utiliser un saut de cinq octets, quitte à devoir remplacer plusieurs instructions.

Lorsqu'on modifie plusieurs instructions, la première instruction est remplacée par une autre instruction valide. Cependant, la deuxième et celles après sont remplacées par les octets internes du saut inséré, qui peuvent être vus comme des instructions invalides, soit inappropriées soit qui ne font pas partie du répertoire d'instructions x86-64. Par conséquent, il est impératif que le processeur ne les exécute jamais directement, afin de ne pas modifier le résultat du programme ou le terminer abruptement.

Il existe deux situations où ces instructions invalides seraient exécutées. Premièrement, il est possible qu'un fil d'exécution soit en train d'exécuter exactement ces instructions pendant qu'on les modifie. Ce cas peut être évité soit en attendant que le fil finisse l'exécution de ces instructions, ou en déplaçant ce fil vers des instructions équivalentes ailleurs dans la mémoire.

Deuxièmement, il est possible qu'un saut ailleurs dans la fonction cible les instructions qui ont été remplacées. Pour éviter cette situation, il faut analyser la fonction avant de l'instrumenter, afin de s'assurer qu'aucun saut ne cible ces instructions. Pour ce faire, il suffit de vérifier la ou les cibles de chaque saut. Pour un saut utilisant un adressage immédiat, ou direct, il suffit de décoder l'instruction pour avoir la cible. Pour un saut utilisant un adressage indirect, il est impossible d'obtenir la cible seulement en décodant l'instruction, car elle dépend de l'exécution du logiciel. Par conséquent, il faut utiliser des méthodes beaucoup plus complexes et difficiles à implémenter pour déterminer les cibles. À cause de cette complexité, le cas où un saut indirect est présent n'est pas supporté dans les systèmes existants comme Uftrace ou Kprobe, et l'instrumentation ne peut être effectuée.

Par exemple, lorsqu'on veut instrumenter toutes les fonctions d'un programme, il y a presque toujours un sous-ensemble de fonctions où l'instrumentation échoue. Dans certains cas, le remplacement d'instructions multiples par un saut de cinq octets est impossible, car un saut cible une instruction invalide. Dans le cas des sauts indirects, il se peut que ce soit possible, mais qu'on ne puisse pas déterminer la cible de ces sauts, et donc écarter la possibilité d'exécuter des instructions invalides. Dans un tel cas, l'instrumentation n'est pas effectuée. Des techniques plus sophistiquées pour déterminer la cible des sauts indirects peuvent donc permettre d'instrumenter un plus grand nombre de fonctions dans un programme.

1.3 Objectifs de recherche

L'objectif de ces travaux est de déterminer la cible des sauts indirects, dans les sondes entièrement dynamiques, afin d'augmenter le taux de succès de l'instrumentation.

Lorsqu'on découvre la cible des sauts indirects, il est possible d'estimer exactement, sous-estimer ou surestimer les cibles. Il est impératif que notre méthode ne sous-estime pas les cibles, car cela reviendrait à ne pas prouver qu'il est impossible de sauter vers des instructions invalides. La technique proposée doit être simple et efficace, car elle s'exécute dans un environnent de traçage, là où le surcout est un facteur important.

1.4 Contributions

Lors de ces travaux, un nouvel algorithme a été mis au point à des fins d'instrumentation dynamique. Celui-ci met l'accent sur la rapidité, plutôt que sur le recouvrement de toutes les cibles de saut indirect. Il a été développé dans une bibliothèque et intégré au sein d'une implémentation existante et mature d'instrumentation dynamique. Il a ensuite été validé sur une grande variété de logiciels représentatifs. En conséquence, la nouvelle bibliothèque permet d'obtenir un meilleur taux de succès, pour l'instrumentation dynamique, que les outils existants de l'état de l'art.

L'article en lien avec les travaux a été soumis dans le journal *Software : Practice and Experience* le 11 Décembre 2021. Nous sommes en attente de la réponse des examinateurs.

1.5 Plan du mémoire

Le chapitre 2 de ce mémoire suit et présente la revue de littérature dans le domaine du traçage et du recouvrement des sauts indirects. Par la suite, le chapitre 3 présente le corps du mémoire sous la forme d'un article scientifique. Finalement, le chapitre 4 présente la synthèse des travaux et les possibles améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE / LITERATURE REVIEW

Il existe une grande variété d'outils pour tracer, profiler ou déboguer différents systèmes. Chaque outil disponible utilise différentes manières d'instrumenter un logiciel, afin d'obtenir de l'information sur son exécution. Dans ce chapitre, on présente une série d'outils documentés et disponibles ouvertement dans le but de vérifier l'exécution d'un programme. Premièrement, on présente les outils permettant d'instrumenter statiquement et dynamiquement les programmes en mode noyau ou utilisateur. Par la suite, on présente des outils d'analyse et de traçage se basant sur ces derniers outils. Ensuite, différentes techniques d'instrumentation dynamique avancées sont présentées. Finalement, on discute des techniques utilisées pour extraire les graphes de flot de contrôle et les sauts indirects.

2.1 Outils d'Instrumentation

2.1.1 DWARF

Certains outils, dont un débogueur, doivent avoir accès à de l'information de débogage afin de pouvoir extraire de l'information sur l'exécution du programme. Sous Linux, le format standard est Debugging With Attributed Record Formats (DWARF). Celui-ci fournit beaucoup d'information. Par exemple, il fournit une table de correspondance entre les numéros de ligne dans le code source et les adresses des instructions du programme. Ou encore, il définit les différents types décrivant les structures utilisées dans le programme. De plus, il est possible d'obtenir la structure de la pile d'appels lors, par exemple, d'un point d'arrêt [21].

Finalement, DWARF décrit aussi comment accéder aux différentes variables accessibles à travers chaque adresse du programme. Ceci est important, car la variable peut se trouver à une adresse fixe dans l'espace mémoire, elle peut être sur la pile ou aussi dans un registre. Des fois, une variable peut se trouver dans la pile, mais ensuite temporairement dans un registre pour effectuer des calculs. Le débogueur doit être en mesure d'obtenir la valeur de la variable, peu importe où elle se trouve.

Pour ce faire, DWARF décrit la position d'une variable à une instruction particulière, à l'aide des expressions DWARFs. Celles-ci sont composées de plusieurs instructions s'exécutant sur une machine simple ayant quelques registres et une pile. Lorsque le débogueur veut obtenir la position d'une variable, il n'a qu'à simuler cette machine et y exécuter les instructions [22]. Quoiqu'encore peu utilisé, ceci est particulièrement important pour les outils d'instrumentation et de traçage, leur permettant d'interpréter le code source et de lire les variables au lieu

de seulement lire les registres.

2.1.2 Kprobe

Kprobe est un sous-système du noyau Linux [42] permettant aux concepteurs d'insérer des sondes entièrement dynamiques dans la majorité des fonctions du noyau. Il supporte x86, ARM, PowerPC et beaucoup d'autres architectures. De base, Kprobe remplace une instruction par une trappe de débogage (selon l'architecture). L'instruction qui est remplacée est exécutée hors ligne lors du lancement de la sonde. Les sondes peuvent être utilisées pour enregistrer des données comme des registres d'intérêt.

Lorsque la trappe est lancée, la routine d'interruption du noyau est appelée. Celle-ci détermine la sonde source qui a été lancée et appelle l'instrumentation de l'utilisateur. L'utilisateur peut spécifier deux fonctions à être appelées pour une même sonde. La première s'exécute avant d'exécuter l'instruction hors-ligne, tandis que la deuxième s'exécute après. Une fois l'instrumentation exécutée, le contrôle retourne à la fonction originale. La figure 2.1 illustre le flot d'exécution lors du lancement d'une sonde.

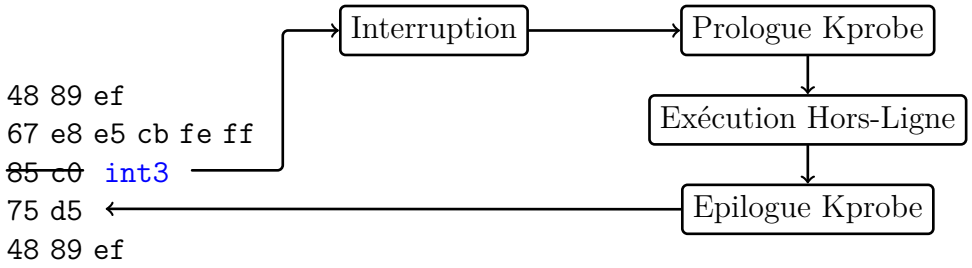


FIGURE 2.1 Exemple du lancement d'une sonde Kprobe.

Si demandé par la suite, Kprobe peut optimiser la sonde en la remplaçant, par exemple sous x86, par un saut. Cela élimine les changements de contextes ayant lieu lors de la gestion de l'interruption. Le flot d'exécution est similaire. Par contre, plusieurs instructions peuvent être exécutées hors-ligne. La figure 2.2 résume ce flot d'exécution.

Avant d'optimiser une trappe sous x86, Kprobe doit s'assurer qu'aucun autre fil ne soit exécuté actuellement, ou puisse s'exécuter ultérieurement, dans les instructions qui se trouvent dans les quatre octets suivant la trappe. En effet, la trappe plus ces quatre octets seront modifiés et remplacés par un saut de cinq octets.

Pour vérifier qu'aucun fil ne va exécuter les instructions modifiées dans le futur, Kprobe doit analyser les sauts de la fonction afin de déterminer leurs cibles. La cible des sauts directs peut facilement être déterminée en décodant l'instruction. Pour les sauts indirects, les cibles

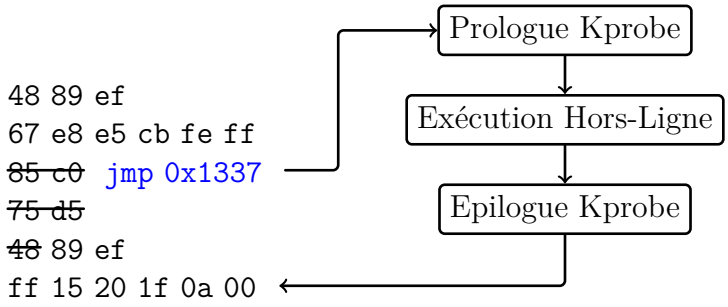
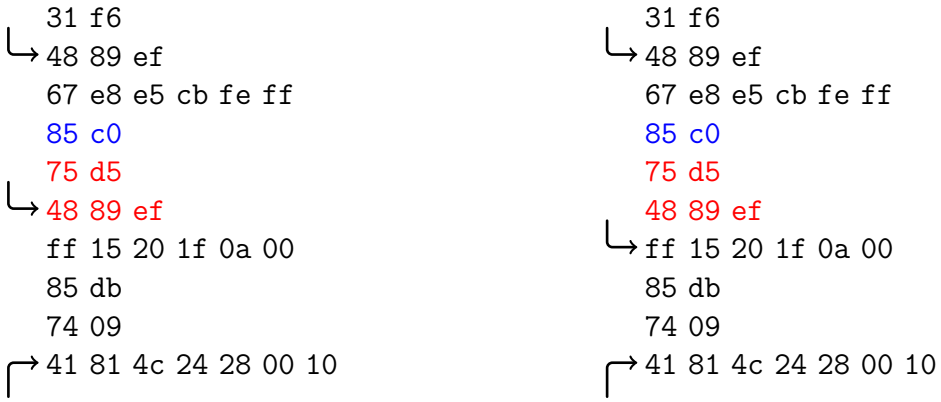


FIGURE 2.2 Exemple du lancement d'une sonde Kprobe optimisée.

ne peuvent pas être déterminées seulement en analysant l'instruction, car elles dépendent de l'état du programme durant l'exécution. Par conséquent, il faut effectuer une analyse considérablement plus complexe afin de déduire les cibles. Kprobe ne sait pas déduire ces cibles et n'optimise donc pas les fonctions contenant des sauts indirects. Même si dans la majorité des cas, il est probablement sécuritaire d'effectuer l'optimisation, le noyau ne prend pas ce risque, puisqu'il pourrait résulter en un échec du système en entier.

Une fois les sauts déterminés, il faut s'assurer qu'aucun ne cible les instructions après le premier octet à remplacer par le saut. Si c'est le cas, l'optimisation ne peut pas s'entreprendre. La figure 2.3 illustre deux exemples de sondes pouvant ou non être optimisées.



(a) cas non supporté

(b) cas supporté

FIGURE 2.3 Exemple de cas d'optimisation. Les flèches démontrent les cibles d'autres sauts dans la fonction. Les instructions en bleu sont les instructions qui seront instrumentées. Les instructions en rouge sont les instructions qui seront modifiées et qui ne devraient pas être exécutées. Dans la figure 2.3a, Kprobe n'optimiserait pas la trappe, car un saut cible une instruction qui serait modifiée. Dans la figure 2.3b, Kprobe est en mesure d'optimiser la trappe, car aucun saut ne cible ces instructions.

Pour s'assurer qu'aucun fils n'exécute les instructions pendant l'écriture du saut, Kprobe utilise un appel à la fonction Read-Copy-Update (RCU) spécialisée `synchronize_rcu_tasks`. Une fois cette synchronisation effectuée, le saut relatif peut remplacer la trappe originale.

Sans aller dans les détails, RCU est un mécanisme qui permet de libérer la mémoire en étant assuré qu'aucun coeur de l'Unité Centrale de Traitement (UCT) traitement n'ait encore une référence à l'objet libéré [43]. Pour ce faire, les accès à cet objet en lecture doivent être délimités par `rcu_read_lock()` et `rcu_read_unlock()`. Cette section critique s'assure qu'il n'y aura pas de changement de contexte. Par souci de performance, le lecteur ne devrait pas garder cette section critique trop longtemps. Lors de la suppression, un appel à `synchronize_rcu()` doit premièrement être fait. Celui-ci bloque jusqu'à ce que tous les coeurs actuels aient effectué un changement de contexte. Puisque ceux-ci sont désactivés dans la section critique RCU, il est garanti qu'il n'y a aucun lecteur dans la structure de données.

Dans le cas des sondes Kprobe, on ne peut pas assumer que la fonction instrumentée effectue des appels aux fonctions RCUs ci-dessus. Par conséquent, on ne peut pas se baser seulement sur les changements de contexte pour vérifier qu'aucun fil n'exécute une mauvaise instruction, car ceux-ci peuvent survenir à tout moment. Le mécanisme RCU-tasks ajoute la contrainte que les tâches doivent *volontairement* effectuer un changement de contexte [13]. De cette manière, toutes les tâches qui se seraient fait préempter en exécutant les mauvaises instructions sont garanties de s'être exécuté à nouveau et d'avoir éventuellement exécuté un changement de contexte volontairement.

2.1.3 Jprobe

Les sondes Jprobe, pour *jump probe*, permettent d'instrumenter l'entrée d'une fonction et de pouvoir facilement accéder à ses paramètres. Pour les installer, l'utilisateur doit premièrement créer une seconde fonction contenant l'instrumentation et possédant la même signature que la fonction à être instrumentée. Par la suite, Jprobe utilise une sonde Kprobe non optimisée pour instrumenter le début de la fonction. Dans l'instrumentation de la sonde Kprobe, Jprobe redirige l'exécution vers la seconde fonction avec les paramètres intacts. À la fin de la seconde fonction, un appel vers `jprobe_return` s'occupe de rediriger le flot d'exécution à la fonction originale à l'aide d'une autre trappe de débogage [35]. Jprobe utilise les trappes de débogage afin de pouvoir rediriger l'exécution dans un contexte différent sans effet secondaire. La figure 2.4 illustre le lancement d'une telle sonde.

Un avantage des sondes Jprobe est qu'il est très facile d'accéder aux paramètres de la fonction originale. Il n'est pas nécessaire d'avoir des routines spécifiques à l'architecture ciblée, le compilateur s'occupe de tout. Par contre, il existe plusieurs cas où ces sondes sont dangereuses.

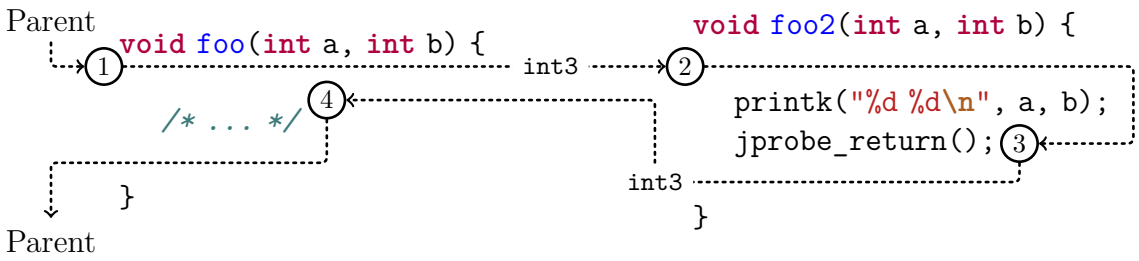


FIGURE 2.4 Exemple du lancement d'une sonde Jprobe. Lorsque la fonction est exécutée par son parent, le contrôle est immédiatement envoyé à la seconde fonction (1). Celle-ci effectue son instrumentation avec les arguments de la fonction originale (2). À sa fin (3), elle quitte et résume l'exécution de la fonction originale (4).

Premièrement, lorsque le compilateur construit la seconde fonction, il est libre d'optimiser à sa guise et d'écraser les paramètres, se trouvant sur la pile ou dans les registres, de la fonction originale. Pour conserver l'exécution originale de la fonction, les paramètres sont sauvegardés par Jprobe à l'entrée. Une portion de la pile, 64 octets sur x86, est copiée au début et rétablie à la fin de la seconde fonction. De plus, il y a toujours le risque que la définition d'une fonction change et que la deuxième ne soit pas ajustée. Dans ce cas, le comportement n'est pas défini et le programme pourrait planter. Finalement, cette méthode ne fonctionne pas pour les fonctions écrites en assembleur. Pour ces multiples raisons et autres, Jprobe est maintenant déprécié en faveur d'autres méthodes qui seront explorées plus loin [36].

2.1.4 Kretprobe

Kretprobe est une application de Kprobe servant à instrumenter l'entrée et la sortie d'une fonction dans le noyau Linux. Pour ce faire, Kretprobe insère une sonde Kprobe au début de la fonction voulue. Lorsque cette sonde est lancée, la fonction est détournée vers l'instrumentation. Pour ce faire, l'adresse de retour de la fonction est remplacée par celle de l'instrumentation de sortie. Selon l'architecture, ceci est possible en modifiant la pile d'appels ou un registre particulier. Dans l'instrumentation, l'utilisateur a accès aux registres et à la mémoire de la fonction originale. Pour accéder aux arguments ou à la valeur de retour, il doit utiliser manuellement la convention d'appel. Quand la fonction finit son exécution et retourne le contrôle à son parent, l'instrumentation est exécutée au lieu de la fonction parent. Celle-ci se charge ensuite de retourner le contrôle à la vraie fonction parent. La figure 2.5 illustre le flot d'exécution d'une sonde Kretprobe.

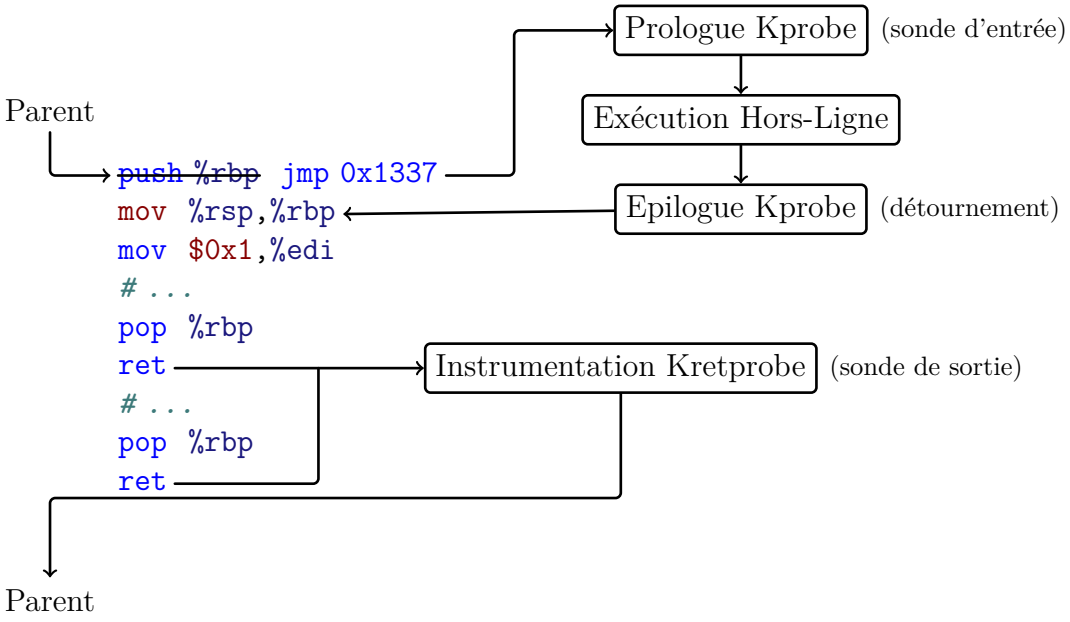


FIGURE 2.5 Exemple du lancement d'une sonde Kretprobe.

2.1.5 Uprobe

Uprobe est un sous-système du noyau Linux [38] permettant aux développeurs d'insérer des sondes entièrement dynamiques dans des programmes de l'espace utilisateur. Il est possible d'insérer des sondes en utilisant l'interface de fichiers de cet outil, mais celle-ci est peu flexible. Pour prendre pleinement avantage de Uprobe, les sondes doivent être insérées à partir du noyau. Lorsque la sonde est lancée, l'instrumentation appelée doit aussi se trouver dans le noyau.

Techniquement, Uprobe remplace l'instruction ciblée par une trappe de débogage. L'instruction est écrite directement dans la page mémoire du programme ou de la bibliothèque. L'instruction remplacée est exécutée hors-ligne dans un tampon après la routine d'instrumentation. Lorsque la sonde est lancée, l'interruption est gérée par le gestionnaire du noyau. Celui-ci détermine la sonde source et exécute l'instrumentation liée. Par la suite, il exécute hors-ligne l'instruction remplacée et retourne le contrôle au programme utilisateur [14]. La figure 2.6 illustre le flot d'exécution de la sonde.

Un avantage d'Uprobe est qu'il est relativement simple d'insérer des sondes entièrement dynamiques dans un programme utilisateur. Cela dit, les sondes ajoutent un surcout d'exécution considérable, car il y a des changements de contexte additionnels entre le noyau et l'espace utilisateur. De plus, tous les processus utilisant le programme ou la bibliothèque instrumentés sont affectés, car la page mémoire est modifiée directement [29]. Uprobe offre à l'utilisateur

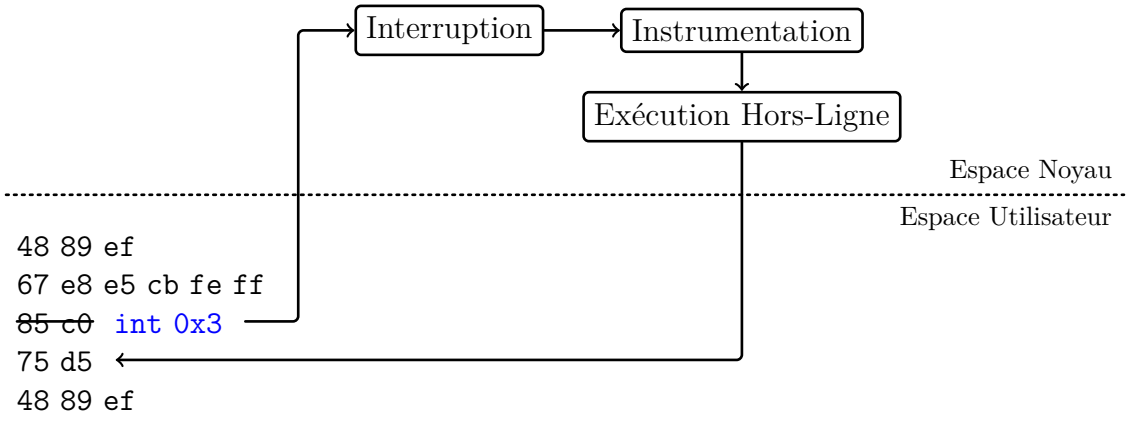


FIGURE 2.6 Exemple du lancement d'une sonde Uprobe.

de fournir une deuxième fonction qui filtre les évènements avant la fonction d'instrumentation. Cependant, la trappe de débogage est tout de même toujours exécutée pour chaque processus, même si ces évènements sont filtrés.

2.1.6 Perf Probe

Les sondes Perf sont une application des sondes Kprobe. L'avantage principal de ces sondes est qu'il est possible d'extraire la valeur des variables locales à une position dans la fonction. Pour ce faire, Perf doit calculer la position de la variable dans la mémoire ou dans un registre à une instruction précise. Perf précalcule l'expression DWARF liée à la position de la variable. Par la suite, il insère une sonde Kprobe avec sa propre instrumentation effectuant la lecture de la ou des variables.

2.1.7 Compilateur

Les différents compilateurs offrent plusieurs méthodes pour instrumenter des fonctions. Une option de base, supportée par la majorité des compilateurs, est d'ajouter `-pg` au programme [59]. Celle-ci instrumente chacune des fonctions du programme en ajoutant une sonde statique après le prologue. La figure 2.7 présente un exemple de cette instrumentation.

Historiquement, cette fonction était utilisée à des fins de profilage. Pour l'utiliser, une bibliothèque dynamique fournissant cette fonction est chargée dans l'espace mémoire du processus. Lorsqu'elle est appelée, elle sauvegarde l'adresse des fonctions parent et enfant, afin de construire un graphe d'appels [27]. Par la suite, on peut mesurer, ou déterminer, quelles fonctions sont le plus souvent exécutées ou ralentissent le programme. Cela dit, d'autres programmes ou bibliothèques l'utilisent ensuite pour faire du traçage, effectuant plus d'actions

```
push   %rbp
mov    %rsp,%rbp

mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(a) option désactivée

```
push   %rbp
mov    %rsp,%rbp
call   mcount@GLIBC_2.2.5
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(b) option activée

FIGURE 2.7 Fonction instrumentée avec `-pg`.

comme sauvegarder les paramètres d'entrées.

Le désavantage principal de `-pg` est qu'il est peu flexible et demande généralement plus de travail lorsqu'il faut extraire des données spécifiques de la fonction appelante. Pour remédier à cela, les compilateurs offrent maintenant aussi `-pg -mfentry`. La différence principale est que cette option ajoute la sonde statique avant le prologue de la fonction [59]. Cela permet à la sonde d'avoir directement accès à la fonction appelante et aussi de pouvoir détourner la fonction vers une autre plus facilement. La figure 2.8 présente un exemple de cette instrumentation.

```
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(a) option désactivée

```
call   __fentry__@GLIBC_2.13
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(b) option activée

FIGURE 2.8 Fonction instrumentée avec `-pg -mfentry`.

Si l'outil veut aussi instrumenter la sortie de la fonction, il doit détourner le contrôle à la sortie de la fonction. Pour ce faire, il peut modifier l'adresse de retour, qui se trouve sur la pile du programme, vers une autre fonction.

Il est aussi possible d'ajouter la troisième option `-mnop-mcount` afin de remplacer l'appel par un NOP, instruction nulle, de cinq octets [59]. Lorsque l'outil veut activer la sonde, il n'a qu'à remplacer le NOP par un saut de cinq octets vers l'instrumentation. L'ajout de

cette option est très semblable à `-pg -mfentry`. Elle est souvent préférée lorsque seulement un sous-ensemble des fonctions doit être instrumenté, afin de diminuer le surcout lié aux fonctions sans intérêt. La figure 2.9 présente un exemple de cette instrumentation.

```
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(a) option désactivée

```
nop    # 5 octets
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(b) option activée

FIGURE 2.9 Fonction instrumentée avec `-pg -mfentry -mnop-mcount`.

Le compilateur GNU offre l'option `-finstrument-functions`. Celle-ci ajoute des appels après le prologue et avant l'épilogue des fonctions [59]. L'avantage de cette méthode est qu'elle est très simple à utiliser. L'instrumentation peut être une fonction écrite en C, alors que les autres méthodes demandent d'avoir une partie en assembleur. Par contre, le surcout lié à cette instrumentation est assez élevé, en raison de la grande quantité d'instructions ajoutées. La figure 2.10 présente un exemple de cette instrumentation.

Afin d'obtenir la couverture du code, les compilateurs offrent l'option `-fprofile-arcs` [59] [39]. Pour chaque ligne de code source, un compteur est alloué dans une section du programme. Par la suite, quelques instructions sont ajoutées pour chaque ligne. Celles-ci s'occupent de calculer un pointeur vers le compteur associé à la ligne et d'ensuite l'incrémenter. Aucun appel de fonction n'est ajouté lors de cette instrumentation. Lorsque le programme se termine, un fichier est écrit par unité de compilation, contenant la valeur finale de chaque compteur. Outre pour la couverture de code, cette instrumentation peut être utile lorsque le programme est ensuite construit avec les optimisations guidées. Dans ce cas, le compilateur compile une seconde fois le programme en ajoutant des indications au processeur concernant quelles branches sont plus probables d'être exécutées.

2.1.8 Compilateur GNU

En plus des options décrites dans la section 2.1.7, GCC permet à l'utilisateur d'ajouter des plug-ins additionnels durant la compilation d'un fichier. Ces plug-ins ajoutent des passes pouvant effectuer d'autres types d'optimisation [32], effectuer des analyses statiques addi-

```
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(a) option désactivée

```
push   %rbp
mov    %rsp,%rbp
push   %rbx
sub    $0x8,%rsp
mov    0x8(%rbp),%rax
mov    %rax,%rsi
lea    -0x17(%rip),%rdi
call   __cyg_profile_func_enter@plt
mov    $0x0,%edi
call   foo
mov    %eax,%ebx
mov    0x8(%rbp),%rax
mov    %rax,%rsi
lea    -0x36(%rip),%rdi
call   __cyg_profile_func_exit@plt
mov    %ebx,%eax
mov    -0x8(%rbp),%rbx
pop    %rbp
ret
```

(b) option activée

FIGURE 2.10 Fonction instrumentée avec `-finstrument-functions`.

tionnelles [25], ajouter de l'instrumentation personnalisée [40] et encore plus. Un avantage du système de plug-in GCC est qu'il offre une API relativement stable au programmeur. Celui-ci peut donc développer des plug-ins personnalisés à un projet qui seront utilisés à long terme. Par exemple, le noyau Linux offre dans son code source une panoplie de plug-ins GCC, la majorité afin d'augmenter la fiabilité et la sécurité de celui-ci [12].

2.1.9 Compilateur LLVM

En plus des options décrites dans la section 2.1.7, le compilateur de l'infrastructure LLVM offre l'instrumentation XRay [39] développée par Google [5]. XRay permet d'instrumenter les entrées et les sorties de chaque fonction dans un programme. Pour ce faire, il octroie onze octets pour l'instrumentation de l'entrée ou de la sortie de chaque fonction. Un outil externe, comme un traceur ou un profileur, peut les remplacer afin de modifier le flot d'exécution du programme.

Avant l'épilogue de la fonction, XRay insère un NOP de neuf octets avec un saut de deux octets afin de le contourner. Ceci permet à l'outil de modifier neuf octets en toute sécurité

sans se soucier des fils d'exécutions en parallèle pouvant exécuter la fonction en même temps. Une fois ces neuf octets modifiés, les deux octets avant peuvent maintenant être remplacés atomiquement pour activer la sonde. Après l'épilogue de la fonction, dix octets forment un NOP. De la même manière, ceux-ci peuvent être modifiés sécuritairement et l'instruction de retour de un octet peut ensuite être modifiée atomiquement. La figure 2.11 donne un exemple de l'instrumentation XRay.

```
push   %rbp
mov    %rsp,%rbp
mov    $0x0,%edi
call   foo
pop    %rbp
ret
```

(a) option désactivée

```
           jmp    prologue   # 2 octets
           nop               # 9 octets
prologue:  push   %rbp
           mov    %rsp,%rbp
           mov    $0x0,%edi
           call   foo
           pop    %rbp
           ret                # 1 octet
           nop                # 10 octets
```

(b) option activée

FIGURE 2.11 Fonction instrumentée avec XRay.

Un avantage de cette méthode est qu'un saut absolu de huit octets peut être effectué afin d'aller à l'instrumentation. Par conséquent, il est possible de changer le flot d'exécution vers n'importe quelle adresse dans la mémoire. De plus, le surcout est pratiquement nul quand l'instrumentation est désactivée, car seulement un seul saut au prologue serait exécuté. Par contre, cette méthode ajoute 21 octets de plus à chaque fonction. Si le programme est très large et contient beaucoup de fonctions, XRay ajoute un grand surcout en matière d'espace.

LLVM supporte également des plug-ins personnalisés. Ceux-ci ajoutent différentes passes durant la compilation d'un logiciel. Ils ont été utilisés afin d'ajouter de l'instrumentation personnalisée [60], d'effectuer des analyses statiques [53] et encore plus.

2.1.10 Ptrace

L'appel système `ptrace` est utilisé afin de permettre à un processus d'observer ou de contrôler un autre processus [48]. Par exemple, un débogueur doit pouvoir écrire dans la mémoire d'un autre processus afin d'y insérer des trappes de débogage. De plus, il doit pouvoir lire ou écrire dans les registres de l'autre processus. Il doit aussi pouvoir avancer instruction par instruction le programme, remplacer les signaux ou attendre un appel système.

Tout cela peut être effectué avec `ptrace` et ses différentes options. Par défaut, un processus parent est en mesure de tracer ses processus enfants avec cet appel système. Pour tracer d'autres processus, celui-ci doit obtenir plus de privilèges. Les options de cet appel système sont assez bas niveau. Pour effectuer des tâches complexes, il faut effectuer plusieurs appels. Par conséquent, `ptrace` peut ajouter un surcout considérable, selon son utilisation [11].

2.2 Outils de Traçage

2.2.1 Perf

Perf est un sous-système du noyau Linux permettant de lire ou compter différents évènements à travers le système entier [44]. Il utilise plusieurs sources d'évènements différentes pour obtenir des données. Celui-ci est disponible dans la majorité des distributions de Linux.

À la base, Perf permettait seulement de lire les compteurs matériels de l'UCT. Par exemple, les plateformes Intel offrent des compteurs de performance. Il est possible de compter le nombre de fautes de cache, d'instructions exécutées et plus. Les processeurs Intel offrent aussi les moniteurs de performance. Ceux-ci permettent de compter le nombre de branchements manqués, d'interruptions lancées et encore plus.

À travers les années, d'autres sources d'évènements furent ajoutées, dont les sources évènements logiciels du noyau. Celles-ci sont des compteurs purement logiciels qui ont été ajoutés dans le noyau Linux. Perf offre, par exemple, de compter le nombre de changements de contexte, de fautes de page et encore plus.

Finalement, Perf permet aussi de compter le nombre de lancements des différentes sondes statiques et entièrement dynamiques. Pour ce faire, il utilise l'interface de Ftrace qui, elle-même, utilise Kprobe et les sondes statiques. Cela offre l'option à l'utilisateur de compter, par exemple, le nombre d'appels système particuliers, d'appels à une fonction du noyau ou plus.

Un avantage de Perf est qu'il est très simple à utiliser. Par exemple, il est très facile de comparer le nombre de fautes de cache entre deux versions d'une même fonction. Cepedant, les compteurs ne sont souvent pas suffisants pour déboguer des problèmes, par exemple, de performance. Ils nous permettent de les détecter, mais des outils plus puissants doivent être utilisés pour bien analyser le problème.

Pour ce faire, Perf est aussi en mesure d'effectuer du traçage dynamique. A cette fin, il utilise les sondes Perf basées sur Kprobe ou Uprobe afin d'instrumenter une adresse mémoire. À chaque lancement de la sonde, un évènement est généré et mis dans un tampon. Celui-ci

peut contenir la valeur d'une variable ou d'un argument à la position de la sonde. À la fin du traçage, les évènements peuvent être consultés. Comme outil de traçage, Perf est facile à utiliser et installer. Ses sondes dynamiques sont puissantes et rapides dans le noyau. Par contre, il manque de flexibilité sur ce qu'il trace. Par exemple, pour instrumenter les allocations mémoires d'un processus, il peut instrumenter la fonction `malloc()` dans la bibliothèque C. Par contre, toutes les allocations du système seront tracées et celui-ci en effectue à haute fréquence. Puisque les sondes utilisateurs sont basées sur des trappes de débogage, le système en entier sera considérablement ralenti.

2.2.2 LTTng

Linux Trace Toolkit – nouvelle génération (LTTng) est un traceur de haute performance pour Linux. Il offre la possibilité d'instrumenter le noyau ou des programmes utilisateurs [18]. Pour le noyau, il est en mesure d'utiliser différentes sondes existantes. Par exemple, il utilise les sondes statiques [17] pour enregistrer les appels système et leur argument. Il peut aussi utiliser les sondes entièrement dynamiques de Kprobe pour instrumenter des adresses spécifiques dans le noyau. Ou encore, il peut instrumenter l'entrée et la sortie des fonctions du noyau à l'aide des sondes Kretprobe.

Pour instrumenter un programme utilisateur, LTTng offre différents types de sonde. Les sondes préférées sont les sondes statiques de LTTng. Pour les utiliser, le développeur doit utiliser les macros offertes par LTTng afin de créer la définition d'un évènement. Par la suite, il peut ajouter les sondes manuellement à travers le code source afin de lancer l'évènement et lui donner des valeurs. La figure 2.12 illustre la définition et le lancement d'une sonde statique simple dans LTTng. Durant l'exécution du programme, ces sondes peuvent être désactivées ou activées à tout moment. Lorsqu'elle ne sont pas activées, le surcout d'exécution est négligeable. LTTng supporte aussi les sondes similaires provenant de SystemTap.

Outre les sondes statiques, LTTng permet aussi au programmeur d'insérer des sondes entièrement dynamiques basées sur Uprobe. Pour ce faire, il peut spécifier une adresse mémoire ou un numéro de ligne dans un fichier source. Cela permet au développeur d'instrumenter son programme afin de rapidement obtenir de l'information sur un problème. Ces sondes sont plus lentes et ne permettent pas de lire des données. Il peut ainsi être préférable d'utiliser des sondes statiques.

Finalement, LTTng est aussi en mesure d'instrumenter les entrées et sorties des fonctions dans la mesure où le programme fut compilé avec `-finstrument-functions`. Il ne supporte pas les autres sondes statiques insérées par les compilateurs comme XRay.

```
TRACEPOINT_EVENT(
    counter,
    values,
    TP_ARGS(counter_t*, c),
    TP_FIELDS(
        ctf_integer(int, value, c->value)
        ctf_integer(int, sum, c->sum)
    )
)
```

(a) définition de l'évènement

```
typedef struct counter {
    int current;
    int sum;
} counter_t;

void foo(counter_t* c) {
    for (int i = 0; i < 1000; i++) {
        c->current = i;
        tracepoint(counter, values, c);
    }
}
```

(b) lancement de la sonde

FIGURE 2.12 Exemple de l'utilisation des sondes statiques de LTTng. La figure 2.12a présente la définition d'un évènement contenant deux champs entiers. La figure 2.12b présente le code qui crée cet évènement avec les valeurs d'une structure.

La haute performance du traceur provient des différentes optimisations effectuées lors de la génération d'un évènement de trace. À la base, les évènements sont sérialisés dans un tampon circulaire. Ce dernier est divisé en plusieurs sous-tampons. Lorsque des sondes écrivent dans un sous-tampon, un autre fil d'exécution s'occupe d'enregistrer les évènements d'un autre sous-tampon au disque. De plus, chaque coeur de l'Unité Centrale de Traitement (UCT) écrit dans son propre tampon circulaire. Cela évite d'avoir de la synchronisation dans cette structure de données et diminue aussi les fautes de cache. De plus, chaque évènement est enregistré sous le format Common Trace Format (CTF), un format binaire, ouvert et optimisé pour le traçage [55].

2.2.3 DTrace

DTrace a originalement été un traceur rapide et flexible pour la plateforme Solaris. À la base, le développeur doit écrire un script avec le langage D. Celui-ci est ensuite converti en C, compilé, inséré dans le noyau [1] et lié à une sonde. Le script spécifie quelles sondes devraient être utilisées et aussi l'instrumentation liée à leur exécution. Par conséquent, l'utilisateur est en mesure de filtrer, transformer ou enregistrer des données particulières à chaque exécution des sondes. La grande force de DTrace est qu'il est flexible et rapide d'instrumenter et d'analyser un système.

Son grand succès l'amène à être porté vers d'autres plateformes. Sous Linux, deux versions différentes ont été créées. La première s'appelle *dtrace4linux* et fonctionne essentiellement comme l'originale. La deuxième s'appelle Linux DTrace et est maintenue par Oracle. La

grande différence de celle-ci est que les scripts D ne sont pas compilés en tant que module Linux, mais bien en tant que programme BPF. Un grand avantage à cette méthode est que l'instrumentation est beaucoup plus sécuritaire et empêche de planter le noyau au complet, car les programmes sont exécutés dans un bac à sable.

Pour les deux implémentations sous Linux, en général, DTrace est en mesure de tracer le noyau ainsi que des logiciels utilisateurs. Pour *dtrace4linux*, différentes sources d'instrumentations sont utilisées pour y attacher des programmes. Entre autres, le traceur utilise les sondes statiques du noyau, les sondes entièrement dynamiques de Kprobe, les évènements de Perf et encore plus.

DTrace permet aussi à l'utilisateur de définir ses propres sondes statiques pour usage dans un module noyau ou un programme utilisateur. Elles sont similaires à celles de LTTng en matière d'utilisation. Par contre, la définition doit être écrite avec le langage D. Celle-ci doit être traduite en C pour l'utiliser dans l'application. La figure 2.13 illustre l'équivalent en DTrace de l'exemple de LTTng à la figure 2.12.

L'avantage de ces sondes est qu'elles n'ajoutent virtuellement aucun surcout quand elles sont désactivées. Contrairement à LTTng qui désactive ses sondes avec un branchement, les sondes de DTrace sont désactivées en étant remplacées par un ou des NOPs [47]. Pour une sonde statique dans le noyau, un NOP de un octet suivi d'un NOP de cinq octets est inséré. DTrace utilise ensuite Kprobe pour l'instrumenter. Pour une sonde statique dans un programme utilisateur, DTrace utilise les sondes de SystemTap, expliqué à la prochaine section, en insérant un seul octet. Par contre, elles ajoutent un grand surcout d'exécution, car elles utilisent une trappe de débogage.

```
provider counter {
    probe values(int, int);
};
```

(a) définition de l'évènement

```
void foo() {
    int sum = 0;
    for (int i = 0; i < 1000; i++) {
        sum += i;
        DTRACE_PROBE2(counter, values, i, sum);
    }
}
```

(b) lancement de la sonde

FIGURE 2.13 Exemple de l'utilisation des sondes statiques de DTrace. La figure 2.13a présente la définition d'un évènement contenant deux champs entiers dans le language D. La figure 2.13b présente le code qui crée cet évènement avec les valeurs de deux variables locales.

2.2.4 SystemTap

SystemTap est comparable à DTrace. L'utilisateur écrit un script SystemTap, celui-ci est ensuite converti en C et compilé en tant que module pour le noyau. Par la suite, le module est inséré dans le noyau et configuré pour s'exécuter lors des évènements spécifiés [54]. Comme source d'évènement, il est possible d'utiliser les sondes statiques du noyau. Celles-ci permettent, par exemple, de s'abonner aux entrées/sorties des appels système ou à d'autres évènements spécialisés. SystemTap utilise aussi les sondes entièrement dynamiques du noyau afin de pouvoir souscrire, par exemple, aux entrées/sorties des différentes fonctions présentes ou même des adresses ou lignes de code arbitraires dans le noyau.

L'outil est aussi en mesure de tracer des processus dans l'espace utilisateur de la même manière. Il utilise les sondes statiques dans le noyau pour obtenir des évènements sur un processus dont le début/fin d'un processus ou ses appels système. Il est en mesure d'instrumenter une adresse mémoire ou ligne de code arbitraire à l'aide des sondes entières dynamiques utilisateurs d'Uprobe.

SystemTap offre aussi des sondes statiques pour l'utilisateur. Pour ce faire, l'utilisateur doit utiliser les macros `STAP_PROBE`n, où n est le nombre d'arguments, afin de définir une sonde. Lorsque le programme est compilé, un NOP de un octet est ajouté à l'endroit de la sonde. La figure 2.14 illustre un exemple de ces sondes.

De l'information est aussi ajoutée à une section dans le programme lors de la compilation. Un outil externe, comme SystemTap, peut ensuite lire cette section afin de déterminer quelles sondes sont présentes dans le programme et leur attacher de l'instrumentation. Un désavantage de cette méthode est que lorsque la sonde est activée, une trappe de débogage est insérée afin d'exécuter son instrumentation. Cela est simple à implémenter, mais ajoute un surcout considérable, surtout pour un programme utilisateur, en raison des changements de contexte. Par contre, la sonde n'ajoute virtuellement aucun surcout lorsqu'elle est désactivée, en raison de l'utilisation d'un NOP.

2.2.5 Ftrace

Ftrace est le traceur de fonction inclus dans le noyau Linux. Il supporte la majorité des architectures existantes [51]. Celui-ci utilise les sondes statiques insérées par le compilateur (`-pg` et `-mfentry`) afin de générer des évènements d'entrées de fonction. Il détourne aussi le pointeur de retour des fonctions afin de générer des évènements de sorties de fonction [23]. La routine d'instrumentation peut accéder aux arguments de la fonction afin de les sauvegarder. Par la suite, l'information est disponible à travers une interface de fichiers dans le système.

```
int sum = 0, i;
for (i = 0; i < 100; i++) {
    sum += i;
    STAP_PROBE2(foo,
        values, i, sum);
}
return sum;
```

(a) lancement de la sonde

```
movl $0x0,-0x4(%rbp)
jmp  1157
mov  -0x4(%rbp),%eax
add  %eax,-0x8(%rbp)

addl $0x1,-0x4(%rbp)
cmpl $0x3e7,-0x4(%rbp)
jle  114d
```

(b) sans la sonde

```
movl $0x0,-0x4(%rbp)
jmp  1158
mov  -0x4(%rbp),%eax
add  %eax,-0x8(%rbp)
nop
addl $0x1,-0x4(%rbp)
cmpl $0x3e7,-0x4(%rbp)
jle  114d
```

(c) avec la sonde

FIGURE 2.14 Exemple de l'utilisation des sondes statiques de SystemTap. La figure 2.14a donne un exemple d'un programme avec une sonde. La figure 2.14b donne les instructions de ce programme si la sonde n'était pas compilée. La figure 2.14c donne les instructions si la sonde était compilée. On peut y voir un NOP d'un octet ajouté.

De base, Ftrace trace toutes les fonctions afin de pouvoir générer un graphe d'appels par la suite. Cela dit, il offre d'autres configurations pour des analyses spécifiques. Par exemple, il offre la configuration `blk` qui ne trace que les fonctions d'intérêts dans le sous-système de la couche bloc du noyau.

2.2.6 Uftrace

Uftrace est un traceur de fonction, comme Ftrace, mais pour l'espace utilisateur. Il supporte les architectures x86 et ARM [37]. D'une manière comparable à Ftrace, il supporte les mêmes sondes statiques insérées par le compilateur, mais aussi `-finstrument-functions` et XRay de LLVM. Il détourne aussi l'exécution afin d'instrumenter la sortie d'une fonction, si nécessaire, et il offre la possibilité d'enregistrer les arguments ou la valeur de retour de la fonction.

Il est aussi possible d'instrumenter un logiciel à l'aide de sondes entièrement dynamiques. Pour ce faire, Uftrace utilise une technique similaire à celle des sondes optimisées de Kprobe, c'est-à-dire qu'il remplace une ou des instructions par un saut de cinq octets sous x86. Les instructions remplacées sont aussi exécutées ou simulées dans un tampon de mémoire. Pour une exécution sécuritaire, Uftrace s'assure qu'aucun saut direct ne cible les quatre derniers octets du saut inséré. De plus, les sondes sont seulement insérées avant l'exécution du programme. Par conséquent, le programme n'a pas à être modifié pendant son exécution. Cela supprime la majorité des défis liés à l'instrumentation dynamique. Il n'est pas nécessaire d'arrêter ou de déplacer un autre fil pour l'empêcher d'exécuter une mauvaise instruction. L'outil n'offre pas la possibilité de supprimer les sondes existantes.

Lorsqu'un programme est tracé par Uftrace, l'entièreté du traçage est effectuée par une bibliothèque appelée `libmcount.so`. Celle-ci fournit les fonctions nécessaires au fonctionnement des sondes statiques insérées par le compilateur, comme celles de `-pg`, `-mfentry`, `-mnop-mcount`, `-finstrument-functions` et XRay. Elle s'occupe aussi d'effectuer des sondes entièrement dynamiques et de lire les arguments ou les valeurs de retour. Afin d'insérer cette bibliothèque, Uftrace crée un nouveau processus et configure la variable d'environnent `LD_PRELOAD`. Lorsque l'initialisation de la bibliothèque C est exécutée par le programme, celle-ci charge la bibliothèque dans l'espace mémoire et exécute son constructeur. D'autres paramètres sont aussi envoyés à l'aide d'autres variables d'environnement. La figure 2.15 illustre le démarrage d'un programme sous Uftrace.

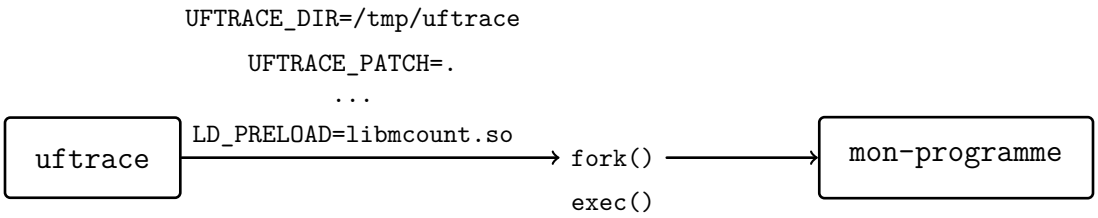


FIGURE 2.15 Démarrage d'un processus sous Uftrace.

2.2.7 eBPF

Berkeley Packet Filter (BPF) est une technologie permettant à l'utilisateur d'exécuter un programme sécuritairement et rapidement dans une machine virtuelle à l'intérieur du noyau [10]. À la base, elle était utilisée afin de pouvoir modifier les règles de filtrage dans le sous-système réseau. Dans celles-ci, il est important que le filtrage soit rapide, donc dans le noyau, mais il doit quand même être flexible afin de pouvoir facilement le changer sans recompiler le noyau ou redémarrer le système. Une machine virtuelle avec son propre assembleur est idéale, car cela permet d'exécuter le code, provenant de l'espace utilisateur, et de restreindre ses accès sécuritairement.

De nos jours, le extended Berkeley Packet Filter (eBPF) est utilisé. Celui-ci applique les principes de base de BPF à une panoplie de cas d'utilisations différents dans le noyau dont le traçage. Par exemple, il est possible d'attacher un programme eBPF sur des sondes statiques du noyau ou entièrement dynamiques du noyau (Kprobe). Cela peut être utile afin d'exécuter une analyse très spécifique sur plusieurs noyaux différents, sans les modifier. Cela dit, eBPF a beaucoup évolué depuis BPF. Sa machine virtuelle est maintenant plus complexe et offre beaucoup plus de fonctionnalités. L'exécution d'un programme dans une machine virtuelle

augmente la fiabilité du système, contrairement à SystemTap, par exemple, où il causait historiquement plus de fautes non-recouvrable dans le noyau [28].

2.2.8 Strace

Le programme `strace` est un traceur d'appel système. Celui-ci est basé entièrement sur l'utilisation de l'appel système `ptrace` [52]. Avec celui-ci, il est en mesure de configurer un processus afin qu'il s'arrête avant ou après un appel système. Une fois le processus arrêté, `strace` est en mesure d'obtenir l'appel système exécuté et de l'enregistrer avec ses paramètres dans un tampon. Par la suite, le processus continue son exécution.

Un avantage de `strace` est qu'il est simple et rapide d'obtenir tous les appels système d'un processus et ses enfants. Par contre, l'utilisation de `ptrace` fait en sorte que son surcout, par rapport au programme sans instrumentation, est très élevé.

2.2.9 GDB

Le débogueur GNU (GDB) est probablement le débogueur le plus répandu dans l'écosystème Linux. Il utilise `ptrace` pour insérer des trappes de débogage et attendre les évènements d'un autre processus [24]. Toutefois, cette approche ajoute beaucoup de surcout en raison de la communication et de la synchronisation entre les deux processus. Par conséquent, il n'est pas toujours possible de reproduire certaines classes de problèmes, comme les situations de compétition pour accéder une ressource, car celles-ci sont dépendantes de la vitesse d'exécution.

Pour remédier à ce problème, GDB offre la possibilité d'avoir un agent à l'intérieur du processus ciblé [16]. Pour ce faire, l'outil est en mesure d'injecter une bibliothèque dans l'espace mémoire du processus. Le constructeur de la bibliothèque peut ensuite démarrer un fil d'exécution supplémentaire pouvant recevoir des commandes ou envoyer des données à GDB. Cet agent ne remplace pas les fonctionnalités de base de l'outil, mais il permet à l'utilisateur d'ajouter des sondes entièrement dynamiques.

Afin d'être rapides, les sondes remplacent une instruction d'au moins cinq octets par un saut de cinq octets vers l'instrumentation. L'utilisateur peut choisir les actions à effectuer lorsqu'une sonde est exécutée. Par exemple, il peut demander de sauver des registres particuliers ou des variables du programme. Pour sauver les variables, il utilise les expressions DWARF afin de résoudre l'adresse dans la mémoire [22].

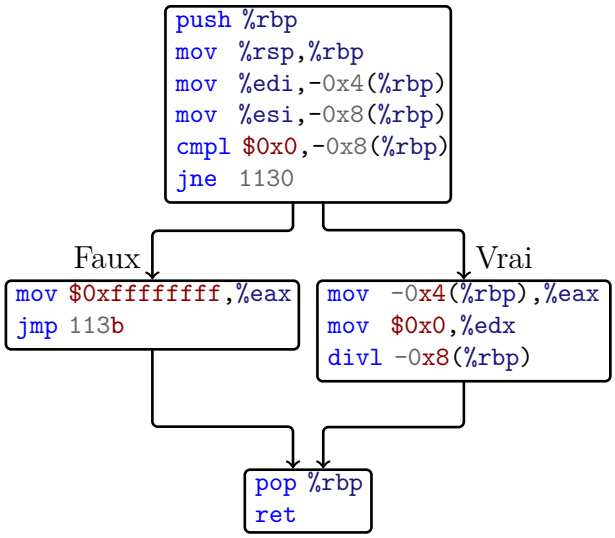
2.2.10 DynInst

DynInst est un outil permettant d'instrumenter un logiciel au niveau du Graphe de Flot de Contrôle (GFC). Le GFC est une abstraction utilisée pour décrire et analyser les différents chemins d'exécutions dans un programme. Un noeud représente une série d'instructions linéaire, ou un bloc, sans branchement interne, ayant seulement une entrée et une sortie. Une arête lie deux blocs ensemble, par exemple avec un saut ou directement séquentiels, l'un après l'autre. La figure 2.16 montre un exemple de GFC.

```
int foo(uint a, uint b) {
    if (b == 0) return -1;
    return a / b;
}
```

```
push %rbp
mov  %rsp,%rbp
mov  %edi,-0x4(%rbp)
mov  %esi,-0x8(%rbp)
cmpl $0x0,-0x8(%rbp)
jne  1130
mov  $0xffffffff,%eax
jmp  113b
mov  -0x4(%rbp),%eax
mov  $0x0,%edx
divl -0x8(%rbp)
pop  %rbp
ret
```

(a) source et assembleur



(b) graphe de flot de contrôle

FIGURE 2.16 La figure 2.16a donne un exemple de code C d'une fonction simple et ses instructions après être compilée. La figure 2.16b illustre le GFC extrait en utilisant l'outil Radare2.

Pour instrumenter un programme, DynInst ne permet pas d'instrumenter une instruction spécifique. Il offre seulement la possibilité d'instrumenter à un point d'instrumentation. Celui-ci est défini comme étant soit une entrée ou sortie de fonction, le début ou la fin d'une boucle, le début ou la fin d'une itération ou le début ou la fin d'un bloc [4]. En résumé, il remplace une arête entre deux blocs par au moins deux autres arêtes allant vers d'autres noeuds contenant l'instrumentation. La figure 2.17 illustre un exemple.

Pour remplacer une arête, DynInst remplace une instruction par un saut relatif de cinq octets. Lorsque celui-ci veut instrumenter l'entrée ou la sortie d'une fonction, il déplace, ou copie, la fonction ailleurs dans la mémoire. La fonction originale est remplacée par un saut vers

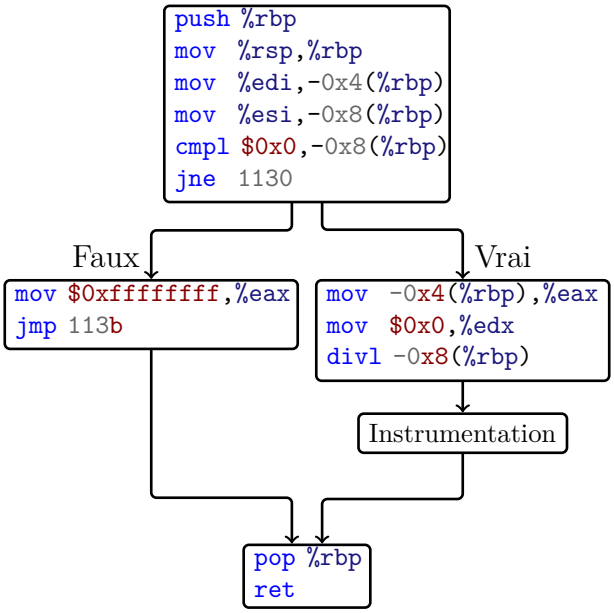


FIGURE 2.17 GFC de la figure 2.16b augmenté après avoir été instrumenté par DynInst. Une arête a été remplacée par deux afin de rediriger l'exécution vers l'instrumentation.

l'instrumentation et ensuite la fonction copiée. À chaque fois que des instructions originales sont déplacées, DynInst doit les ajuster afin de compenser les adressages relatifs.

Afin d'insérer son instrumentation, l'utilisateur doit premièrement la concevoir. Il peut l'écrire directement avec l'API C++ de DynInst afin de créer manuellement un GFC contenant l'instrumentation. Il peut aussi écrire l'instrumentation en DynC, similaire au C [6].

Un avantage de DynInst est qu'il réussit à instrumenter un programme avec un très grand taux de succès. Cependant, son exécution ajoute un très grand surcout pour l'instrumentation. Cela est dû aux analyses effectuées sur le GFC et sur le fait que les modifications du processus sont effectuées à l'aide de `ptrace`. Finalement, le processus instrumenté est aussi arrêté durant l'instrumentation, ce qui rend les instrumentations fréquentes non désirables.

2.2.11 Valgrind

Valgrind peut être vu comme un outil permettant d'écrire des outils d'analyses. Plus spécifiquement, il offre un API permettant d'instrumenter un logiciel durant son exécution. L'instrumentation est ajoutée durant l'exécution du logiciel pour chaque bloc d'instructions. Un bloc est une série d'instructions linéaire ayant à la fin un branchement. Lorsque le programme s'exécute et arrive sur un bloc non instrumenté, Valgrind arrête l'exécution, crée un nouveau bloc avec l'instrumentation et poursuit l'exécution dans ce dernier. En pratique, les

instructions originales ne sont jamais exécutées, le programme exécute toujours une copie des instructions qui a été instrumentée.

Pour instrumenter un bloc, Valgrind commence par le désassembler dans l'architecture originale. Ces instructions sont ensuite traduites dans une représentation intermédiaire définie par Valgrind. Cette représentation simule une machine implémentant un jeu d'instruction réduit à deux adresses utilisant des registres virtuels. Par la suite, Valgrind effectue des optimisations de base sur les instructions intermédiaires. Il fait ensuite appel à l'outil d'instrumentation afin d'insérer l'instrumentation voulue. De plus, il alloue les registres virtuels à des registres physiques de l'architecture originale. Finalement, il traduit, ou génère, la représentation intermédiaire vers l'architecture de la machine. Chaque bloc traduit est mis dans une table de correspondance à taille fixe. Lorsque celle-ci est près de sa capacité, de vieux blocs traduits sont enlevés et devront être traduits à nouveau plus tard [46].

Lorsque deux blocs peuvent être enchainés l'un après l'autre, la dernière instruction du premier bloc contient un saut vers le deuxième. Si le bloc suivant n'a pas encore été traduit, alors l'instruction fait appel à Valgrind pour le traduire. Lorsque deux blocs ne peuvent pas être enchainés, le premier bloc finit en appelant Valgrind. Celui-ci vérifie dans une table de correspondance le prochain bloc qui doit être exécuté.

Valgrind n'a pas besoin d'extraire les sauts indirects lors de son analyse pour obtenir les blocs. Lorsque le programme exécute un saut indirect, Valgrind intercepte la cible. Si la cible pointe vers du code qui a déjà été traduit, alors il l'exécute. Si la cible pointe vers le milieu d'un bloc qui a été traduit, alors Valgrind traduit à nouveau le bloc en démarrant à la cible du saut indirect.

Valgrind par lui-même n'instrumente pas l'application. Pour ce faire, il faut lui fournir l'outil qui effectue l'instrumentation. L'outil probablement le plus utilisé est Memcheck. Durant la traduction des blocs, celui-ci instrumente chaque accès en lecture ou écriture de la mémoire. Il instrumente et garde une trace de la mémoire qui a été allouée ou libérée. Avec cette instrumentation, il est en mesure de détecter les mauvais accès vers de la mémoire, par exemple non allouée. Il peut aussi détecter les fuites de mémoire à la sortie du programme.

Un autre outil digne de mention est Helgrind. Durant la traduction des blocs, celui-ci instrumente aussi tous les accès mémoires, mais en plus les fonctions de synchronisation comme `pthread_mutex_lock()` et `pthread_mutex_unlock()`. Son analyse se concentre sur les accès vers la même adresse mémoire par des fils différents, n'ayant pas d'appels à des fonctions de synchronisation entre ces accès. Par conséquent, il est en mesure de détecter des situations de concurrence possibles.

Valgrind est un outil puissant et l'utilisation de ses outils est simple. Par contre, la traduction des blocs ajoute un surcout énorme, selon l'outil utilisé. De plus, il est très difficile d'écrire un outil personnalisé, car cela implique d'interfacer avec le programme afin d'y insérer des instructions personnalisées, le tout avec le jeu d'instructions personnalisé. Il est possible d'effectuer des appels vers une fonction en C, mais celle-ci va elle aussi devoir être traduite par bloc.

2.3 Techniques d'Instrumentation Dynamique

Kprobe est souvent considéré comme étant l'état de l'art pour l'instrumentation dynamique. Toutefois, le taux de succès de ses sondes optimisées est réduit, en partie, à cause de son incapacité à résoudre les sauts indirects. Lorsqu'il analyse une fonction pour découvrir les cibles des différents sauts, Kprobe extrait le graphe de flot de contrôle de la fonction. Dans cette section, on explore d'autres techniques d'instrumentation dans la littérature n'ayant pas besoin d'extraire ce GFC et par conséquent, les sauts indirects.

2.3.1 NOProbe

La technique NOProbe évite de retrouver les sauts indirects en profitant du fait que les fonctions sont souvent suivies de rembourrage afin d'aligner la fonction suivante sur une ligne de cache [3].

L'idée principale de cette méthode est d'utiliser un branchement avec un déplacement de un octet lorsqu'un branchement avec déplacement de quatre octets ne peut pas être utilisé. Ce branchement, pouvant seulement atteindre des adresses relatives à -128 ou 127 octets, cible un second branchement de quatre octets se trouvant dans le rembourrage de la fonction. Ce dernier cible l'instrumentation liée à la sonde. La figure 2.18 résume le lancement d'une telle sonde. Un appel de fonction est utilisé pour le premier branchement, afin de permettre à l'instrumentation de résoudre la sonde qui a été lancée. L'instruction remplacée est bien sûr exécutée hors-ligne afin de conserver le comportement du programme.

Si la fonction est grande, par exemple plus de 255 octets, alors une grande quantité de sauts n'atteint pas le rembourrage à la fin de la fonction dernière ou courante. Néanmois, lorsque possible, cette sonde est considérablement plus rapide qu'une sonde basée sur une trappe. Par contre, l'instrumentation doit résoudre quelle sonde a été appelée, ce qui ajoute un peu de surcout. Malgré tout, le surcout additionnel reste négligeable par rapport à une sonde basée sur un saut de cinq octets.

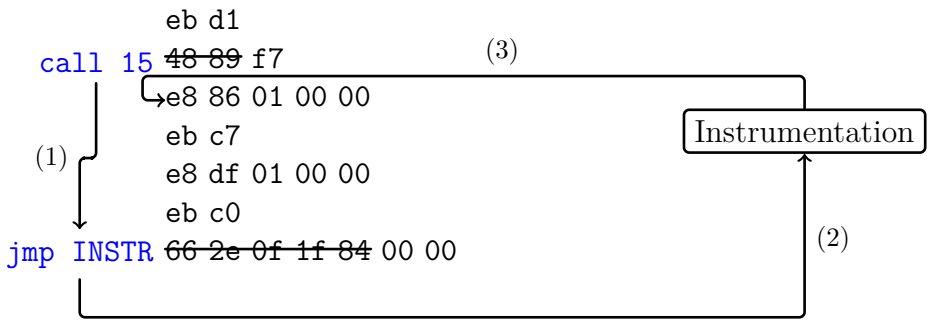


FIGURE 2.18 Exemple du lancement d'une sonde NOProbe. Lorsque la sonde est lancée (1), celle-ci saute premièrement dans le rembourrage. Un deuxième saut est exécuté afin de rediriger l'exécution vers l'instrumentation (2). Lorsque celle-ci se termine, l'exécution se résume à la prochaine instruction.

2.3.2 Dyntrace

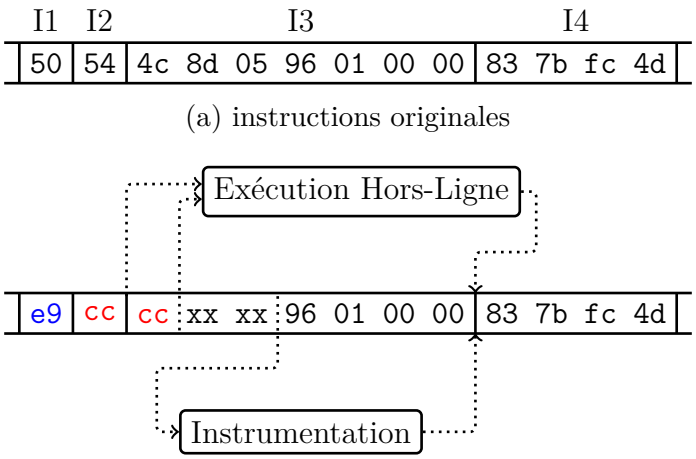
Comme NOProbe, Dyntrace utilise une technique supprimant le besoin de retrouver les sauts indirects. Pour ce faire, il utilise une méthode similaire à Kprobe, il insère des sauts de cinq octets vers l'instrumentation. Lorsque le saut est plus grand que l'instruction, il doit remplacer plusieurs instructions.

Si un saut indirect est présent dans la fonction actuelle, celui-ci n'est pas retrouvé et la sonde est tout de même insérée. Cependant, l'adresse cible de la sonde est choisie de sorte que des trappes de débogage y sont présentes au début de chaque instruction remplacée, après la première [30]. Si un saut ailleurs dans la fonction cible une des instructions qui a été remplacée, alors une trappe est exécutée. Le gestionnaire de la trappe s'occupe d'exécuter hors-ligne les instructions manquantes et de continuer l'exécution après la sonde.

Dyntrace exploite le fait qu'il est peu probable qu'un saut indirect, dans la fonction, cible exactement une instruction remplacée. Par conséquent, on évite d'avoir recours à une trappe de débogage comme sonde. Les performances sont aussi bonnes, tant et aussi longtemps que le saut indirect ne cible pas les trappes intégrées dans la sonde Dyntrace. Le grand désavantage de cette méthode est qu'elle demande beaucoup plus de mémoire. En effet, l'adresse de l'instrumentation n'est pas totalement contrôlée. Par conséquent, il est possible qu'une page mémoire soit allouée pour chacune des sondes existantes.

2.3.3 LiteInst

ListInst est semblable à Dyntrace dans le sens que celui-ci utilise un saut de cinq octets, même si un saut indirect est présent dans la fonction. Lorsqu'une sonde doit remplacer



(a) instructions originales

(b) instructions modifiées

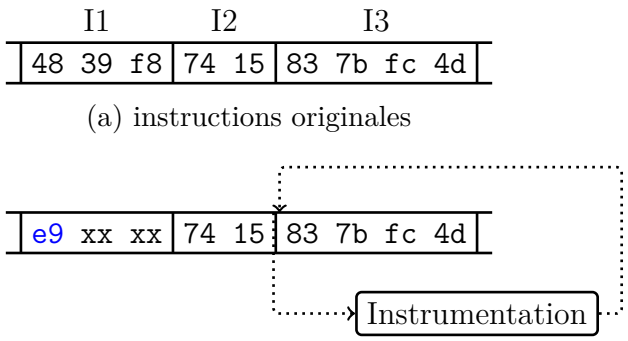
FIGURE 2.19 La figure 2.19a donne l'encodage de quatre instructions dont les deux premières sont de la taille d'un octet chacune. La figure 2.19b illustre comment la sonde serait insérée. La première instruction est remplacée par un code d'opération, en bleu, suivi de l'addresse relative de quatre octets. Le premier octet des deux instructions suivantes, en rouge, est remplacé par une trappe. Les deux octets suivants (`xx`) contiennent la portion de l'adresse cible que Dyntrace peut ajuster pour se rendre à l'instrumentation.

plusieurs instructions, et qu'un autre saut cible une des instructions suivant la première, ce qui est possible dans un saut indirect, seulement la première instruction est modifiée [7]. Par conséquent, la cible du saut est en partie fixée selon les instructions suivant la première. La figure 2.20 illustre une sonde insérée par LiteInst.

LiteInst souffre de problèmes semblables à Dyntrace. Puisque la cible de l'instrumentation n'est pas totalement contrôlée, dans le meilleur des cas, plusieurs sondes partagent quelques pages mémoire. Dans le pire cas, une page mémoire est allouée par sonde. Cette méthode peut être plus rapide dans le cas où d'autres sauts cibleraient les instructions suivant la sonde. Dans le cas de Dyntrace, une trappe serait exécutée. Ici, l'exécution normale continue.

2.3.4 E9Patch

L'outil E9Pach est un programme permettant de modifier un programme ou une bibliothèque avant son exécution. Pour ce faire, un programme lui est donné en entrée avec les sondes à insérer. Celui-ci génère à sa sortie le programme modifié. Celui-ci peut ensuite être exécuté. Pour une bibliothèque, il suffit d'utiliser la variable d'environnent `LD_PRELOAD` pour remplacer la bibliothèque originale par celle qui a été modifiée. Puisque l'instrumentation est effectuée avant l'exécution et non pendant, il n'est pas nécessaire de se préoccuper des problèmes liés



(a) instructions originales

(b) instructions modifiées

FIGURE 2.20 La figure 2.20a donne l'encodage de deux instructions, de trois et deux octets respectivement. La figure 2.20b illustre comment la sonde serait insérée. La première instruction est remplacée par le code d'opération, en bleu, suivi de deux octets (`xx`) contenant partiellement la cible. Les deux octets suivants de la cible sont contenus dans la seconde instruction qui n'est pas modifiée.

à la modification des instructions pendant l'exécution du programme.

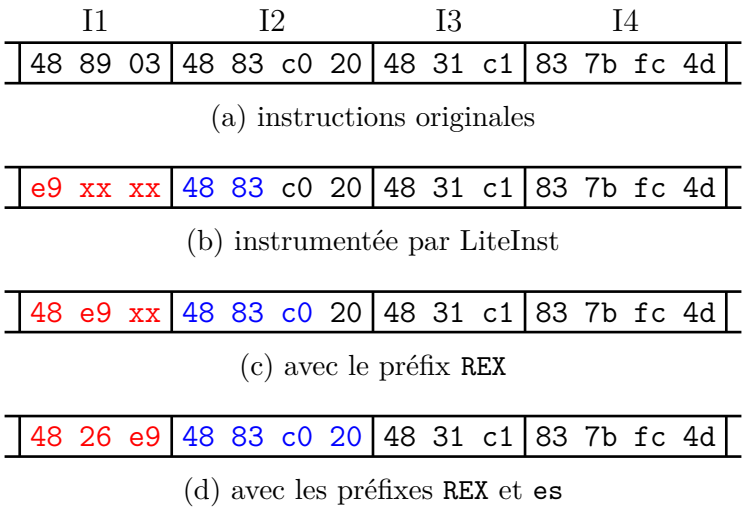
L'outil étend la technique de LiteInst afin d'obtenir une meilleure couverture d'instructions. Puisque la cible de l'instrumentation n'est pas totalement contrôlée, la sonde de LiteInst, et aussi Dyntrace, peut cibler une adresse invalide, comme une adresse nulle, négative ou déjà allouée. E9Patch définit trois méthodes différentes, toutes basées sur celle de LiteInst, afin d'insérer une sonde dans les cas où l'adresse de l'instrumentation serait contrainte à une adresse invalide [20]. Comme les deux autres outils, ces méthodes ne demandent pas de retrouver les sauts directs ou indirects.

Technique 1

La première technique est essentiellement comme ListInst, mais E9Patch permet l'ajout d'un préfixe à l'instruction de saut. On ajoute normalement un saut de cinq octets. Toutefois, il est possible d'y ajouter des préfixes, `REX` et `es`, afin d'étendre la longueur du saut à six et sept octets respectivement. La figure 2.21 illustre ces deux cas. Bien sûr, plus l'instruction est longue, moins l'outil peut contrôler les octets de la cible.

Technique 2

La seconde méthode consiste à éviter l'instruction suivante afin de pouvoir mieux contrôler la cible du saut. Pour ce faire, il commence par instrumenter, avec la première technique, l'instruction suivant celle qui est ciblée. Lorsque cette sonde est exécutée, l'instruction originale à



(a) instructions originales

(b) instrumentée par LiteInst

(c) avec le préfix `REX`

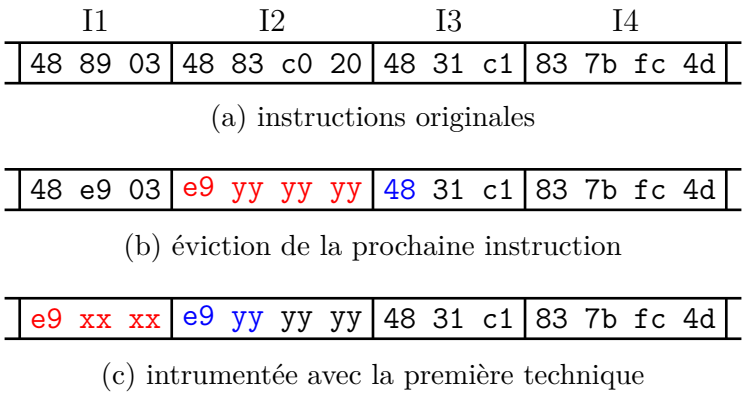
(d) avec les préfixes `REX` et `es`

FIGURE 2.21 La figure 2.21a montre les instructions originales avant l'instrumentation. La figure 2.21b illustre comment LiteInst instrumenterait ces instructions. La figure 2.21c ajoute à ce dernier le préfixe `REX` au saut, perdant ainsi le contrôle sur un octet de l'adresse du saut. La figure 2.21d illustre l'ajout du préfixe `es`. Ici, la cible du branchement est totalement contrainte par l'instruction suivante.

cette adresse est tout simplement simulée, ou exécutée hors-ligne. Par la suite, on instrumente l'instruction ciblée, encore avec la première méthode. L'avantage de cette technique est que la cible se contrôle aussi avec la seconde instruction. La figure 2.22 illustre cette technique.

Technique 3

Si les deux premières méthodes n'ont pas réussi à cibler une adresse valide, alors les auteurs de E9Patch décrivent une troisième technique qui fonctionne dans la majorité des cas. Au lieu d'éviter l'instruction suivante, ils se permettent d'éviter une instruction dans la portée d'un saut de un octet. Pour ce faire, un endroit pour le saut de cinq octets est premièrement déterminé. Ce saut peut être entièrement contraint ou non. Par la suite, un saut de un octet est inséré à l'adresse de la sonde. Ce saut cible le saut de cinq octets. Finalement, l'instruction qui a été modifiée est ensuite évitée par un autre saut de cinq octets. Son instrumentation simule l'instruction originale. La figure 2.23 illustre cette technique. Cette technique est très puissante, car il y a beaucoup de combinaisons différentes d'adresses possibles autour d'un saut de un octet.



(a) instructions originales

(b) éviction de la prochaine instruction

(c) intrumentée avec la première technique

FIGURE 2.22 La figure 2.22a montre les instructions originales avant l'instrumentation. La figure 2.22b illustre l'instrumentation de la seconde instruction. La figure 2.22c illustre l'instrumentation de l'instruction ciblée. Par conséquent, trois octets (`xx`, `xx` et `yy`) parmi ceux de la cible peuvent être ajustés pour cibler l'instrumentation, contrairement à deux si la première technique avait été utilisée.

2.4 Reconstruction du Graphe de Flot d'Exécution

Dans cette section, on présente premièrement les différentes sources de sauts indirects. On discute aussi d'un paradoxe lié à leur recouvrement. Par la suite, on explore différentes techniques pour découvrir les sauts indirects, utilisés dans des outils qui extraient le graphe de flot de contrôle d'un programme. Ce domaine de recherche est surtout motivé par les outils de rétro-ingénierie, pour analyser un programme malicieux par exemple, ou par certains outils de tranduction dynamique, comme Valgrind.

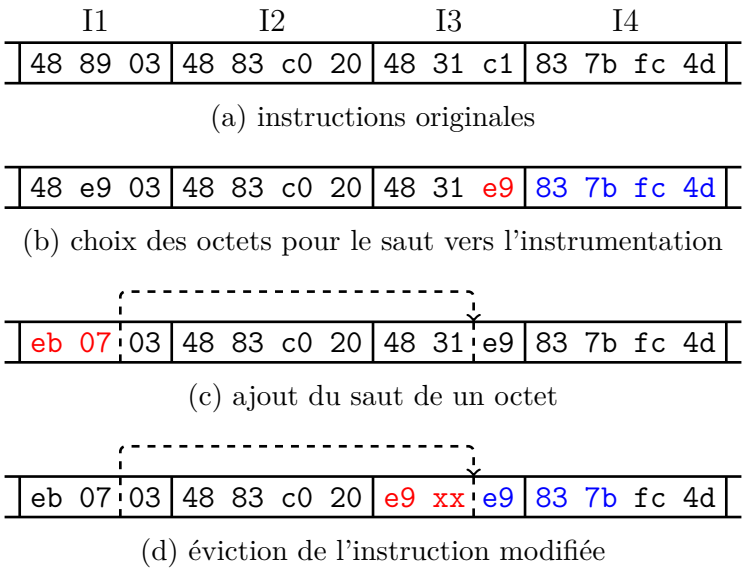
2.4.1 Utilisation de Sauts Indirects

Il existe principalement 4 raisons, ou sources, qui expliquent l'utilisation de sauts indirects dans un programme.

Cible Reconstruite

Les architectures RISC comme RISC-V ou ARM, 32 ou 64 bits, utilisent principalement des instructions à longeur fixe de 32 bits. Dans l'encodage de ces instructions, une partie est réservée au code d'opération. Par conséquent, les architectures ne supportent pas les sauts directs et absolus de 4 octets ou de 8 octets. Pour effectuer ces types de sauts, il faut reconstruire la cible dans un registre et ensuite effectuer un saut indirect.

Ceci est aussi le cas pour l'architecture x86, même si celle-ci a une taille d'instruction variable.



(a) instructions originales

(b) choix des octets pour le saut vers l'instrumentation

(c) ajout du saut de un octet

(d) éviction de l'instruction modifiée

FIGURE 2.23 La figure 2.23a montre les instructions originales avant l'instrumentation. Dans la figure 2.23b, une adresse contenant le saut de cinq octets est déterminée. Sa cible est entièrement contrainte. La figure 2.23c illustre l'ajout du saut de un octet. Finalement, la figure 2.23d illustre l'instruction évitée. Celle-ci est nécessaire parce qu'elle a été modifiée à la figure 2.23b et il est nécessaire de conserver le comportement original du programme.

Elle n'offre pas de branchement absolu vers une adresse immédiate. Il faut copier l'adresse vers un registre et ensuite l'utiliser. Par contre, ceci est rarement émis par les compilateurs, car ceux-ci peuvent généralement résoudre les symboles de manière relative. La figure 2.24 illustre un exemple de ces branchements indirects.

```
void (*f)(void);
f = 0x11223344;
f();
```

(a) Code C

```
li   a5,0x11223000
addi a5,a5,0x344
jalr a5
```

(b) RISC-V

```
movw r3, #0x3344
movt r3, 0x1122
blx  r3
```

(c) ARM

```
movl $0x11223344, %eax
call  *%rax
```

(d) x86

FIGURE 2.24 La figure 2.24a donne un appel absolu en C. Les figures 2.24b, 2.24c et 2.24d donnent les instructions émises pour un programme RISC-V, ARM et x86-64 respectivement. Chaque branchement est indirect. L'adresse `0x11223344` est copiée dans un registre avant le branchement. Chaque appel pourrait être optimisé en un saut indirect s'il est une fin d'appel.

Branchement à N Conditions

Une des sources de sauts indirects provient d'un branchement à N conditions. Dans un langage comme le C, ceux-ci proviennent d'une série de `if-else-if` ou d'une déclaration `switch-case`. Dans les deux cas, l'utilisation d'un saut indirect est une optimisation que le compilateur effectue si nécessaire. Au lieu de vérifier N conditions différentes, celui-ci peut directement calculer l'adresse liée à la condition et y sauter. Par conséquent, il ne doit en général que vérifier la condition liée aux bornes.

Pour ce faire, une série de conditions doit être respectée. Dans le cas d'un `if-else-if`, chaque condition doit dépendre de la même variable. Dans un `switch-case`, ceci est déjà le cas. De plus, les valeurs des conditions liées à cette variable ne doivent pas être trop dispersées, sinon le compilateur risque de générer une recherche binaire ou même une table de hachage. Dans le cas idéal, l'ensemble des valeurs utilisé est contigu. Si certaines valeurs sont contigües et d'autres dispersées, le compilateur peut générer un mélange entre une table de correspondance, une recherche binaire et une table de hachage. La figure 2.25 illustre deux exemples de conditions pouvant être optimisées. De plus, la figure 2.26 donne l'assembleur qui serait généré pour les deux exemples.

```
if (x == 0) { ... }
else if (x == 1) { ... }
else if (x == 2) { ... }
else if (x == 3) { ... }
else if (x == 4) { ... }
else if (x == 5) { ... }
else { ... }
```

(a) `if-else-if`

```
switch (x) {
case 0: ...
case 1: ...
case 2: ...
case 3: ...
case 4: ...
case 5: ...
default: ...
}
```

(b) `switch-case`

FIGURE 2.25 Deux exemples de code C pouvant être optimisés en saut indirect.

Optimisation de Fin d'Appel

Une autre source de sauts indirects provient des appels de fonctions optimisés à la fin d'une fonction. Lorsqu'une fonction en C ou C++ est appelée à la fin d'une autre fonction, le compilateur peut faire le choix de réutiliser l'espace sur la pile pour la fonction enfant. Cette optimisation permet d'épargner un peu de mémoire à chaque appel et est souvent idéale

```
; vérification des bornes
cmpl $5, %edi
ja X_AUTRE

; début de la table de sauts
leaq TABLE_SAUTS(%rip), %rdx

; copie de la variable x
movl %edi, %eax

; décalage dans la table
movslq (%rdx,%rax,4), %rax
addq %rdx, %rax

; saut indirect
jmp *%rax
```

(a) section `.text`

```
  .align 4
  .align 4
.TABLE_SAUTS:
  .long .X_AUTRE-.TABLE_SAUTS
  .long .X_EQ_0 -.TABLE_SAUTS
  .long .X_EQ_1 -.TABLE_SAUTS
  .long .X_EQ_2 -.TABLE_SAUTS
  .long .X_EQ_3 -.TABLE_SAUTS
  .long .X_EQ_4 -.TABLE_SAUTS
  .long .X_EQ_5 -.TABLE_SAUTS
```

(b) section `.rodata`

FIGURE 2.26 La figure 2.26a illustre un exemple de saut indirect généré par le compilateur GCC pour un exécutable à position indépendante. Celui-ci effectue une vérification de la borne pour déterminer le cas par défaut. Ensuite, il calcule l'adresse dans la table de sauts contenant la valeur du saut indirect relatif. La figure 2.26b contient la définition de la table de correspondance dans la section de données en lecture seule.

pour les fonctions récursives, car celles-ci peuvent manquer d'espace sur la pile après un trop grand nombre d'appels [45]. Dans ce contexte, le compilateur remplace tout simplement l'instruction d'appel, comme un `call`, par un branchement relatif, comme un `jump`. Après l'exécution de la fonction enfant, le contrôle retourne à la fonction grand-parent au lieu de la fonction parent.

Lorsque l'appel est indirect, le compilateur effectue l'optimisation à l'aide d'un saut indirect. En C, la seule méthode pour effectuer un appel indirect à une fonction est d'utiliser un pointeur de fonction. En C++, les méthodes virtuelles utilisent aussi des appels indirects. Par conséquent, ces deux cas peuvent générer des sauts indirects s'ils sont placés à la fin d'une fonction. La figure 2.27 illustre un exemple en C et C++ pour ces deux cas.

Saut Manuellement Écrit

Finalement, il existe d'autres cas spécifiques pour lesquels un saut indirect peut être utilisé. Par exemple, il est possible que le programmeur écrive une routine directement en assembleur

```c
typedef void (*func_t)();

void test(func_t call_me) {
    /* appel indirect */
    call_me();

    /* saut indirect */
    call_me();
}
```

(a) appels indirects en C

```cpp
class exemple {
public:
  virtual void call_me();
};

void test(exemple& ex) {
    /* appel indirect */
    ex.call_me();

    /* saut indirect */
    ex.call_me();
}
```

(b) appels indirects en C++

FIGURE 2.27 Deux exemples d'appels et sauts indirects en C et C++. Dans les deux cas, la première fonction appelée est effectuée avec un appel normal, car elle ne se situe pas à la fin de la fonction. La deuxième fonction appelée est optimisée avec un saut indirect, car elle se situe à la fin.

qui utilise un saut indirect. Ou encore, il existe des programmes se modifiant eux-mêmes, comme des logiciels malicieux ou certaines machines virtuelles. Ceux-ci sont beaucoup plus difficiles à analyser, car ils peuvent utiliser les sauts indirects d'une manière non standard ou inconnue.

2.4.2 Dépendance Circulaire

Pour reconstruire un graphe de flot de contrôle, il faut déterminer les cibles de chaque branchement dans les instructions analysées. Pour les sauts directs, il suffit d'analyser l'opérande de l'instruction. Pour les sauts indirects, il faut les déterminer avec des analyses plus complexes. Dans tous les cas, les méthodes pour retrouver les cibles demandent d'analyser les instructions menant à son exécution. Il peut y avoir plusieurs chemins menant au saut indirect. Pour obtenir les chemins possibles, il faut premièrement reconstruire le GFC. Par conséquent, on arrive à un paradoxe. La reconstruction du GFC dépend de l'extraction des sauts indirects, mais ceux-ci dépendent de la reconstruction du GFC. Pour résoudre ce paradoxe, les techniques assument habituellement que les chemins menant au saut indirect ne proviennent pas du saut indirect lui-même, ni d'un autre.

2.4.3 Traqueur d'Expression Simple

Cette technique se concentre seulement à retrouver un saut indirect provenant d'une cible reconstruite. Pour ce faire, elle rebobine les instructions effectuant le calcul de la cible [19].

L'algorithme est récursif et prend en paramètre une instruction et une valeur non constante, comme un registre ou un emplacement de mémoire. Si la valeur non constante est un registre, alors il détermine la dernière instruction modifiant ce dernier et appelle récursivement l'algorithme. Si la valeur variable provient d'une lecture mémoire à partir d'une adresse dans un registre, alors il détermine la dernière instruction modifiant ce registre et appelle l'algorithme récursivement. Si l'instruction ne contient que des constantes, alors le résultat de l'appel peut être calculé et retourné directement. Le résultat est ensuite propagé et modifié à travers chaque appel afin de former la cible finale du saut indirect.

Cette méthode est simple à implémenter, mais ne supporte que les sauts indirects ayant une seule cible, comme ceux provenant d'une cible reconstruite. Ceci est partiellement dû au fait que son critère d'arrêt à la récursion est lorsqu'il n'y a plus de valeur variable. L'algorithme échoue aussi lorsqu'il rencontre des branchements durant son rebobinage, car ceci donnerait la possibilité d'avoir différentes cibles possible. Finalement, il échoue aussi lorsqu'une instruction a plus d'une valeur variable, comme une addition de deux registres. Cela dit, cette méthode est très efficace.

2.4.4 Traqueur d'Expression Bornée

Cette technique est une amélioration de la technique précédente. Au lieu de permettre à chaque instruction i de modifier une seule valeur, chaque instruction peut modifier un ensemble V_i de valeurs, tel que décrit par l'équation

$$V_i = \left\{ a_i + b_i \cdot x \,\middle|\, x \in \mathbb{Z}, \begin{array}{c} c_i \leq x \leq d_i, \text{ si l'ensemble est successif} \\ x < c_i \leq d_i < x, \text{ si l'ensemble est une négation} \end{array} \right\},$$

où a_i, b_i, c_i et d_i sont des constantes [19]. La variable x est la valeur bornée. Par conséquent, les valeurs peuvent former un ensemble linéaire de valeurs ou le contraire de celles-ci. Au début de l'algorithme, les cibles sont décrites par un registre ou un emplacement de mémoire aucunement borné. Par conséquent, $a_0 = 0$, $b_0 = 1$ et x n'est pas bornée par c_i ou d_i. Cet ensemble est ensuite restreint en fonction de la dernière instruction modifiant cette variable.

Par exemple, si la dernière instruction additionne une constante k, alors

$$V_{i+1} = \left\{ (a_i - k) + b_i \cdot x \;\middle|\; x \in \mathbb{Z}, \begin{array}{c} c_i - k \leq x \leq d_i - k, \text{ si l'ensemble est successif} \\ x < c_i - k \leq d_i - k < x, \text{ si l'ensemble est une négation} \end{array} \right\}.$$

Les branches conditionnelles vont aussi restreindre les constantes c_i et d_i. Les opérations sur le signe de la valeur x peuvent aussi la borner. Par exemple, si une valeur est détectée comme non signée, alors $c_i = 0$. Pour les opérateurs binaires, comme le *et* ou le *ou*, leur résultat ne peut pas être représenté par un ensemble de la forme V_i. Pour supporter ces cas, l'ensemble des valeurs à une instruction i peut être le résultat d'une opération sur deux ensembles, comme une union ou une intersection. L'algorithme s'arrête lorsque x est borné.

Cette méthode fonctionne généralement bien pour résoudre les sauts indirects provenant d'un branchement à N conditions. Selon les tests expérimentaux, de 89.56% à 95.3% des cibles furent recouvertes pour différents programmes et architectures. Cependant, cette méthode est incapable de résoudre les sauts indirects provenant des optimisations de fin d'appel. Ceci est dû au fait que la cible provient habituellement d'un paramètre à la fonction, comme un objet, et qu'aucune instruction ne borne la cible du saut indirect.

2.4.5 Correspondance de Patrons

Une des techniques les plus utilisées pour retrouver un saut indirect est la correspondance de patrons. Dans celle-ci, on analyse premièrement les types de patrons que les différents compilateurs peuvent émettre lors de l'utilisation d'un saut indirect. Par la suite, on tente de déterminer quel patron est utilisé par un saut indirect.

Une implémentation connue dans le domaine est celle de l'outil UQBT. Cet outil de traduction dynamique permet de convertir un programme d'une architecture à une autre. Celle-ci a besoin du GFC pour traduire les différents blocs linéaires à travers le programme. Sa méthode recouvre seulement les sauts indirects provenant des branchements à N conditions [8]. Celle-ci prend en entrée un saut indirect et commence par extraire les instructions dépendantes à celui-ci. Pour ce faire, elle analyse le registre utilisé dans le saut indirect et obtient la ou les dernières instructions le modifiant. Ces instructions sont ensuite analysées à leur tour pour déterminer les instructions modifiant leurs paramètres et ainsi de suite. Le critère d'arrêt d'un chemin est lorsque les registres analysés sont bornés ou qu'ils proviennent de la mémoire, d'une fonction appelée ou des arguments de la fonction actuelle.

Pour chaque chemin de dépendances, une représentation intermédiaire, sous la forme de notation de transfert de registres, est générée. Ce chemin est ensuite simplifié en remplaçant

les expressions dépendantes dans une instruction par leurs dépendances définies dans d'autres instructions. Le résultat de cette opération est une description du saut indirect sous une forme très compacte. Cette forme compacte est comparée à trois types de formes connues, données dans la figure 2.28. Si une forme y correspond, il est ensuite facile de calculer les cibles du saut indirect.

```
jcond (r[v] > num_u) X
jmp [T + <expr> × w]
```

(a) forme A

```
jcond (r[v] > num_u) X
jmp [T + <expr> × w] + T ou
jmp PC + [PC + (<expr> × w + k)] ou
jmp PC + [PC + (<expr> × w + k)] + k
```

(b) forme O

```
jcond (r[v] > num_u) X
jmp [T + ((<expr> & mask ) × 2 × w) + w]
```

(c) forme H

FIGURE 2.28 Différentes formes, ou patrons, supportés par UQBT.

Cette méthode a été implémentée et testée sur des machines x86 et SPARC. Elle semble bien retrouver les sauts indirects provenant d'un branchement à N conditions. De plus, il est relativement facile d'ajouter une architecture, car les formes normales sont agnostiques à celles-ci. Par contre, ils ne spécifient pas et ne vérifient aucunement la qualité des sauts découverts. Par exemple, il est possible que l'ensemble des cibles liées à un saut indirect soit sous-approximé, sur-approximé ou approximé exactement.

2.4.6 Analyse d'Ensemble de Valeurs

La technique du *Value Set Analysis* (VSA) [2] est utilisée afin de récupérer les valeurs des différentes variables à travers le programme. Par le fait même, elle est en mesure de retrouver les sauts indirects afin d'obtenir un graphe de flot de contrôle complet d'un programme. Cette méthode est utilisée dans le but d'analyser des programmes malicieux où l'information de débogage et le code source ne sont pas disponibles. Par conséquent, son analyse peut être utilisée pour récupérer les sauts indirects, mais elle offre beaucoup plus d'informations.

La technique utilise IDA Pro afin de désassembler le programme. Celui-ci est en mesure de détecter les différentes fonctions dans le programme (ce qui est normalement offert par l'information de débogage), les adresses de différentes variables statiques et encore plus [31].

L'analyse fonctionne au niveau du programme en entier ou au niveau de chaque fonction. Un graphe de flot de contrôle est créé à partir des sauts directs. Dans ce graphe, chaque noeud est une seule instruction. Pour décrire l'ensemble de valeurs possibles d'une variable à chaque noeud (instruction), la technique utilise un *stockage abstrait*. Cette structure de donnée spécialisée est en mesure de décrire une grande variété d'ensembles de valeurs. L'algorithme explore ensuite le GFC. À chaque arête, le stockage abstrait est modifié selon la

dernière instruction. L'algorithme décrit donc un *transformateur* pour chacune des instructions possibles. Plus l'algorithme avance, plus l'ensemble des valeurs du stockage abstrait est restreint. Une fois l'ensemble des valeurs étant bornées dans le programme, le GFC est modifié pour ajouter les cibles provenant des stockages abstraits des sauts indirects.

Les auteurs ne spécifient pas les performances concernant le taux de recouvrement des sauts indirects. Cependant, cette méthode devrait être en mesure de bien retrouver les sauts indirects provenant des tables de sauts. Par contre, ils spécifient que les sauts indirects sont souvent sur approximés par un grand facteur. De plus, cette méthode demande beaucoup de mémoire et de temps d'exécution.

2.4.7 Discussion

Dans ce chapitre, nous avons exploré une grande variété d'outils de traçage et d'instrumentation. Chaque outil à ses forces et ses faiblesses selon la manière dont il est adopté par l'utilisateur. Par conséquent, il faut bien choisir l'outil en fonction de ses requis. Par exemple, de la simple journalisation peut suffire si l'on teste des prototypes rapidement. Pour des sondes statiques de haute performance, LTTng est un bon choix. Si l'on souhaite étudier les appels de fonctions, un outil comme Ftrace ou Uftrace avec des sondes statiques insérées par le compilateur peut être idéal. Lorsque des requis de traçage dynamique sont demandés, il est idéal d'utiliser un logiciel se basant sur les sondes dynamiques comme Kprobe, Uprobe, etc.

La technique basée sur l'insertion d'un branchement, comme Kprobe, est souvent considérée comme l'état de l'art. Cela dit, son taux de succès est incomplet à cause, en grande partie, de la présence de sauts indirects. Les techniques comme NOProbe, LiteInst, DynTrace et E9Patch évitent le problème de recouvrir ces sauts indirects de différentes manières. Dans ce mémoire, on caractérise et s'attaque directement à ce problème en tentant d'appliquer une méthode provenant d'un autre domaine pour recouvrir ses sauts indirects.

CHAPITRE 3 ARTICLE 1 : x86 Indirect Jump Recovery Using Pattern Matching For Tracing Applications

Authors

Gabriel-Andrew Pollo-Guilbert

École Polytechnique de Montréal

gabriel.pollo-guilbert@polymtl.ca

Michel Dagenais

École Polytechnique de Montréal

michel.dagenais@polymtl.ca

Keywords : Indirect Jump Recovery, Tracing, Dynamic Instrumentation, Binary Analysis, Control-Flow Reconstruction.

Submitted to : Software : Practice and Experience, December 11 2021.

Abstract

Dynamic instrumentation is a method to insert tracepoints in a running application. The easiest approach, based on single byte trap instructions, always works but incurs significant overhead. Modern approaches, such as the Linux optimized Kprobe, try to optimize tracepoints by replacing them with 5-bytes jumps. State-of-the-art techniques can currently optimize about 90% of tracepoints. Among the 10% non-optimized tracepoints, a large proportion target shorter than 5 bytes instructions, inside functions containing indirect jumps. Since the indirect jumps targets are not easily known, this prevents replacing multiple small instructions with 5 bytes jumps. Resolving indirect jumps has been achieved in advanced reverse engineering frameworks. Yet, it was always avoided in tracing, because of its complexity. In this paper, we present a new, efficient and reliable method, based on pattern matching, for recovering jump table indirect jumps. We show that our method can reduce the number of non-optimized tracepoints by 20% to 40%, depending on the target program. We also present a few ideas for recovering other types of indirect jumps.

3.1 Introduction

When debugging a problem in a software module, information about the program execution is extremely useful. Software tracing is often used as an efficient way to extract this data. A common way to achieve this is to instrument the application at the source code level, using static tracepoints. It is typically the most efficient, flexible and simple method. However, it requires knowing in advance the relevant locations to instrument. If a tracepoint is missing at runtime, the developer needs to recompile and restart the application with the new tracepoint.

In some scenarios, such as production environments, inserting a new tracepoint and restarting the application is time consuming or downright not feasible. Dynamic instrumentation is a solution to this problem. The developer can insert a tracepoint while the application is running, without even restarting it. This technique is usually much more complicated than its static counterpart, but it can help fill the gap in situations where recompiling or restarting the application is not a viable option.

Dynamic instrumentation needs to modify the instructions of a program, often at runtime, in order to gather data about its execution. Under the x86 instruction set [34], dynamic instrumentation methods, such as the Linux Kernel Kprobe, replace the first byte of the instruction at the target address with a trap instruction. The associated interrupt executes a user-provided handler [42]. While inserting a trap always works, it adds a considerable overhead for each probe, because of the context switches generated. In userspace, this is even worse, because of additional context switches, not only to interrupt mode but between kernel and userspace modes.

When possible, the tracepoint can be optimized by inserting a 5 bytes jump instead of a trap. For this to work reliably, several conditions must be met. Replacing an instruction of 5 bytes or more is always safe, but many instructions are fewer than 5 bytes. In those cases, multiple instructions need to be overwritten. This leads to two problems. First, if multiple instructions are replaced, then it is possible that another thread is currently executing the second or subsequent instructions within the 5 bytes. This may be solved by either waiting for the thread to advance, or moving the thread to equivalent instructions elsewhere. Secondly, it is possible that a jump from elsewhere lands in the middle of our 5 bytes jump, expecting it to be one of the instructions that was in fact replaced, as shown in Figure 3.2. Figure 3.1 shows a safe jump target. This is typically avoided first analyzing the program to insure that no such case exists. Failing that, the tracepoint is not optimized.

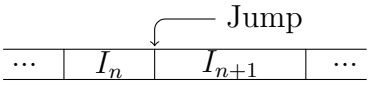


FIGURE 3.1 Jump target safely landing at the end of a instruction I_n and the start of I_{n+1}.

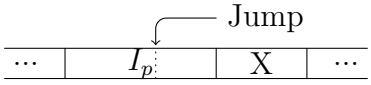


FIGURE 3.2 Jump target landing in the middle of the patched instruction I_p written over I_n and I_{n+1}. If executed, it may lead to a program crash.

It is generally assumed that functions will not jump in the middle of other functions. Therefore, only the present function needs to be analyzed for jump targets. It is easy to check for jump targets coming from direct jumps, since they can be extracted by only looking at the immediate value stored in the opcode. This is not the case for indirect jumps, however. These jumps often depend on runtime values such as an index or a pointer.

When an indirect jump is found in the current function, the tracepoint is usually not optimized, and the trap instruction used, since we cannot ensure that the execution is safe when replacing multiple instructions with a 5 bytes jump. Resolving the target of indirect jumps statically is a complex task that has seen significant research in reverse engineering applications. This is problematic, since higher optimization levels, typically used in production environments, tend to generate many indirect jumps.

In this paper, we present an efficient method, based on pattern matching, to resolve indirect jump targets for userspace tracing applications. The rest of this paper is organized as follows. In Section 2, we present existing methods used in control-flow graph reconstruction and reverse engineering. Section 3 details our new algorithm and Section 4 presents experimental results. Finally, Section 5 contains a summary and discusses possible improvements.

3.2 Background and Related Work

In this section we first look at the different sources of indirect jumps, and then review the main methods proposed in the literature to statically resolve indirect jump targets.

3.2.1 Indirect Jump Sources

Materialized Destination Address

Common instruction set architectures usually offer multiple kinds of jump or call instructions to change the control flow of a program. In many cases, the program needs to go to a near address and the target can be encoded as an immediate value in the instruction. The x86-64

architecture provides three sizes of relative jumps : one byte offset, two bytes offset and four bytes offset. If a program needs to branch at an offset further than what is accessible by a four bytes offset, it needs to do an absolute jump. The target is stored in a register and the jump is indirect. Each time the branch is executed, the target must first be *materialized* in a register.

Nonetheless, these are quite rare in x86-64 programs. Most targets can be encoded in a relative four bytes, or less, jump. The compiler usually does not generate indirect jumps in those cases, because programs are typically much smaller than 2 Go. Even dynamic library calls, which may lead to sparse memory pages, use 4-bytes immediate jumps. However, they are much more common in RISC programs. The branch address is always smaller than the instruction size, since some of the bits are reserved for the instruction opcode or modifiers. RISC processors, with fixed length instructions, are thus more likely to use materialized destination addresses. For example, the *jump-and-link* (`JAL`) instruction of 32-bit RISC-V processors can only target addresses within 20 bits, allowing relative jumps to ±512 Ko [56]. Function calls to dynamic libraries use indirect jumps, because the libraries are loaded in different memory pages of size 4 Ko.

N-conditional Branches

The compiler may generate indirect jumps from N-conditional branches, such as a series of `if-else-if` or `switch-case` statements in C, as shown in Figure 3.3 and Figure 3.4. The values for the cases involved need to be consecutive numbers. If the cases are too sparse, the compiler will generate a binary search tree. If there are not enough cases, the compiler will simply generate consecutive compare and jump instructions. In some cases, it may generate a mix of the 3.

```
switch (n) {
case 0: ...; break;
case 1: ...; break;
case 2: ...; break;
case 3: ...; break;
case 4: ...; break;
case 5: ...; break;
default: ...; break;
}
```

FIGURE 3.3 N-conditional statements from multiple consecutive *switch-case* constructs.

```
if (n == 0) { ... }
else if (n == 1) { ... }
else if (n == 2) { ... }
else if (n == 3) { ... }
else if (n == 4) { ... }
else if (n == 5) { ... }
else { ... }
}
```

FIGURE 3.4 N-conditional statements from multiple *if-else* constructs.

Tail-Optimised Indirect Call

A common compiler optimization is reusing the current stack frame for the next function call. This is possible when the callee is called at the end of the caller. The compiler can generate a jump instruction, rather than a call instruction.

An indirect function call is a method for calling a procedure without knowing at compile time the target function. In C, it is usually achieved through a function pointer, while in C++ it can be generated by a virtual function call. In both cases, the function called depends on the state of the application.

With a direct function call, tail-optimization will generate a direct jump. For a tail indirect call, the optimization will generate an indirect jump. Pattern matching for these jumps does not work well. Hence, they are not handled in the scope of this paper. Nonetheless, we suggest interesting alternatives at the end of this paper.

Hand-Written Assembly

In some cases, a function may have been written by hand in assembly. There are multiple use cases where a developer would do that. For example, it may be more efficient or it might be to counter reverse engineering. There are also cases of self-modifying codes that rely on indirect jumps. In any case, the executed instructions may be something entirely different than what a compiler would generate. We focus on production applications where it is assumed that

minimum to no assembly is hand-written. Hand-written indirect jumps can be the hardest to recover, and are not handled in the scope of this paper.

3.2.2 Current Methods to Resolve Indirect Jumps

Recovering indirect jumps has been an area of research for quite some time in binary analysis and reverse engineering [19]. When analyzing a binary, recovering indirect jumps is necessary to rebuild the control-flow graph (CFG) properly. In binary translation, it is crucial that the rebuilt CFG be as correct as possible. In this context, indirect jump recovery methods can be quite complex.

Circular Dependency of Indirect Jumps Recovery

The recovery of indirect jump targets depends on the instructions that preceded the jump. Different lists of instructions may precede the indirect jump, depending on which code path was taken. Thus, all code paths must be analyzed when recovering indirect jumps. This also includes the code paths from the indirect jump itself. Therefore, proper recovery of an indirect jump requires that the indirect jump be already recovered.

This paradoxical situation prevents an indirect jump from being recovered with 100% certainty. In most cases, current methods ignore code paths from indirect jumps when recovering them. It is then resolved by reapplying the method a second time, after recovering the indirect jump. Nevertheless, we still cannot be certain that the recovered targets were all found or are valid.

Simple Expression Trackers

The Simple Expression Tracker [19] aims at recovering indirect jumps coming from materialized targets. It starts with the register of an indirect jump. It finds the last instruction that wrote into the register. If this instruction writes a constant value, then the value is known. If this instruction writes a value that depends on another register, the algorithm is executed again, but for the current instruction. The algorithm does not support an instruction with 2 or more dependencies on other registers. It can be implemented recursively or with a loop and a stack. The idea is that the moment that a constant is encountered, then the stack is unwound and the results are propagated to, eventually, the indirect jump register.

This method is simple and works well for materialized jumps. However, it does not work for anything else. Since the algorithm stops at a constant value, it is impossible to recover a jump table, because they usually contain multiple values.

Bounded Expression Trackers

This method is similar to the last. The simple method describes the indirect jump with a single value. Hence, it follows the dependencies until a constant is found. The Bounded Expression Tracker describes the indirect jump target with a specialized set of values [19]. At the start of the algorithm, the set is totally unbounded and can describe any value. When a dependency is processed, the set is restrained based on which mathematical operation was done. The algorithm continues following the dependencies until the set is finally bounded.

This method works well on indirect jumps that come from N-conditional statements. Based on their tests, they are able to recover 89.56% to 95.30% of indirect jump targets for different architectures [19]. However, they do not specify which kind of indirect jumps they were able to recover. They also do not recover targets from tail-optimized calls.

Slicing and Expression Substitution

A powerful method, described in [8], uses a two-step process to recover jump tables.

First, they use instruction slicing to extract all instruction dependencies of the indirect jump. The stop criterion, for determining when to stop backtracking instructions, is when a register is read from memory, returned by another function or received as an argument. Figure 3.5 shows an example of instructions slicing, using an intermediate representation.

```
eax = m[ebp-8]
eax = eax - 2
ZF = (eax - 5) = 0  1 : 0
CF = (~eax@31 & 5@31) |
     ((eax-5)@31 & (~eax@31 | 5@31))
PC = (~CF & ~ZF) = 1  0x80489dc : PC
PC = m[0x8048a0c + eax * 4]
```

FIGURE 3.5 Sample output of instructions slicing. The indirect jump, at the end, has been backtracked and all dependencies were found. The instruction slicing output can also be architecture independent.

Instructions slicing requires the control-flow graph (CFG) in order to work properly. Hence, it needs to resolve the circular dependency between the CFG and the indirect jumps. Surprisingly, they do not mention anything related to this problem.

The subset of instructions is still too complex to analyze. By means of forward substitution, the slice is converted into a much simpler representation. For example, Figure 3.6 is the

simplified expression of Figure 3.5. The simplified expression is then compared with a known *normal form*, Figure 3.7 and Figure 3.8 show 2 of the 3 normal forms specified in [8].

```
jcond ([ebp-8] > 7) 0x80489dc
jmp [0x8048a0c + ([ebp-8] - 2) * 4]
```

FIGURE 3.6 Sample output of the expression substitution phase.

$$\begin{aligned} &\text{jcond } (E > K_1) \cdot X \\ &\text{jmp } [K_2 + E \cdot K_3] \end{aligned}$$

FIGURE 3.7 Type A normal form.

$$\begin{aligned} &\text{jcond } (E > K_1) \cdot X \\ &\text{jmp } [K_2 + E \cdot K_3] + T \quad \text{or} \\ &\text{jmp PC} + [\text{PC} + (K_2 + E \cdot K_3)] \end{aligned}$$

FIGURE 3.8 Type O normal form.

The advantage of that method is that very few normal forms, or patterns, are needed to describe a wide variety of indirect jump patterns. It can also be easily adapted to other architectures. Still, it does pose some implementation challenges in terms of complexity. The expression slicing process requires each opcode of the ISA to be interpreted to know which register/memory it reads or writes. Each opcode also needs to be converted into an equivalent intermediate representation. However, this method is able to recover a good percentage of indirect jumps. Additional patterns would be needed to support position-independent code.

Value-Set Analysis

A more complex method, described in [2], tries to approximate a safe set of values for all data locations in the program, be it register or memory. By recovering all of those approximations, it eventually recovers any indirect jump. It supports both static and dynamic analysis. The latter is usually preferred in order to recover a more precise set of values.

It starts by creating a rudimentary CFG, based only on direct jumps/calls inside a single function (intraprocedural analysis). Each node in the graph is one instruction. For conditional statements, there may be multiple successor nodes.

The algorithm uses abstract stores, which are a kind of advanced mathematical set, for describing the possible values of a register or address at a specific instruction. At each edge

of the CFG, a transformer is applied on an abstract store in order to constrain further the possible values at the next instruction. Different kinds of instructions require specific transformers to be defined. By going through the entire CFG, from the entry point, it slowly restricts the values of certain registers or memory locations at each instruction.

The method goes further by conducting an intraprocedural analysis (between functions). This additional step is not described here, because it is not relevant to the work presented in this paper.

When it encounters an indirect jump, it estimates the target addresses based on the current abstract store. For indirect calls, it also assumes that it will not jump into the middle of the current function. This way, they try to circumvent the circular dependency between the CFG recovery and indirect jumps/calls.

In most cases, this method works well for recovering indirect jumps. In the worst case, it will overestimate target addresses by a large factor. Since its primary goal is to recover memory accesses for all instructions, the technique is complex. It needs a transformer for all kinds of instructions, and large number of abstract stores will require a large amount of memory.

Tailored Value-Set Analysis

This method, described by [9], is a more generic version of the previous value-set analysis. They first start by lifting the machine instructions, into an LLVM intermediate representation, and applying a variety of optimizations. This allows the method to be architecture independent.

Starting from an indirect jump, they find all possible code paths that end on the instruction. The start of a path is found when a memory read occurs, or when the start of the function is encountered. If the path depends on another indirect jump, then the path is traced recursively into it.

All paths that end on an indirect jump create a direct acyclic graph (DAG). Each node corresponds to a memory pointer access in the Satisfiability Modulo Theories (SMT) form. An edge corresponds to a data dependency between two nodes. Each path in the graph can be solved using an SMT solver, in their case Z3 [15].

At first, this method yields very high false positive rates. But by optimizing how they solve the SMT expressions, they are able to achieve a very good jump table recovery rate. In some cases, they achieved a recovery rate of up to 97%. However, they do not mention if the recovered tables contain overapproximations or if they are all exact. The main drawback of this approach is that it requires a large amount of memory, and it is slower than the other

methods, taking multiple seconds per binary.

3.3 Algorithm Description

Most of the existing work, described in the last section, is used in binary analysis (reverse engineering tools) or binary translation. In those fields, it is generally assumed that the target binary does not contain debugging information (i.e., a proprietary binary, possibly containing malware). In that case, indirect jump/call recovery is mainly used to recover the entire CFG of the program, with precision being much more important than computational cost. In the case of binary translation, an incorrect CFG might produce an invalid binary after the translation. Those constraints impose that recovering indirect jumps/calls be as precise as possible.

Fortunately, tracing does not bear the same constraints. Tracing tools are often used by engineers that want to extract information about their own systems, in which case they should have access to debugging information. They may even recompile the program with fewer optimizations (i.e., while testing in a non-production environment), since compiler optimizations can make the recovery even harder. Unlike for CFG recovery, our method does not need to recover indirect calls, since we assume that calls do not land in the middle of a function.

Recovering indirect jumps, for dynamic instrumentation optimization, further reduces the constraints. Failing to recover the target of an indirect jump is not catastrophic, it only means that dynamic tracepoints will not be optimized in the function (i.e., a single byte trap instruction will be used). Similarly, it is not too problematic if the indirect jump recovery algorithm over-approximates the targets, it simply means that tracepoints are less likely to be optimized. Hence, we suggest a flexible and computationally efficient algorithm to recover indirect jumps, based on pattern matching.

3.3.1 Pattern Matching Tree

Regular expressions are a powerful and efficient representation for patterns matching a string. Algorithms, such as the Glushkov [26] or Thompson constructions [58], are able to convert a regular expression into a non-deterministic finite automaton (NFA). In order to match a given string, the characters of the string are used as input into the NFA. If the last state is an end state, then the string matches. As long as the NFA has been created during preprocessing, this technique is efficient, since string matching can be executed in $\mathcal{O}(n)$ steps, where n is the length of the input string.

Our method is similar to regular expression matching. Rather than having characters as state input, we take instructions as input. In addition, our patterns are not represented using a specific language like regular expressions. The patterns are linear and not very complex. For instance, they do not contain complex constructs like loops. Hence, the construction of the pattern matching tree is straightforward.

For a given instruction, such as a `mov`, there are many different ways to encode it, depending on the addressing mode used, immediate values or registers. Building a pattern tree out of all those combinations would not be practical. Therefore, the immediate values and registers are replaced with *placeholders* that are allocated in an ascending order. Figure 3.9 shows a pattern tree that can match two patterns.

The instruction pointer, `%rip`, is not considered as a register and will not be replaced by a placeholder. Its value is known, like a constant, and is directly used later when computing the targets.

3.3.2 Watch List

When recovering an indirect jump, the algorithm needs to process all dependent instructions. The slicing of instructions is done using a *watch list*. At each node in the graph $\mathcal{G}$, four kinds of actions can be specified :

— add a register to the watch list,
— remove a register from the watch list,
— add an instruction to the watch list and
— remove an instruction from the watch list.

When the next instruction is encountered, the watch list is used to check if a register of interest was written or if an instruction of interest was encountered. Pattern matching is achieved when an end node is encountered, or if the watch list is empty. The main use of the watch list is that we can stop matching the pattern as early as possible, and we can match instructions, such as `cmp`, that do not write to any register.

3.3.3 Pseudo-Code

Let $\mathcal{G}$ be the graph of nodes $\mathcal{N}$, $\mathcal{W}$ be the watch list, $\mathcal{I}$ and $\mathcal{I}_{\text{p}}$ be an instruction without and with placeholders respectively. The context $\mathcal{C}$ is used to keep track of which register or constant placeholders have been allocated. Algorithm 1 shows the pseudo-code of the pattern matching function.

At the start, the first node is the root of the graph, it is a special node that only matches if an

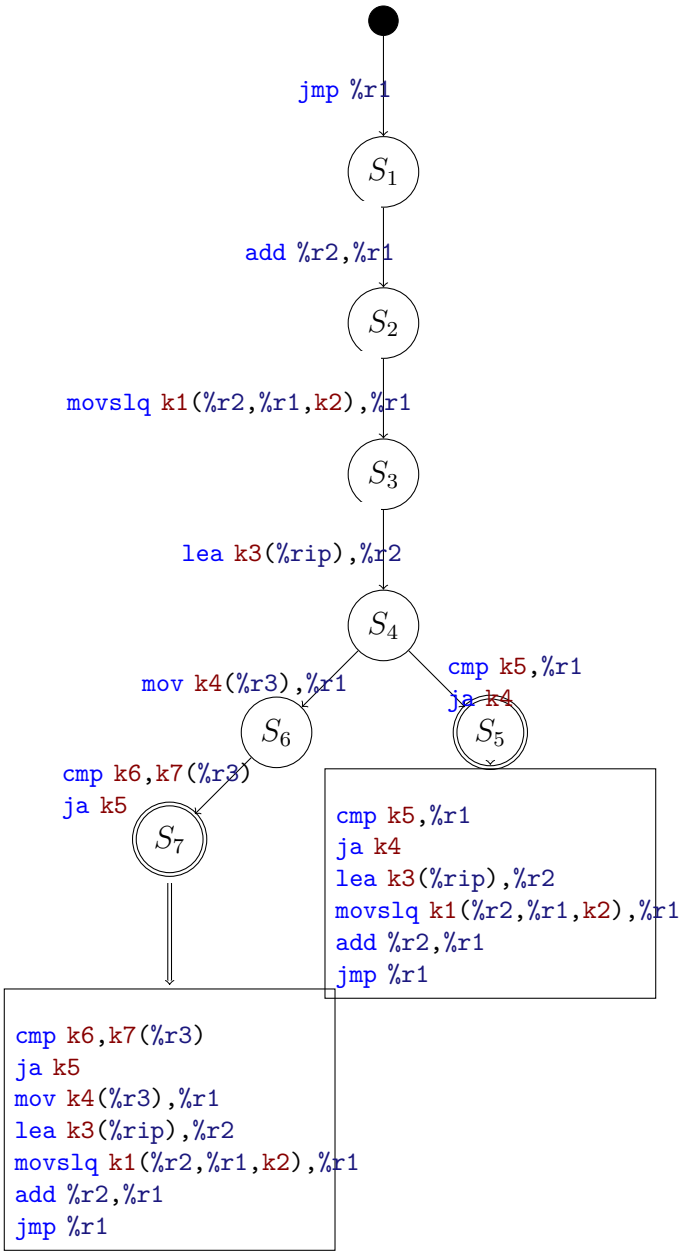


FIGURE 3.9 Example of a pattern tree that can match 2 indirect jump patterns. The patterns are mostly identical, except that the jump index is memory-addressed in the left one, while it is register-addressed in the right pattern.

indirect jump is encountered. From there, the loop tries to match the previous instructions.

Algorithm 1 Pattern matching indirect jumps

```
function MatchIndirectJumps(G, I)
    N ← GetRoot(G)
    W ← ApplyActions(W, N)
    C ← Context(∅)

    for W ≠ ∅ & I ≠ ∅ & N ≠ ∅, I ← Next(I) do
        I_p ← WithPlaceholders(C, I)
        if Matches(W, I_p) then
            C ← CommitPlaceholders(C, I)
            N ← Follow(N, I_p)
            W ← ApplyActions(W, N)
        end if
    end for

    if IsNotEndNode(N) then return ∅
    return ComputeTargets(C, N)
end function
```

The function $\textsc{Next}(\mathcal{I})$ returns the source instruction of $\mathcal{I}$. In most cases, it is the previous instruction, but it is possible that the source is a direct jump. The case where multiple jumps land into the current instruction is not handled. Supporting it would require splitting the context and following multiple branches at the same time. While it is not necessarily difficult to handle, it was not implemented because of the additional processing involved, and the fact that those cases were rare in the applications that we examined.

Each instruction is first converted into an equivalent instruction $\mathcal{I}_\mathrm{p}$ using placeholder values. This instruction is then filtered using the watch list. If it matches, any placeholder that has been allocated is committed into the context. Thereafter, we continue with the next node.

When the watch list is empty, the function start was reached or a pattern end node was matched, and the algorithm has finished. In order to compute the targets, the node must be an end node. Otherwise, the algorithm fails. The function $\textsc{ComputeTargets}(\mathcal{C}, \mathcal{N})$ uses the process memory and the various constant values that have been matched to compute the targets. It will also do some sanity checks, depending on which pattern was matched. For example, the jump index may be accessed multiple times in a pattern. If the addressing mode is relative to a register, then the offset has to be the same each time the values are accessed. If not, the algorithm fails to recover the targets.

3.3.4 Example

Figure 3.10 shows a simple list of instructions that can be matched by the pattern tree in Figure 3.9. Starting from the indirect jump and going up, the algorithm will encounter the following instructions :

1. The algorithm first start by matching `jump %rax`. Since `%rax` as not encountered before, it is allocated as `%r1` and added to the watch list.
2. The algorithm then encounter `mov $4,%rsi`. Since it does not write to any instruction in the watch list, this instruction is not matched and not registers are allocated or added to the watch list.
3. The algorithm next matches `add %rcx,%rax`. It adds `%rcx` into `%rax`, which is in the watch list. Register `%rcx` is allocated as `%r2` and added to the watch list.
4. The algorithm next matches `movslq (%rcx,%rax,4),%rax`. The register `%rax` is written. No new registers are allocated and the watch list is unchanged. The constants `k1=0` and `k2=4` are saved.
5. The algorithn next matches `lea -132(%rip),%rcx`. It writes a constant into `%rcx`. Register `%rcx` (`%r2`) si removed from the watch list. The constants `%rip[0]` and `k3=-132` are saved.
6. The algorithm next matches `ja 8342`. It is a branch and we want to find the corresponding `cmp`. Hence, we add `cmp` to the watch list. The constants `k4=8342` is saved.
7. The algorithm finally matches `cmp $5,%r1` because it is in the watch list. The constants `k5=5` is saved and the algorithm stops because we are at a terminal node.

When ComputeTargets$(\mathcal{C}, \mathcal{N})$ would be called, it would return the set of targets described by the following expression :

$$\begin{aligned}&\left\{r_1 + (k_3 + rip_0) + mem\left[k_1 + (k_3 + rip_0) + k_2 \times r_1\right] \,\middle|\, \forall r_1 \in [0, k_5]\right\} \\ &= \left\{r_1 + (rip_0 - 132) + mem\left[(rip_0 - 132) + 4 \times r_1\right] \,\middle|\, \forall r_1 \in [0, 5]\right\}.\end{aligned}$$

3.3.5 Limitations

This method should be able to match materialized jumps. However, none were actually encountered in our testing. As mentioned earlier, x86-64 compilers rarely, if ever, generate them in normal circumstances. In addition, this method does not support tail call optimizations. While it may be possible to match certain patterns, they are hard to predict. Defining patterns for each of them would certainly be inefficient.

```
cmp $5,%rax
ja 8342
lea -132(%rip),%rcx
movslq (%rcx,%rax,4),%rax
add %rcx,%rax
mov $4,%rsi
jmp %rax
```

FIGURE 3.10 Example of few instructions before an indirect jump

Multiple Patterns from Multiple Paths

When rewinding instructions from an indirect jump, it is possible that multiple paths land on a given instruction (e.g., it is the target of another jump). For example, a jump table pattern always checks that the index is bounded. Figure 3.11 shows an example where the bound check is bypassed with the constant value 0.

```
    mov $0,%rax
    jmp JUMP
    # <other instructions>
    cmp $8,%rax
    ja SKIP
JUMP:
    lea 1000(%rip),%rbx
    movslq (%rbx,%rax,4),%rax
    add %rbx,%rax
    jmp %rax
SKIP:
    # <other instructions>
```

FIGURE 3.11 Jump table with two patterns from two code paths.

Implementing multiple patterns is a relatively easy improvement to the algorithm. When rewinding instructions, and finding a path fork, the context should be copied to a new path. Both contexts could then continue independently, and their results later be merged. Other corner cases may also appear, such as cycles in paths. To keep the algorithm simple, these cases were left unsupported. Indeed, no such cases were observed during our experimentations.

Multiple Indirect Jumps in a Function

Our algorithm does not support cases where there are multiple indirect jumps in a single function. This is in part because we do not support multiple paths, and because it creates a circular dependency between two or more indirect jumps. While it would probably, in most cases, be safe to recover them and then instrument functions based on the targets, we decided not to take the risk, since those are corner cases.

Figure 3.12 shows a graph of the number of indirect jumps per function. We analyzed a large number of binary executables in order to get a clear idea of the indirect jumps distribution. In our tests, only 19% of the functions contained one or more indirect jumps. Among those functions, only 18% contained 2 or more indirect jumps, and thus cannot be supported by our implementation of the proposed algorithm.

Calling Convention

There are some cases where a value in a jump pattern, such as the index, was also used as an argument to a function call. In the System V ABI for x86-64, some registers, such as `%rbx`, are preserved across function calls, in which case the call instruction could be ignored if we were tracking only the `%rbx` register [57]. This is presently not supported, but very few cases were found.

Similar Patterns

About 20 patterns were used for the experiments in the next section. Some of those patterns are very similar, varying only by one instruction. For example, if the jump index is a global variable, it can be accessed with relative-addressing. However, if the jump index is a local

TABLEAU 3.1 `uftrace` metrics gathered from different binaries.

Program	Total	Before		After	
		Patched	Failed	Patched	Failed
`binutils`	18689	16802 (89.90%)	1887 (10.10%)	17310 (92.62%)	1379 (7.38%)
`dbus`	566	519 (91.70%)	47 (8.30%)	527 (93.11%)	39 (6.89%)
`git`	5420	5124 (94.54%)	296 (5.46%)	5182 (95.61%)	238 (4.39%)
`nginx`	1188	1117 (94.02%)	71 (5.98%)	1126 (94.78%)	62 (5.22%)
`pulseaudio`	134	122 (91.04%)	12 (8.96%)	128 (95.52%)	6 (4.48%)
`systemd`	6328	6025 (95.21%)	303 (4.79%)	6151 (97.20%)	177 (2.80%)
`vim`	5657	5036 (89.02%)	621 (10.98%)	5152 (91.07%)	505 (8.93%)

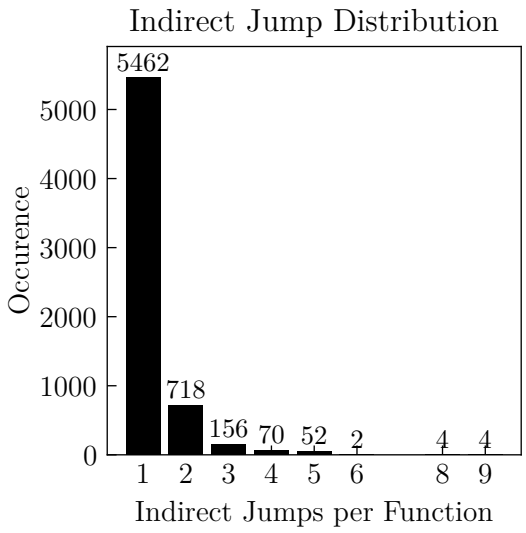


FIGURE 3.12 Distribution of indirect jumps per function.

variable, it may be in a register or on the stack. Finally, if it is within a structure, it may use indirect addressing. Some patterns have four variations, to cover these four cases, but they are all specified separately. It would be interesting to cover those multiple cases simultaneously, as an improvement to this method, to reduce the number of almost identical patterns.

3.4 Results

The new proposed algorithm has been implemented as a C++ library [49]. Then, `uftrace` [37] was modified to use this library when it encounters indirect jumps in its dynamic instrumentation implementation. The `uftrace` tracer was chosen because of its mature userspace dynamic instrumentation for the x86-64 architecture, and also because it uses the same disassembler, Capstone [50].

All programs were compiled with `gcc` using option `-O2`, in order to emulate a production environment, where dynamic tracing is typically used. Link-time optimizations were disabled, because not all test programs would compile otherwise.

3.4.1 Test Environment

All the test programs were compiled using GCC version 12, with and without the custom patches presented below.

3.4.2 Compiler Emission of Jump Table Informations

When the library recovers a jump table, it outputs a set t containing n target addresses. There are many ways in which the result could be incorrect, t could underapproximate or overapproximate the real set of target addresses.

In order to validate the results, GCC was modified to output additional debugging information concerning where jump tables are located in the read-only binary. It did so by emitting unique symbols at the start and end of each jump table. Using this debugging information, the jump table could be extracted.

Given a binary, the following is extracted :

— The *ground truth* represents the *true* jump tables, as generated by the compiler and extracted from the debugging informations.

— The *recovered* jump tables are obtained from running `uftrace` with the proposed new algorithm.

— The *valid* recovered jump tables are all recovered tables that are in the ground truth.

— The *invalid* recovered jump tables are all recovered tables that are not exactly in the ground truth.

— The *missing* recovered jump tables are all true jump tables that are not in the recovered tables.

Table 3.2 presents these results extracted from multiple binaries from different projects.

3.4.3 Built-In Patching Metrics

With verbose and debug output activated, unmodified `uftrace` already prints information concerning the number of functions where patching succeeded, failed or was skipped. This

TABLEAU 3.2 Recovered jump tables from different binaries.

Program	Ground Truth	Recovered Tables		Missing
		Valid	Invalid	
`binutils`	1363	661 (48.50%)	0 (0.00%)	702 (51.50%)
`dbus`	8	8 (100.00%)	0 (0.00%)	0 (0.00%)
`git`	72	68 (94.44%)	0 (0.00%)	4 (5.56%)
`nginx`	34	13 (38.24%)	0 (0.00%)	21 (61.76%)
`pulseaudio`	6	6 (100.00%)	0 (0.00%)	0 (0.00%)
`systemd`	177	155 (87.57%)	0 (0.00%)	22 (12.43%)
`vim`	241	172 (71.37%)	0 (0.00%)	69 (28.63%)

data was collected for multiple binaries, from different projects, before and after resolving indirect jumps with the new proposed algorithm. Table 3.1 presents those results.

3.4.4 Instrumentation Improvements

Table 3.1 shows an interesting reduction in the number of functions that could not be instrumented. On average, about 27% fewer functions could not be instrumented. It is important to note that for the functions that were not instrumented, while some were due to indirect jumps (e.g., other jump table or tail-optimized call), many were not caused by indirect jumps, but by other unsupported cases in the `uftrace` dynamic instrumentation. Its implementation is not as mature as the Kprobe implementation. For example, its checks concerning which instructions can be executed out-of-line is incomplete. Hence, many of those cases would be supported by a more complete dynamic instrumentation implementation.

Table 3.2 provides insight into how many more functions could be instrumented by resolving indirect jumps. On average, about 77% of the indirect jumps coming from jump tables are recovered. It is also important to note that, out of the recovered tables, there were no invalid recoveries. In other words, each recovered jump table correctly approximates the true jump table.

All 23 patterns that were used for these experiments come from the analysis of the `git` binary. This explains why its results are slightly better than other programs with a similar function count. In the future, other patterns can easily be added, as the indirect jump from other programs are examined. It is one advantage of this method, adding a new pattern is almost trivial.

Similar experiments were run on binaries using no optimization (`-O0`). Before resolving indirect jumps, `uftrace` could instruments about 5% more functions. When we recovered indirect jumps, we could also instruments about 5% more functions than the binary with optimizations. In some cases, we were able to instrument 100.00% of the functions. On non-optimized binaries, we were able to recover about 12% more jump tables on average, with more binaries where all jump tables were recovered.

For binaries that could be compiled with link-time optimizations, unmodified `uftrace` could patch about 1% fewer functions. When we resolved indirect jumps, fewer than 1% functions could not be instrumented, in comparison to the `-O2` binaries. Finally, the number of jump tables that were recovered decreased approximately by the same amount.

When quickly analyzing the remaining jump tables, we found that they were mostly cases of the limitations listed earlier. Exact numbers are not available since classifying them is

very time consuming or requires an implementation without those limitations. Nonetheless, we randomly sampled 111 unrecovered indirect jumps and manually classified them in Table 3.3. We can see that the biggest improvements would come by resolving tail call optimizations.

3.4.5 Instrumentation Cost

Additional benchmarks were run over the same programs to measure the instrumentation cost. We first measured the time it takes to instrument a function without resolving indirect jumps. Then, we remeasured that time when indirect jumps were being resolved. We also gathered data about the function instruction count. Figure 3.13 shows those results with respect to the instruction count in the function.

Before resolving indirect jumps, the relationship between the instrumentation time and the instruction count is linear. This is expected since `uftrace` needs to analyze the entire function before determining that it is safe for instrumentation. When resolving indirect jumps, we see a second linear relationship, just above the first one. This is for the functions containing a single indirect jump that needs to be resolved. The second line has a constant offset from the first one. This can be explained by the constant maximum depth of about 8, for our instruction pattern matching tree. Thus, about the same number of instructions need to be checked each time.

Nevertheless, there are several outlier points above the second line. This can be explained by two things. First, even though our instructions tree has a constant depth, it is possible that the pattern is sparse and a lot of independent instructions are not matched. Secondly, different indirect jumps may target different numbers of locations. For example, we might recover an indirect jump with 300 target locations, and another with only 5. Upon closer examination, we could confirm that all the outliers were linked to indirect jumps with a large number of target locations.

TABLEAU 3.3 Classification of indirect jumps recovery failure.

Type	Count
Tail Call	71 (63.97%)
Missing Pattern	5 (4.50%)
Multiple Branches	6 (5.41%)
Multiple Indirect Jumps	29 (26.12%)

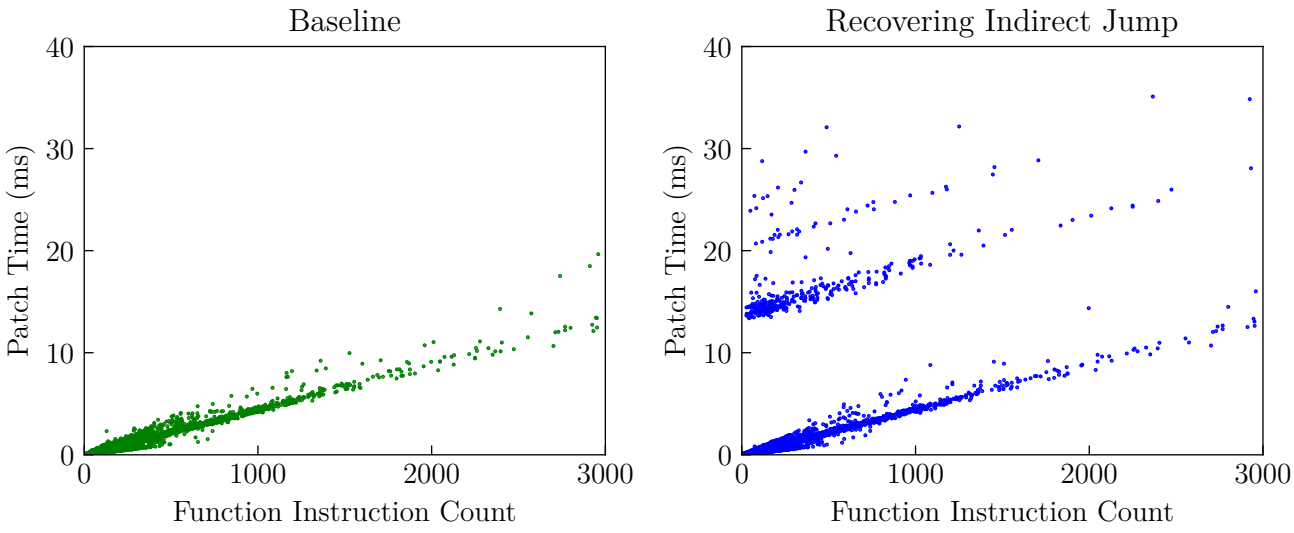


FIGURE 3.13 Instrumentation cost before and after resolving indirect jumps.

3.5 Discussion

3.5.1 Tail-Optimised Calls

Our proposed method currently does not support indirect jumps from tail-call optimization. While not implemented, we found multiple methods that could be explored.

The first thing would be to try analyzing the instructions around the jump. We can detect some of them with good certainty simply by looking at the few instructions preceding the indirect jump. Based on the calling convention, certain registers need to be preserved across function calls. Hence, the prologue of functions that overwrite these registers will push these values on the stack. The epilogue of these functions will pop the values back into the right registers. If the function is tail-optimized, it still has to restore those registers. Hence, if we detect that these registers get popped right before the indirect jump, this is a great indication that it might be an indirect jump. Nevertheless, it is not a certainty and there would still be a risk if we instrumented the function. This technique could perhaps be used with a special option that the user can specify to use insecure, but probably safe techniques.

DWARF version 5 [21] added attributes to debugging information concerning call sites. If we assume that function calls always land at the start of a function, we can use the newly added attribute `DW_AT_call_site_pc` to determine if the indirect jump is a tail-optimized call. At present time, however, this attribute is not emitted consistently across binaries, or between different compilers.

Another method is to take advantage of binaries that are compiled with the Intel Control-flow Enforcement Technology (CET) [33]. In those binaries, all indirect jumps/calls must land on a `endbr` instruction, except if the `notrack` prefix is present. Jump table indirect jumps usually have the prefix present, but not tail-optimized calls. In that case, we could assume that all tracked indirect jumps are function calls, and land at the start of another function. If some compilers decide to remove the `notrack` prefix from a jump table branch, then we simply need to check where the `endbr` instructions in the function to determine its targets.

Finally, we could try extracting the control flow graph from the compiler itself. GCC offers the option `-fdump-tree-vcg-lineno` that can dump the CFG. However, it is from a higher level than the assembly that it generates. Hence, it might be difficult to match the CFG to the instructions of a function. We could also add the `-save-temps` arguments in order to save temporary files generated by GCC. This includes the compiled assembly with all the jump and raw jump tables. With this method, we would not even need to recover indirect jumps, since all the jump labels are available. This solution is viable in tracing environments, since we usually have access to the build process or the source code, unlike in reverse engineering environments. Of course, it is important that the binary and the assembly files match exactly.

3.5.2 Extending debugging information

In our experiments, we briefly talked about compilers emitting jump table information. If we want to go further and eliminate the problem of indirect jump recovery entirely, it would be interesting to add this information inside the standard binary debugging information. This is a valid solution because we target environments where debugging information is expected to be available, either inside the binary or in a separate file. The main challenges would be finding other use cases to this debug information. For example, GDB fast tracepoints also use 4 bytes jumps, but it does not support the overwriting of multiple instructions. If it ever supports that, it could benefit from this additional debug information.

One way of doing this would be to modify existing compilers to emit an additional binary section. This new section would contain an array of structures, where each would contain the address of an indirect jump, and all of its possible target addresses.

Another way is to add a new DWARF die that would contain attributes describing the targets of an indirect jump. The challenge would be to justify the addition. The DWARF debugging format is mainly intended for use in debuggers, such as GDB.

3.6 Conclusion

Recovering indirect jumps for dynamic instrumentation brings fewer constraints than binary analysis and reverse engineering. Pattern matching is an interesting solution in that case, since it is both simpler and faster, while achieving an excellent success rate. In our experiments, each recovered jump table contained exactly the original targets. We were able to significantly reduce the number of instructions that previously could not be instrumented with `uftrace`. Moreover, most of the remaining functions were not due to indirect jumps, but rather to other unhandled cases in `uftrace`. Similar improvements could be expected in the Linux Kernel Kprobe, if this new proposed algorithm was incorporated. As future work, it will be interesting to resolve tail-optimized calls and the other remaining unhandled cases. With the work presented here, and some of the improvements suggested, we are getting closer to an efficient solution to dynamically instrument almost 100% of the functions in a binary.

3.7 Acknowledgement

The financial support of Ericsson, Ciena, AMD, EfficiOS, Prompt and the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

CHAPITRE 4 DISCUSSION GÉNÉRALE

Contrairement aux méthodes comme E9Patch ou DynTrace, nous avons montré qu'il est possible de résoudre le graphe de flot contrôle afin d'améliorer la couverture de l'instrumentation. Un avantage de notre méthode versus celles nommées plus haut est qu'elle permet d'avoir des sondes plus efficaces en matière de temps d'exécution et aussi en matière de mémoire.

Si une implémentation existante d'instrumentation dynamique cherche à augmenter sa couverture, il suffit d'implémenter notre algorithme, ou d'utiliser notre bibliothèque afin de recouvrir les cibles des sauts.

Un autre aspect important de notre méthode approxime exactement les cibles. Une sur approximation aurait été acceptable, car cela diminue tout simplement la probabilité qu'une sonde optimisée puisse être insérée. Une sous-approximation est inacceptable, car cela peut amener à instrumenter une instruction qui ne devrait pas l'être. Contrairement à littérature consultée, nous avons réussi à concevoir une méthode automatique afin de vérifier l'exactitude des sauts recouverts. Pour ce faire, nous avons modifié le compilateur afin d'obtenir les vrais sauts. Par la suite, il était tout simplement question de comparer nos sauts avec la vérité. Dans la littérature, la méthode de vérification était souvent omise. Sinon, elle se faisait soit à la main ou en utiliser un autre algorithme pouvant recouvrir les sauts indirects.

CHAPITRE 5 CONCLUSION

5.1 Synthèse des travaux

Dans ce projet de recherche, nous avons avancé les techniques existantes de traçage dynamique afin d'améliorer la couverture de leur instrumentation. Pour ce faire, nous avons modifié une implémentation existante afin de résoudre les cibles des sauts indirects dans les fonctions instrumentées. Auparavant, celles-ci n'étaient jamais résolues et la fiabilité des sondes ne pouvait pas être prouvée. L'instrumentation devait souvent recourir à une méthode considérablement moins rapide.

Pour ce faire, nous avons étudié différentes méthodes effectuant cette tâche dans le domaine de la traduction dynamique et la rétro-ingénierie de programmes. Nos requis de recouvrement des cibles étant moins demandants que ceux de ces domaines, nous avons opté pour une méthode basée sur la correspondance de patron. Nous sommes donc en mesure de retrouver une majorité des sauts indirects provenant des tables de sauts. Les cibles de chaque saut indirect analysées sont déduites exactement, contrairement à des méthodes plus avancées qui peuvent générer des approximations. De plus, cette méthode est très rapide, car elle est basée sur un arbre de recherche.

5.2 Limitations de la solution proposée

La méthode implémentée est optimisée pour les sauts indirects provenant de cibles reconstruites ou de table de sauts. Les sauts indirects provenant de l'optimisation de fin d'appel ou d'assembleur personnalisé ne sont pas supportés. Dans les cibles reconstruites, leurs formes ne sont pas assez consistantes entre elles pour y extraire des patrons les représentants. Dans les tables de sauts, il est difficile de définir des patrons pour tous les types d'utilisations possibles d'un saut indirect. Néanmoins, ces cas sont rares, car une faible proportion de programmes est maintenant écrite en assembleur.

La méthode implémentée n'est toutefois pas complète. Certaines tables de sauts ne furent pas découvertes. Dans quelques cas, l'algorithme a découvert un patron qu'il ne connaissait pas. Dans d'autres cas, il existait plusieurs patrons vers un même saut indirect. Quoique ces derniers cas ne sont pas extrêmement difficiles à implémenter, ceux-ci représentaient une minorité des cas non supportés.

5.3 Améliorations futures

Selon nos analyses, les sauts indirects provenant d'une optimisation de fin d'appel sont maintenant les cas les plus nombreux qui n'ont pas été trouvés. Par conséquent, ceux-ci devraient être la cible des améliorations futures.

Nous avons discuté de plusieurs méthodes possibles afin d'extraire la cible de ces sauts. Dans tous les cas, on exploite le fait que le domaine du traçage a accès à l'information de débogage ou au code source. On se permet d'utiliser de l'information additionnelle. Une méthode devant être étudiée plus en détail est celle d'utiliser l'information contenue dans la version 5 du format DWARF [21]. Celui-ci expose de l'information concernant les appels. Cette information est liée à une instruction. Si cette instruction est un saut indirect, nous pouvons donc le considérer comme une fin d'appel optimisée.

Pour les programmes étant compilés avec Intel Control-flow Enforcement Technology (CET) [33], il est possible d'exploiter l'information additionnelle contenue dans les instructions. Cette technologie augmente la sécurité d'une application en empêchant certaines attaques logicielles basées sur les sauts ou appels indirects. Elle s'assure que tous les sauts ou appels indirects tombent sur une instruction de type `endbr`. En d'autres termes, les cibles de tous les sauts indirects d'une fonction peuvent facilement être découvertes. Il suffit de regarder où sont les instructions `endbr`. Seulement les processeurs modernes de Intel supportent cette technologie. Par contre, les instructions qu'elle ajoute sont perçues comme des NOPs sur les anciens processeurs. Par conséquent, il est possible de compiler un programme avec cette technologie seulement afin d'extraire les cibles des sauts indirects.

En résumé, il existe plusieurs autres techniques à étudier pour trouver les cibles des sauts indirects provenant d'une fin d'appel optimisée. Il existe aussi d'autres cas, représentant une minorité des cas non supportés. Cependant, supporter les sauts indirects provenant d'une fin d'appel optimisée représenterait la plus grande addition à la technique.

RÉFÉRENCES

[1] *About DTrace*. Accessed: 2021-10-04. URL : http://dtrace.org/blogs/about/.

[2] Gogul Balakrishnan et Thomas Reps. "Analyzing memory accesses in x86 executables". In : *International conference on compiler construction*. Springer. 2004, p. 5-23.

[3] Anas Balboul, Ahmad Shahnejat Bushehri et Michel Dagenais. "NOProbe: A Fast Multi-Strategy Probing Technique for x86 Dynamic Binary Instrumentation". In : (2020).

[4] Andrew R Bernat et Barton P Miller. "Anywhere, any-time binary instrumentation". In : *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 2011, p. 9-16.

[5] Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson et Ning Wang. "XRay: A function call tracing system". In : (2016).

[6] Bryan Buck et Jeffrey K Hollingsworth. "An API for runtime code patching". In : *The International Journal of High Performance Computing Applications* 14.4 (2000), p. 317-329.

[7] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro et Ryan R Newton. "Instruction punning: Lightweight instrumentation for x86-64". In : *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, p. 320-332.

[8] Cristina Cifuentes et Mike Van Emmerik. "Recovery of jump table case statements from binary code". In : *Science of Computer Programming* 40.2-3 (2001), p. 171-188.

[9] Lucian Cojocar, Taddeus Kroes et Herbert Bos. "JTR: A binary solution for switch-case recovery". In : *International Symposium on Engineering Secure Software and Systems*. Springer. 2017, p. 177-195.

[10] Jonathan Corbet. *Extending extended BPF*. https://lwn.net/Articles/603983/. Accessed: 2021-10-08.

[11] Jonathan Corbet. *Introducing utrace*. https://lwn.net/Articles/224772/. Accessed: 2021-10-20.

[12] Jonathan Corbet. *Kernel building with GCC plugins*. https://lwn.net/Articles/691102/. Accessed: 2021-10-20.

[13] Jonathan Corbet. *The RCU-tasks subsystem*. https://lwn.net/Articles/607117/. Accessed: 2021-10-20.

[14] Jonathan Corbet. *Uprobes in 3.5*. https://lwn.net/Articles/499190/. Accessed: 2021-10-20.

[15] Leonardo De Moura et Nikolaj Bjørner. "Z3: An efficient SMT solver". In : *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, p. 337-340.

[16] *Debugging with GDB*. The GNU Operating System et the Free Software Movement. 2021. url : https://sourceware.org/gdb/current/onlinedocs/gdb.pdf.

[17] Mathieu Desnoyers. *Using the Linux Kernel Tracepoints*. https://www.kernel.org/doc/html/latest/trace/tracepoints.html. Accessed: 2021-10-02.

[18] Mathieu Desnoyers et Michel R Dagenais. "The lttng tracer: A low impact performance and behavior monitor for gnu/linux". In : *OLS (Ottawa Linux Symposium)*. T. 2006. Citeseer. 2006, p. 209-224.

[19] Alessandro Di Federico et Giovanni Agosta. "A jump-target identification method for multi-architecture static binary translation". In : *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. 2016, p. 1-10.

[20] Gregory J Duck, Xiang Gao et Abhik Roychoudhury. "Binary rewriting without control flow recovery". In : *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, p. 151-163.

[21] *DWARF Debugging Information Format*. Version 5. DWARF Debugging Information Format Committee. Fév. 2017. url : http://www.dwarfstd.org/DownloadDwarf5.php.

[22] Michael J. Eager. "Introduction to the DWARF Debugging Format". In : (2012).

[23] Mike Frysinger. *function tracer guts*. https://www.kernel.org/doc/Documentation/trace/ftrace-design.txt. Accessed: 2021-09-27.

[24] *GDB Internals Manual*. Accessed: 2021-10-11. url : https://sourceware.org/gdb/wiki/Internals.

[25] Taras Glek. *DeHydra Source Analysis Tool*. 2007.

[26] Victor Mikhaylovich Glushkov. "The abstract theory of automata". In : *Russian Mathematical Surveys* 16.5 (1961), p. 1.

[27] *GNU gprof – The GNU Profiler*. Free Software Foundation. 1998. URL : https://www.cs.tufts.edu/comp/150PAT/tools/gprof/gprof.pdf.

[28] Brendan Gregg. *Choosing a Linux Tracer*. https://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html. Accessed: 2021-10-27. 2015.

[29] Brendan Gregg. *Linux uprobe: User-Level Dynamic Tracing*. https://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html. Accessed: 2021-10-08.

[30] Christian Harper-Cyr, Michel R Dagenais et Ahmad S Bushehri. "Fast and flexible tracepoints in x86". In : *Software: Practice and Experience* 49.12 (2019), p. 1712-1727.

[31] Hex Rays. *IDA Pro*. Version 7.6. URL : https://hex-rays.com/ida-pro/.

[32] Yuanjie Huang, Liang Peng, Chengyong Wu, Yuriy Kashnikov, Jörn Rennecke et Grigori Fursin. "Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation". In : *2nd International Workshop on GCC Research Opportunities (GROW'10)*. 2010.

[33] *Intel*. 334525-002. Revision 2.0. Control-flow Enforcement Technology Preview. Juin 2017.

[34] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel Corporation. Juin 2021. URL : https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[35] Jim Keniston et Prasanna S Panchamukhi. *Kernel Probes (Kprobes) – v2.6.17*. https://kernel.googlesource.com/pub/scm/linux/kernel/git/penberg/linux/+/v2.6.17-rc6/Documentation/kprobes.txt. Accessed: 2021-09-27.

[36] Jim Keniston, Prasanna S Panchamukhi et Masami Hiramatsu. *Kernel Probes (Kprobes)*. https://www.kernel.org/doc/html/latest/trace/kprobes.html. Accessed: 2021-09-27.

[37] Namhyung Kim. *Function graph tracer for C/C++/Rust*. 2021. URL : https://github.com/namhyung/uftrace.

[38] Jim Kineston et Srikar Dronamraju. "Uprobes: User-Space Probes". 2010. URL : https://events.static.linuxfound.org/slides/lfcs2010_keniston.pdf.

[39] *LLVM Documentation*. Version 8. LLVM Project. Août 2018. URL : https://buildmedia.readthedocs.org/media/pdf/llvm/latest/llvm.pdf.

[40] Brandon Lucia et Luis Ceze. “Data provenance tracking for concurrent programs”. In : *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, p. 146-156.

[41] William Mahoney et J Todd McDonald. “Enumerating x86-64–It’s Not as Easy as Counting”. In : ().

[42] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy et Masami Hiramatsu. “Probing the guts of kprobes”. In : *Linux Symposium*. T. 6. 2006, p. 5.

[43] Paul E. McKenney. *A Tour Through RCU’s Requirements*. `https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/Requirements.html`. Accessed: 2021-10-27.

[44] Ingo Molnár et al. *perf: Linux profiling with performance counters*. `https://perf.wiki.kernel.org`. Accessed: 2021-10-14.

[45] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 2014.

[46] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Rapp. tech. University of Cambridge, Computer Laboratory, 2004.

[47] *Oracle Linux – DTrace Guide*. Révision 10882. Oracle. Oct. 2020. url : `https://docs.oracle.com/en/operating-systems/oracle-linux/dtrace-guide/`.

[48] Pradeep Padala. “Playing with ptrace, Part 1”. In : *Linux Journal* 2002 ().

[49] Gabriel-Andrew Pollo-Guilbert. *Resolving N-conditionals indirect jump using pattern matching*. 2021. url : `https://github.com/gpollo/libresolver`.

[50] Nguyen Anh Quynh. *Capstone disassembly/disassembler framework*. 2021. url : `https://github.com/aquynh/capstone`.

[51] Steven Rostedt. *ftrace – Function Tracer*. `https://www.kernel.org/doc/Documentation/trace/ftrace.txt`. Accessed: 2021-09-27.

[52] *strace – linux syscall tracer*. Accessed: 2021-10-04. url : `https://strace.io/`.

[53] Yulei Sui et Jingling Xue. “SVF: interprocedural static value-flow analysis in LLVM”. In : *Proceedings of the 25th international conference on compiler construction*. 2016, p. 265-266.

[54] *SystemTap Beginners Guide – Introduction to SystemTap*. Édition 4.4. Red Hat. 2013. url : `https://sourceware.org/systemtap//SystemTap_Beginners_Guide.pdf`.

[55] *The Common Trace Format – A Flexible, High-Performance Binary Trace Format*. Version 1.8.3. The DiaMon Workgroup. Mai 2020. URL : https://diamon.org/ctf/.

[56] *The RISC-V Instruction Set Manual*. Volume I: User-Level ISA. Document Version 2.2. RISC-V Foundation. Mai 2017. URL : https://riscv.org/technical/specifications/.

[57] *The System V Application Binary Interface*. AMD64 Architecture Processor Supplement. Version 1.0. Various Authors. Jan. 2018. URL : https://gitlab.com/x86-psABIs/x86-64-ABI/.

[58] Ken THOMPSON. "Programming techniques: Regular expression search algorithm". In : *Communications of the ACM* 11.6 (1968), p. 419-422.

[59] *Using the GNU Compiler Collection*. Version 11.2.0. The GCC Developer Community. 2021. URL : https://gcc.gnu.org/onlinedocs/gcc-11.2.0/gcc.pdf.

[60] Martina VITOVSKÁ, Marek CHALUPA et Jan STREJČEK. "SBT-instrumentation: a tool for configurable instrumentation of LLVM bitcode". In : *arXiv preprint arXiv:1810.12617* (2018).