Titre: Title:	Points de trace rapides et efficaces par injection adaptative de sauts en x86
Auteur: Author:	Anas Balboul
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Balboul, A. (2020). Points de trace rapides et efficaces par injection adaptative de sauts en x86 [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5271/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/5271/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Points de Trace Rapides et Efficaces par Injection Adaptative de Sauts en x86

ANAS BALBOUL
Département de génie informatique et génie logiciel

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

Mai 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Points de Trace Rapides et Efficaces par Injection Adaptative de Sauts en x86

présenté par **Anas BALBOUL**
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 ma grand-mère. Sans elle, je n'aurais jamais pu arriver là où je suis aujourd'hui. Repose en paix.

Je dédicace ce mémoire à mes parents, ma famille et mes amis. Sans leurs encouragements mon travail n'aurait jamais vu le jour.

Je dédicace ce mémoire à tous ceux qui ont soutenu mon travail et m'ont donné le goût de le poursuivre.

Je dédicace ce mémoire à tous ceux qui m'apportent de l'amour et de la joie.

Je dédicace ce mémoire à tous ceux qui s'acharnent en cherchant le chemin vers la lumière.

REMERCIEMENTS

Je tiens d'abord à remercier mon directeur de recherche, Professeur Michel Dagenais pour l'opportunité offerte par lui. Son soutien, la qualité de son suivi et son expertise m'ont été d'une grande aide tout au long de mon projet de recherche.

J'aimerais adresser mes remerciements à l'ensemble de mes camarades et amis du laboratoire DORSAL qui ont contribué à mon projet de recherche à travers leur soutien ainsi que leur bonne humeur. Surtout merci à Amad pour toutes les idées et conversations intéressantes que nous avons partagées. Je tiens aussi à remercier Abder, Geneviève et Pierre Zins pour tous vos soutiens techniques et théoriques très utiles. Merci à Houssem et Hani et Adel pour vos conversations de bonne humeur.

Je tiens à exprimer ma gratitude à la participation financière d'Ericsson, Ciena, EfficiOS, Prompt et du conseil de recherches en sciences naturelles et en génie du Canada (CRSNG).

Finalement, merci aux lecteurs et notamment les Jurys pour leur intérêt porté à ce mémoire.

RÉSUMÉ

Le traçage est une technique efficace pour analyser la performance des systèmes complexes et parallèles. Il permet d'enregistrer les évènements du programme client dans une trace lorsqu'ils sont déclenchés par des points de trace. Dans les systèmes à fils d'exécution multiples, les évènements peuvent être déclenchés simultanément et apparaissent donc dans un ordre indéfini. Un point de trace avec un grand surcoût d'exécution peut changer l'ordre dans lequel les évènements apparaissent dans la trace. En conséquence, de grands problèmes de performance peuvent être masqués dans la trace. La plupart des techniques d'instrumentation existantes, soit sont trop intrusives pour le programme client (l'arrêtent complètement pour insérer l'instrumentation), soit ajoutent un grand surcoût d'exécution.

Dans ce mémoire, nous présentons un point de trace rapide en *x86* nommé NOProbe. Il utilise différentes stratégies d'instrumentation rapide dans le but d'augmenter l'efficacité. Pour l'une de ces stratégies, nous exploitons les rembourrages de NOP insérés par les compilateurs pour optimiser l'alignement du code. Pour exploiter le maximum possible de rembourrages sans affecter la performance, deux algorithmes sont proposés. Le premier est un algorithme glouton qui met l'accent sur la performance, ce qui le rend un choix intéressant pour l'instrumentation en lot. Le deuxième est localement optimal et sacrifie un peu de performance pour plus d'optimalité.
Pour limiter l'intrusion durant l'insertion du point de trace. Un algorithme de modification de code simultané et adapté à l'espace utilisateur, nommé *Lock and Load*, a été présenté. Sans arrêter le programme client et durant l'exécution, NOProbe se sert de l'algorithme *Lock and Load* pour insérer dynamiquement l'instrumentation. L'algorithme peut modifier le code d'un bloc de base tout entier. D'abord, la première instruction du bloc est remplacée par une instruction d'interruption logicielle pour le verrouiller. Ensuite, les fils d'exécution à l'intérieur sont redirigés en dehors du bloc avant de le modifier. Finalement, avant de déverrouiller la zone modifiée, tous les coeurs des CPU qui roulent les fils d'exécution du processus client sont sérialisés.
Notre solution offre un surcoût d'exécution très bas. Le coût d'insertion est 6 fois moins important que celui de Uprobe lorsque le nombre de fils d'exécution du programme instrumenté est inférieur à 16. Il est beaucoup moins important comparé a Dyninst. Malgré le fait que le coût d'insertion ne s'adapte pas bien au nombre de fils d'exécution dans le pire cas, notre technique surpasse en moyenne les autres techniques d'insertion dynamique.

ABSTRACT

Tracing is an effective technique to analyze the performance of complex multi-thread programs. A program trace consists of a series of events written by probes. The performance of a probe insertion and execution is very important for analyzing high-performance online systems. Existing techniques are either limited in scope, too intrusive or costly in overhead, limiting their applicability in many cases.

We introduce a new instrumentation technique for the x86 architecture, named NOProbe (NOP Probe), that exploits several complementary strategies in order to achieve a higher success rate in insertion of fast jump based probes at different locations in a binary. For one of these strategies, we exploit NOP-paddings inserted by compilers to optimize the code alignment. To achieve this goal, two new algorithms are introduced to assign paddings to probes. One is greedy and emphasizes performance, while the other attempts to locally optimize the assignation of paddings to probes in a function. We also propose an improved algorithm (*Lock* and *Load*) used by NOProbe to dynamically insert probes on the fly. The algorithm can patch a whole basic block by locking it with a trap instruction before redirecting the thread out of it. Next, the basic block is patched and the cores are serialized before unlocking it. The experimental results demonstrate that our approach not only achieves a higher success rate for fast probes insertion and maintains a very low execution overhead, but outperforms competing techniques.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CAS	Compare-And-Swap
CPU	Central Processing Unit, Unité Centrale de Traitement
ELF	Executable and Linkable Format
GFC	Graphe de Flot de Contrôle
GOT	Global Offsets Table
GPF	General Protection Fault
IPA	In-Process-Agent
IR	Représentation Intermédiaire
ISA	Instruction Set Architecture
JIT	Just-In-Time compilation
OLX	Execution Hors-ligne, Out-of-Line Execution
OS	Système d'Exploitation, Operating System
PC	Program Counter
PLT	Procedure Linkage Table
RCU	Read-Copy-Update
SMP	Symmetric multiprocessing
VM	Machine Virtuelle
XMC	Cross-Modifying Code

CHAPITRE 1 INTRODUCTION

L'apparition des systèmes hétérogènes complexes avec des processeurs qui contiennent un grand nombre de cœurs a créé des problèmes modernes et des défis de programmation que les développeurs doivent prendre en compte. Même les développeurs les plus prudents, lors de l'ajout d'un module ou d'une nouvelle fonctionnalité dans ces systèmes parallèles complexes, peuvent impacter négativement la performance, voire même introduire des bogues difficiles à reproduire. Le débogage peut être utilisé pour résoudre certains de ces problèmes en arrêtant le système lorsqu'un point d'arrêt est déclenché. L'utilisateur peut ensuite observer l'état du système lorsqu'une condition particulière se produit. L'arrêt partiel ou complet du système est rarement souhaitable pour identifier et reproduire les bogues de conditions de course dont l'occurrence dépend critiquement de la séquence d'évènements dans une trace. Le temps d'exécution du point de débogage est aussi critique et peut masquer la production de ces bogues. À cet effet, l'utilisation de points de trace performants qui minimisent l'interruption du système est plus appropriée.

L'instrumentation d'un programme peut être divisée en deux catégories : l'instrumentation statique et l'instrumentation dynamique. Un point de trace statique est un appel ajouté directement dans le code source et compilé avec le programme. Un point de trace dynamique est inséré dans un binaire après sa compilation sur disque ou dans la mémoire à la volée. Dans l'instrumentation dynamique, il est souvent compliqué de trouver de l'espace où insérer le point de trace. Pour cela, les outils de traçage peuvent employer deux techniques : l'utilisation d'une interruption logicielle qui consiste en une instruction d'un octet, ou bien l'utilisation d'un branchement relatif, qui consiste en une instruction de saut ou d'appel de fonction de cinq octets. Un branchement relatif est très performant, comparé à l'interruption. Cependant, sa taille de cinq octets le rend moins flexible dans certaines situations. L'interruption avec la taille minimum en *x86*, offre plus de possibilités et peut être insérée dynamiquement presque partout.

1.1 Définitions et concepts de base

1.1.1 Traçage

Le traçage est le processus de récolte de données de bas niveau sur l'état d'un programme sans avoir un grand impact sur sa performance. Il est différent de l'enregistrement des évènements de haut niveau dans un journal (*logging*). Il est aussi différent du débogage qui peut être trop intrusif, en arrêtant le programme pour la visualisation de son état. Dans le reste de cette sous-section, nous allons définir les concepts de base du traçage.

Évènement

Lorsqu'un point de trace est rencontré, une fonction de rappel est appelée et un évènement est déclenché. Cette fonction collecte les données pertinentes à l'évènement pour les agréger ou les enregistrer dans un tampon. Un exemple d'évènement peut être un changement de contexte dans le noyau, l'appel à l'ordonnanceur, la réception d'une requête par un serveur, etc.. Une suite d'évènement l'un après l'autre constitue une trace.

Point de Trace

L'instrumentation insère des points de trace dans un programme. Un point de trace est donc une abstraction de la composante qui déclenche un évènement pour l'enregistrer dans un tampon. Un point de trace ne doit pas interrompre l'exécution du programme. De plus, un point de trace doit être très performant et avec un faible surcoût pour ne pas impacter le traçage et l'exécution du programme. Les points de trace sont souvent utilisés dans l'environnement de production.

Points d'arrêt

Un point d'arrêt est souvent utilisé dans le débogage. Il peut être implémenté avec la même instruction utilisée par le point de trace (*int3*). La différence est qu'un point d'arrêt interrompt l'exécution du programme dans le but de permettre à l'utilisateur de naviguer dans le contexte à l'aide d'un débogueur interactif. Un point d'arrêt ne pose donc pas de contrainte sur la performance.

1.1.2 Instrumentation

L'instrumentation est le processus d'ajout ou de modification du code binaire ou code source dans le but de récolter de l'information et d'analyser l'exécution du programme. Pour tracer un programme, les outils de traçage instrumentent les endroits où des évènements importants peuvent être observés. Il existe deux domaines d'instrumentation : le domaine du noyau et le domaine de l'espace utilisateur. Chaque domaine a accès à des ressources et des implémentations de technique différentes et fait donc face à des défis différents. Il existe trois types d'instrumentation :

- L'instrumentation statique qui ajoute du code ou de l'espace durant/avant la compilation du programme. Le code source est nécessaire pour ce type d'instrumentation. L'instrumentation peut être complètement statique, et aucune insertion de code durant le temps d'exécution n'est nécessaire. Elle peut aussi être partiellement statique dans le cas où le compilateur réserve de l'espace qui sera utilisé pour insérer du code dynamiquement.
- L'instrumentation dynamique qui est ajoutée dans le binaire d'un programme après sa construction. Elle peut être insérée dans le code ISA du CPU physique. Puisqu'il n'y a

pas suffisamment d'espace pour insérer l'instruction d'instrumentation dans le code, les outils d'instrumentation peuvent remplacer une ou plusieurs instructions du code original. Les instructions originales sont donc enregistrées dans un tampon pour être exécutées hors-ligne.

— L'instrumentation dynamique qui est ajoutée dans du *bytecode* à l'aide de la traduction binaire dynamique qui utilise le principe de la compilation JIT. Une machine virtuelle prend le contrôle du programme client et compile ses instructions (souvent groupées dans un bloc de base) vers un code de représentation intermédiaire (*IR*) durant l'exécution. La VM exécute ensuite le code IR dans le contexte du programme client. Une fois que le bloc termine son exécution, la VM reprend le contrôle pour recommencer la même procédure pour le bloc suivant. Durant la compilation JIT, la VM peut simplement insérer du code d'instrumentation dans le code IR.

Instructions d'instrumentation

Dans l'instrumentation statique et dynamique, le code d'instrumentation est souvent une instruction qui change la valeur du PC pour brancher vers le code de traçage. Sur la plateforme *x86*, cela peut être une instruction d'interruption logicielle *int3*, un branchement relatif *jmp/call* ou une combinaison des deux. Malgré leur rare utilisation dans la revue de littérature, d'autres possibilité comme le branchement absolu et l'empilage d'une adresse suivi d'un retour peuvent modifier le compteur de programme.

Bloc de base et Graphe de flot de contrôle

Un bloc de base est une séquence d'instructions dont le contrôle d'exécution est obligé de passer à travers. Aucun branchement ne doit exister au milieu de cette séquence, à l'exception de la dernière instruction. Aucun branchement qui cible une instruction de la séquence ne doit exister, à l'exception de la première instruction. Dans un graphe de flot de contrôle (GFC), chaque sommet est un bloc de base. Les arêtes du graphe représentent des branchements qui relient deux blocs de base entre eux. Les outils d'instrumentation peuvent utiliser le GFC pour une analyse statique du programme et une insertion sécuritaire de points de trace. Certains outils peuvent aussi le modifier en insérant des sommets entiers en tant que code d'instrumentation.

1.2 Contributions

Dans ce mémoire, nous proposons trois contributions importantes dans le domaine de l'instrumentation dynamique et du traçage des systèmes informatiques :

1. Une technique d'instrumentation dynamique qui utilise des instructions de branchement. Purement basée sur les technologies disponibles dans l'espace utilisateur, notre technique combine plusieurs scénarios possibles pour atteindre un taux de succès élevé d'instrumentation avec branchement.
2. L'exploitation des rembourrages ajoutés par le compilateur pour placer un point de trace.
3. Un algorithme d'insertion de code simultané et adapté pour l'espace utilisateur est présenté. Il permet d'insérer l'instrumentation avec le moins d'intrusions possible, sans avoir à arrêter l'exécution. Cet algorithme tire avantage des signaux temps réel POSIX, de l'appel système *membarrier* et des interruptions interprocessus.
4. En utilisant l'algorithme d'insertion de code simultané introduit en 3) et en évitant d'utiliser des mécanismes de synchronisation coûteux, notre technique offre un coût d'insertion et d'exécution très bas. Cela la rend très efficace, notamment pour le traçage.

1.3 Plan

Ce mémoire est organisé de la façon suivante : les deux premières sections du chapitre 2 sont consacrées à la revue des techniques existantes de l'instrumentation statique et dynamique. La dernière section présente les outils existants de traçage et leurs relations avec les techniques d'instrumentation. La présentation du problème sous ses différents aspects se trouve au chapitre 3. Le chapitre 4 contient le corps du mémoire qui est un article scientifique soumis au journal ACM Transactions on Programming Languages & Systems. Ce dernier explique en détail les contributions originales présentées dans la section 1.2 et les solutions aux défis rencontrés. Finalement, le chapitre 6 conclut le mémoire avec un résumé de la discussion des résultats de notre contribution de recherche scientifique. Ce même Chapitre présente les limites de notre solution ainsi que nos directions futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

Dans la première partie de cette section, nous présenterons la revue de l'état actuel des outils d'instrumentation statique qui insèrent et compilent le code d'instrumentation directement avec le code source. Ensuite, nous présenterons les outils d'instrumentation statique qui ajoutent de l'espace dans le code pour que l'instruction d'instrumentation soit insérée durant l'exécution. La deuxième partie de la section traite des techniques d'instrumentation dynamique. Nous y discuterons des mécanismes et des solutions que chaque technique déploie pour résoudre les problèmes de l'instrumentation dynamique suivante :

— Problème d'insertion simultané :
  Le jeu d'instruction de l'architecture *x86* est de taille variable. Avec un tel encodage, les instructions les plus fréquentes peuvent avoir une taille plus petite. Cela diminue la taille de l'exécutable. Même si cela est un avantage pour la performance, il représente un inconvénient pour l'instrumentation dynamique. En effet, pour cette raison un branchement relatif de multiples octets peut chevaucher plusieurs instructions et changer leurs bordures. Si un saut vers une ancienne bordure existe, le flot de contrôle risque d'être corrompu. Un autre problème peut se produire lorsqu'un fil d'exécution remplace des instructions en simultané avec un autre fil préempté, interrompu ou en train d'exécuter l'ancienne bordure.

— Erratum de Intel sur la modification croisée du code (XMC) :
  Lorsqu'un coeur modifie des instructions qui chevauchent deux blocs de cache, d'autres coeurs peuvent observer un résultat partiellement écrit [3]. Dans ce cas, une faute de protection générale (GPF) sera levée et le programme sera arrêté pour ne pas corrompre le système. Intel recommande l'exécution d'une instruction de sérialisation (IRET, CPUID, etc..) par tous les coeurs avant d'exécuter le nouveau code [2]. Cependant, l'algorithme recommandé pour synchroniser l'exécution de l'instruction de sérialisation sur chaque coeur n'est pas applicable dans le cadre de l'instrumentation dynamique [4]. En effet, il exige la vérification d'un drapeau avant d'exécuter le code d'instrumentation. Puisque le code de vérification du drapeau doit être inséré avec le code d'instrumentation, une situation paradoxale se manifeste.

— L'exécution hors-ligne (OLX) :
  Les instructions remplacées par le code d'instrumentation doivent s'exécuter après l'instrumentation pour maintenir l'intégrité du flot de contrôle. Certains outils les copient dans un tampon pour les exécuter dans un autre endroit. Les instructions qui dépendent du PC doivent être déménagées en rectifiant leurs opérandes pour garantir une exécution équivalente. D'autres instructions comme les sauts relatifs ont une portée faible et ne peuvent plus

atteindre leur destination une fois déménagées.

Finalement, dans la dernière partie, nous présenterons les outils de traçage et leurs détails d'architecture.

2.0.1 TRACE_EVENT (noyau Linux)

TRACE_EVENT est la plus récente version de la macro qui définit un point de trace statique dans le noyau Linux [5, 6]. En lui passant la définition des paramètres ainsi que la définition de la fonction qui enregistre l'évènement dans le tampon en anneau du traceur, la macro crée le point de trace. De plus, la macro crée deux fonctions : la première vérifie si le point de trace est activé et la deuxième trace par défaut l'évènement. Le noyau ou un module (d'un traceur) chargé statiquement ou dynamiquement peut enregistrer des fonctions de rappel durant l'exécution. Lorsqu'un point de trace est déclenché, la fonction qui enregistre par défaut l'évènement dans le tampon en anneau du CPU est appelée. Ensuite, les fonctions de rappels sont appelées pour que les traceurs consomment l'évènement. Chaque point de trace crée des fichiers dans le système de fichier *tracefs* à sa création. Ces fichiers sont utilisés pour activer/désactiver le point de trace ou bien pour filtrer l'évènement en fonction d'une expression conditionnelle. Les évènements enregistrés dans le tampon en anneau peuvent être lus depuis un fichier dans le système de fichiers *tracefs*.

Les points de trace statiques définis par *TRACE_EVENT* sont les plus utilisés dans le noyau. En effet, l'insertion d'un point de trace est suffisante pour permettre à n'importe quel outil de traçage de s'attacher à lui. Ftrace, Uftrace, Perf et LTTng utilisent cette infrastructure de points de trace positionnés stratégiquement, pour être notifiés par les évènements du noyau.

2.0.2 Point de trace LTTng-UST

Pour tracer l'espace utilisateur, LTTng-UST permet à l'utilisateur de définir des points de trace statiques insérés et compilés avec le code source [7]. La définition du point de trace LTTng-UST ressemble à celle des TRACE_EVENT. Elle est faite à l'aide de la macro TRACEPOINT_EVENT() qui prend le nom du fournisseur, le nom du point de trace, et ses paramètres. La macro crée une classe et une instance du point de trace. La fonction *tracepoint()* appelle le point de trace et déclenche l'évènement qui lui correspond. La bibliothèque *liblttng-ust* peut, soit être liée au programme tracé durant la compilation, soit être préchargée avant l'exécution. Elle contient l'implémentation de la macro de définition et l'implémentation de la fonction d'appel. Une fonction (*tracepoint_enabled()* est fournie pour vérifier si un point de trace est activé, avant de faire des calculs lourds sur ses paramètres.

2.0.3 Userland Statically Defined Tracing (USDT/SDT)

USDT/SDT est un mécanisme d'instrumentation statique qui réserve de l'espace dans l'application dans le but d'insérer des points de trace. Il permet aux utilisateurs d'instrumenter un programme en connaissant seulement les opérations sémantiques de ce dernier [8]. En effet, l'utilisateur n'a pas besoin de connaître le nom des fonctions ni leurs adresses pour les instrumenter. Les points de trace USDT/SDT ont été conçus pour évoluer avec l'application, même lorsqu'elle renomme ou supprime des fonctions. Ils ont été implémentés pour DTrace, mais plusieurs outils de traçage ont commencé à les utiliser lorsqu'ils sont devenus populaires [9]. L'espace réservé par USDT/SDT peut être utilisé par n'importe quel outil pour insérer son instrumentation. Uftrace, SystemTap et LTTng les prennent en charge nativement. Il est aussi possible d'utiliser manuellement Ftrace avec Uprobe pour les consommer.

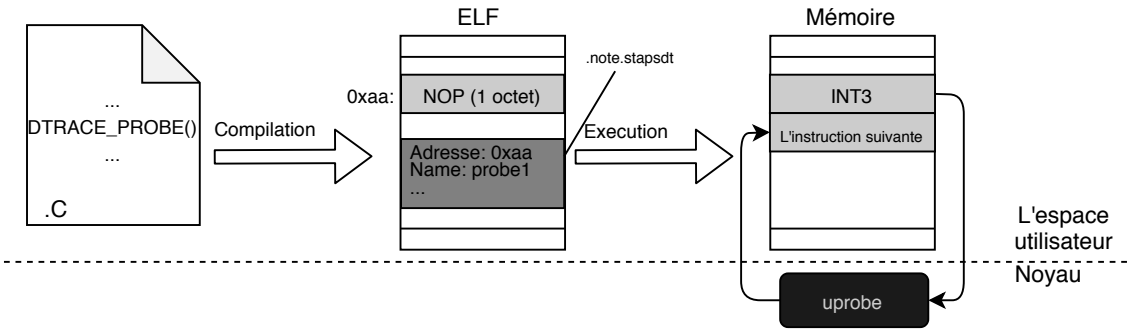


Figure 2.1 La consommation des points USDT a l'aide des points de trace uprobe.

La macro *DTRACE_PROBE()* de l'interface de programmation de *SystemTap* permet d'ajouter un point USDT/SDT dans le code source de l'application. Cela ajoutera un NOP de 1 octet durant la compilation (Figure 2.1). Une section de note ELF nommée *.note.stapsdt* contiendra toutes les informations (Nom du point, sa localisation, etc..) sur les points USDT/SDT. Ces informations seront utilisées par l'outil d'instrumentation pour trouver les NOP et les remplacer par une instruction d'interruption (*int3* dans *x86*). Il est possible d'utiliser un sémaphore pour savoir si le point est actif avant de l'appeler dans l'application [10]. Un sémaphore correspond à un seul point de trace. Son adresse est donc enregistrée dans la même note ELF que le point.

2.0.4 Assisté par le compilateur et installé à la volée

L'instrumentation assistée est du code ajouté statiquement par le compilateur qui sera modifié dynamiquement par une bibliothèque d'exécution [11]. En conséquence, comme toute instrumenta-

tion statique, les endroits d'instrumentation potentiels sont déjà prédéfinis, limitant ainsi la portée. Même si ces techniques ajoutent suffisamment d'espace pour insérer l'instruction d'instrumentation, certains d'entre eux ne préparent pas suffisament le terrain pour une insertion sécuritaire dans les environnements SMP. Cette section présente et discute certains outils d'instrumentation assistés par le compilateur.

XRay

Google XRay est un outil de traçage dynamique et d'instrumentation statique assisté par le compilateur pour instrumenter l'entrée et la sortie des fonctions d'un programme C/C++. Contrairement à la plupart des outils d'instrumentation assistée par le compilateur, XRay peut instrumenter un programme à la volée et sans interrompre son exécution. Le compilateur assigne un identifiant aux fonctions instrumentées pour pouvoir les trouver efficacement dans une structure de données lorsque le point de trace est atteint. Pour pouvoir utiliser XRay, l'exécutable doit être compilé avec l'option *-fxray-instrument* de Clang [12]. Cette option ajoute du code d'instrumentation avant le prologue et après l'épilogue (Figure 2.2). Le code à l'entrée est composé d'un branchement relatif de deux octets et d'une instruction NOP de neuf octets pour un total de onze octets. Le code à la sortie de la fonction n'est rien d'autre qu'une instruction NOP de dix octets. Le rôle du branchement relatif est d'effectuer un saut de neuf octets vers l'avant, sautant ainsi au-dessus des NOPs. Quant aux NOPs, ils seront remplacés durant l'insertion du point de trace par l'appel à la fonction de traçage de l'entrée.

```
00000000004241b0 <foo>:
  4241b0:        eb 09                   jmp    4241bb <foo+0xb>
  4241b2:        66 0f 1f 84 00 00 02    nopw   0x200(%rax,%rax,1)
  4241b9:        00 00
  4241bb:        55                      push   %rbp
  4241bc:        48 89 e5                mov    %rsp,%rbp
  4241bf:        48 83 ec 10             sub    $0x10,%rsp
                       .. foo body ..
  4241ed:        48 83 c4 10             add    $0x10,%rsp
  4241f1:        5d                      pop    %rbp
  4241f2:        c3                      retq
  4241f3:        2e 66 0f 1f 84 00 00    nopw   %cs:0x200(%rax,%rax,1)
  4241fa:        02 00 00
```

Figure 2.2 Le code d'une fonction dans un exécutable compilé avec -xray-finstrument. Les adresses encadrées en vert, rouge, bleu et marron représentent respectivement l'adresse de l'instrumentation de l'entrée, l'adresse du prologue, l'adresse de l'épilogue et l'adresse de l'instrumentation de la sortie.

La figure 2.3 montre les étapes de l'insertion du point de trace durant l'exécution. À l'exécution, XRay doit remplacer le code de l'entrée par l'instruction *mov $id, %r10d ; call __xray_FunctionEntryStub* qui sauvegarde l'identifiant de la fonction tracée avant d'appeler la fonction de traçage de l'entrée [13]. Pour traiter les problèmes de XMC, l'instruction *CALL* et les quatre derniers octets de l'instruction *MOV* sont d'abord écrits. À cette étape, ces instructions ne seront pas exécutées, et une sérialisation de tous les coeurs doit être faite. Ensuite, les deux premiers octets de l'instruction *MOV* remplacent le branchement relatif pour armer le point de trace. Cette dernière écriture doit se faire atomiquement au niveau de la mémoire centrale et de la mémoire cache du processeur. Pour cela, le compilateur prend soin d'aligner le branchement relatif sur la prochaine adresse qui est un multiple d'une puissance de deux (comme la taille d'un mot ou d'une ligne de cache). La raison de l'alignement est qu'une instruction de deux octets, telle que le branchement relatif, ne chevauchera jamais une ligne de cache avec une taille paire. Ceci permet donc une écriture sécuritaire.

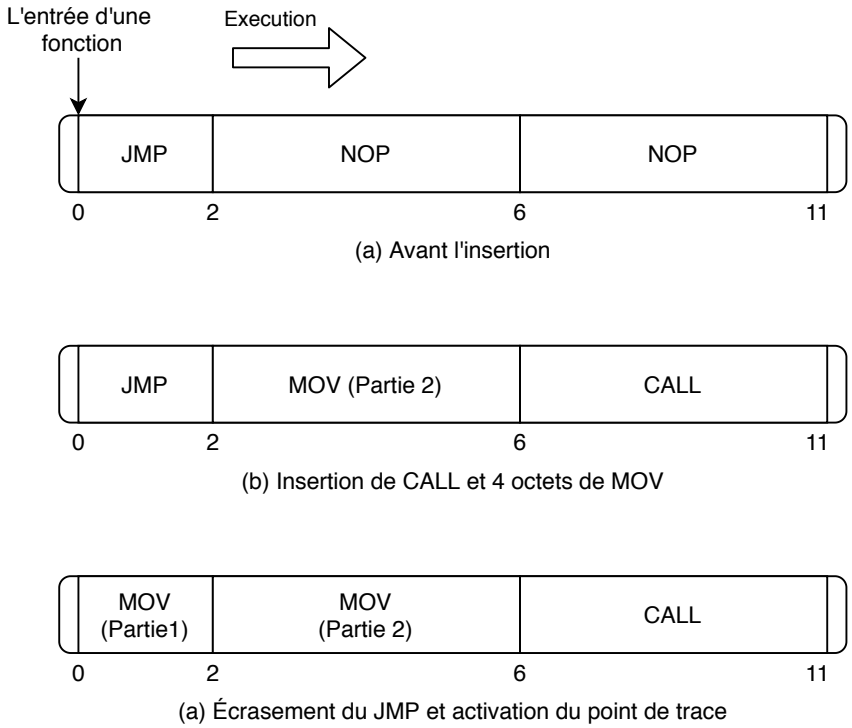


Figure 2.3 Les étapes d'insertion d'un point de trace Google XRay a l'entrée d'une fonction

L'insertion du point de trace dans la sortie se fait de la même façon que l'entrée. La seule différence est que l'instruction de retour de fonction (*RET*) est la dernière à être écrasée par le premier octet de l'instruction *MOV* pour armer le point de trace. Puisque la taille de *RET* est un octet, l'écriture est atomique.

Pour enlever un point de trace, XRay récrit seulement l'instruction de branchement pour l'entrée et l'instruction de retour de fonction pour la sortie. Parce qu'il ne peut pas garantir l'atomicité, XRay ne récrit pas les *NOPs*. Les fils d'exécution qui sont déjà en train d'exécuter le code du traçage ne peuvent pas y rentrer une fois sortis. Pour une raison d'optimisation, XRay remplace la première instruction des fonctions *__xray_FunctionEntryStub()* et *__xray_FunctionExitStub()* par une instruction de retour de fonction (*RET*).

Routine mcount/mcount-nop et cyg-profile

Pour pouvoir profiler un programme et compter le nombre de fois qu'une fonction est appelée, la technique mcount a été ajoutée dans le compilateur GCC et Clang [14]. Cette technique consiste à appeler la fonction *_mcount()* au début de toutes les fonctions du programme. L'appel est ajouté, de façon transparente au développeur, par le compilateur lorsque l'option *-pg* est spécifiée. Lors de l'utilisation de l'option *-pg* le compilateur utilise une version de profilage de la bibliothèque stan-

```
void foo ()
{
    cout << "Hello World" << endl;
}
```

(a) Fonction avant instrumentation

```
void foo ()
{
    __mcount();
    cout << "Hello World" << endl;
}
```

(b) Fonction après instrumentation

dard C qui contient une implémentation de la fonction *_mcount()*. En préchargeant une libraire qui contient un symbole *_mcount()* dans un processus, le symbole original sera écrasé, et le nouveau sera utilisé à la place. Il existe une autre façon pour utiliser un nouveau symbole et écraser l'ancien atomiquement durant l'exécution, sans interrompre le processus. Pour cela il faut interagir avec la table *PLT* du format *ELF* de telle manière à changer l'entrée qui contient l'adresse du symbole correspondant à *_mcount()* par l'adresse du nouveau symbole.

La fonction *_mcount()* ne prend aucun paramètre. Cependant, puisqu'elle est appelée à l'entrée, l'adresse de retour à la fonction tracée (enfant) ainsi que l'adresse de retour à la fonction appelante (parent) se trouvent dans la pile. Ces deux adresses peuvent être utilisées par *_mcount()* pour tracer le graphe d'appel de fonctions. En effet, la technique *mcount* est aussi utilisée comme point de trace. Pour tracer l'entrée et la sortie d'une fonction, lorsqu'une fonction parent appelle une fonction enfant, *_mcount()* est appelée à l'entrée. Ainsi, elle enregistre et écrase (dans la pile) l'adresse de retour au parent avec l'adresse d'une fonction de traçage de sortie. Une fois que la fonction enfant finit son exécution, elle retourne à la fonction de traçage de sortie qui remet l'adresse de retour originale sur la pile, avant de faire un deuxième retour pour continuer la fonction parent. Même si cela marche bien dans la plupart des fonctions C/C++, certains peuvent manipuler l'adresse de retour après l'appel à *_mcount()*. Cela peut causer des interférences et corrompre l'exécution du programme. Ainsi, l'option *-finstrument-functions* peut être utilisée à la compilation pour activer *cyg-profile*. Cette technique ajoute un appel à *cyg_profile_func_enter()* à l'entrée et l'appel *cyg_profile_func_exit()* à la sortie d'une fonction. Les deux appels prennent l'adresse du parent et l'adresse de l'enfant comme paramètre. Contrairement à mcount, le compilateur prend soin de l'initialisation des paramètres avant l'appel. L'interception de l'appel à *cyg-profile* se fait de la même façon que *mcount*.

Il n'est pas possible de contrôler et choisir les fonctions instrumentées par le compilateur. Effectivement, le compilateur ajoute l'appel *_mcount()* à toutes les fonctions du programme. Une technique utilisée par certains outils pour enlever l'appel *_mcount()* de certaines fonctions qui ne veulent pas tracer est de le remplacer par l'instruction *nopl 0(%eax, %eax, 1)*. Si le nombre de fonctions à ne pas tracer est largement supérieur à celles tracées, il est préférable de compiler le programme avec l'option *-pg -mfentry -mnop-mcount*. Cette option ajoute l'instruction *nopl 0(%eax, %eax, 1)* à la

place de l'appel à *_mcount()*. La libraire de traçage doit remplacer le NOP de cinq octets par un appel à *_mcount()*. Contrairement aux autres techniques assistées par le compilateur, durant l'exécution la bibliothèque de traçage doit implémenter un protocole approprié pour assurer l'atomicité du remplacement des NOPs.

2.1 Instrumentation dynamique

Les techniques d'instrumentation dynamique sont ajoutées dans un binaire après sa compilation. Contrairement à l'instrumentation statique, celle dynamique fait face au défi de manque d'espace où insérer l'instrumentation. Pour résoudre ce problème, certains outils remplacent les instructions originales par l'instruction d'instrumentation. D'autres outils décompilent le code pour insérer l'instrumentation avant de le recompiler et l'exécuter. Les complications introduites par l'existence de multiples fils d'exécution ne sont pas présentes lorsque l'instrumentation est ajoutée dans un binaire sur disque (avant son exécution). L'insertion durant le temps d'exécution se fait dans le code chargé dans la mémoire centrale. Ce code partagé par les différents fils d'exécution du processus doit être modifié avec prudence. Dans cette section nous présenterons d'abord les points de trace qui remplacent l'instruction originale par une instruction d'instrumentation. Ensuite, nous présenterons les outils qui instrumentent un programme en le modifiant partiellement ou complètement.

2.1.1 Point de trace par injection de code

Kprobes et Kretprobe

Kprobes est un outil d'instrumentation intégré directement dans le noyau Linux et permet d'installer des points de trace dynamiques dans presque n'importe quel endroit dans le noyau sans perturbation [15]. À la création, l'utilisateur passe l'adresse où le point de trace sera inséré et quatre fonctions de rappel : une fonction de prétraitement, une fonction de post-traitement, une fonction de gestion de faute et une fonction de gestion d'interruption. Seule la fonction de prétraitement est obligatoire. La fonction de gestion d'interruption est réservée pour l'usage interne par Kprobes. Elle est utilisée pour gérer les interruptions levées durant l'exécution du point de trace. La fonction de prétraitement est appelée juste après que le point de trace soit atteint et que le contexte du fil d'exécution soit sauvé. Les instructions originales remplacées par l'instrumentation sont enregistrées dans un tampon OLX pour être exécutées hors-ligne. Si elles soulèvent une faute, la fonction de gestion de faute est appelée. Après l'exécution des instructions originales, une interruption logicielle à la fin du tampon OLX appelle la fonction de post-traitement. Pour avoir accès à l'état du système lorsqu'un Kprobe est frappé, l'adresse du probe ainsi que les registres du fil sont envoyés en tant que paramètres aux fonctions d'appel. La fonction de gestion de faute reçoit un paramètre

supplémentaire qui est le numéro de la faute.

Pour créer un point de trace sans insertion de module, si le noyau est construit avec l'option *CONFIG_KPROBE_EVENTS=y*, le fichier */sys/kernel/debug/tracing/kprobe_events* dans le système de fichier tracefs sert à ajouter des points de trace Kprobe. Si l'utilisateur ne connaît pas l'adresse de l'instruction à instrumenter, il peut utiliser le nom du symbole où elle se trouve. Il est important de noter que certains symboles peuvent produire des situations de récursion si instrumentées. Certains symboles peuvent même introduire des interblocages. Les fonctions non instrumentables sont donc ajoutées dans une liste noire à l'aide de la macro NOKPROBE_SYMBOL. Pour activer ou désactiver un point de trace Kprobe, le fichier */sys/kernel/debug/tracing/events/kprobes/event/enabled* pourrait être utilisé. Tout comme *TRACE_EVENT*, l'utilisateur peut insérer un module dans le noyau pour enregistrer un Kprobe à l'aide de la fonction *register_kprobe(struct kprobe *p)* [1].

Kprobes fait une copie de l'instruction remplacée par le point de trace pour qu'elle puisse s'exécuter dans un tampon OLX. Ensuite, il remplace le premier octet de l'instruction instrumentée par une instruction d'interruption logicielle. À cause de la latence de l'interruption logicielle, Kprobes essaye d'optimiser un point de trace après sa création à l'aide d'une instruction de branchement relative [16]. Kprobes vérifie qu'aucun saut au sein de la fonction instrumenté vers la région modifiée (qui contient les instructions chevauchées) par le branchement relatif n'existe. Dans le cas contraire, l'optimisation est annulée parce que la modification des bordures des instructions dans cette région peut causer des problèmes si jamais le branchement est pris. Ensuite, il verrouille la région à modifier à l'aide de l'instruction d'interruption logicielle insérée dans la tête de la première instruction de la région. Cela garantit que, dorénavant, aucun processeur n'exécutera les instructions de cette région. À l'aide de RCU dans le noyau non préemptif, la fonction *synchronize_sched()* attend que tous les coeurs du système appellent l'ordonnanceur. Cela garantit que tous les coeurs exécutent le code de la synchronisation et qu'ils sont donc à l'extérieur de la région à modifier. L'insertion du branchement peut se faire désormais de façon sécuritaire.
À cause de la taille du code du noyau, un sauf relatif avec une portée de 4 Go peut toujours atteindre la fonction d'entrée dans la section .text. Cette fonction contient suffisamment d'espace pour brancher plus loin si nécessaire.
Un point de trace Kprobe peut être accéléré en remplaçant l'interruption logicielle à la fin du tampon OLX par un saut absolu [17]. Ainsi le point de trace devient plus rapide, mais perd sa capacité d'appeler une fonction de post-traitement.

La fonction *register_kprobe()* doit enregistrer la structure Kprobe dans une table de hachage. Lorsque le point de trace est atteint, l'adresse du Kprobe est trouvée dans la table de hachage et envoyée à la

1. Implémenté dans linux/kernel/kprobes.c et exporté à l'aide de EXPORT_SYMBOL_GPL()

fonction d'appel. Puisque plusieurs lectures et écritures à cette table peuvent se faire en simultané, Kprobes a utilisé un *spinlock* dans une version ancienne. Malheureusement, une grande perte de performance a été remarquée lorsque plusieurs Kprobe sont atteints en simultané. Un prototype qui utilise un *rwlock* a été essayé, mais il a été écarté plus tard pour être remplacé par RCU. Avec RCU, les lectures (Kprobe atteint) peuvent se faire sans prise de verrou. Avec RCU, la libération de la structure du point de trace ne se fait qu'une fois que les lecteurs finissent de l'utiliser. La dernière version qui utilise RCU offre le même surcoût que *spinlock* avec 2 fils d'exécution. Cependant, RCU offre une meilleure évolutivité avec le nombre de fils d'exécution.

Kretprobe est un point de trace bâti sur Kprobe et qui permet de tracer l'entrée et la sortie de la fonction [18]. Pour instrumenter l'entrée et la sortie d'une fonction, *register_kretprobe()* insère un Kprobe normal à l'entrée de la fonction. Lorsque la fonction instrumentée est appelée et le point Kprobe atteint, l'adresse de retour se trouvant dans la pile d'appels est copiée et remplacée par l'adresse du Kretprobe [19]. Lorsque la fonction instrumentée finit son exécution, lorsqu'elle appelle l'instruction *ret* dans l'architecture *x86*, le Kretprobe est appelé. Les fonctions de rappel sont appelées et l'adresse de retour précédemment copiée est restaurée. Le programme peut donc continuer son exécution normale.

Uprobe et Uretprobe

Uprobe est un point de trace implémenté complètement dans le noyau Linux et qui permet d'instrumenter le code dans l'espace utilisateur [20]. Il utilise une instruction d'interruption logicielle pour l'instrumentation. Pour insérer un point de trace uprobe, l'utilisateur doit choisir un exécutable (cela peut être une bibliothèque partagée) sur disque et l'adresse de l'instruction dans cet exécutable. Uprobe ajoute donc un point de trace dans tous les processus qui partagent les pages virtuelles de l'exécutable instrumenté. Après avoir enregistré l'évènement dans le tampon de traçage, l'instruction originale est exécutée. Dans une ancienne version de uprobe, un tampon OLX dans la mémoire virtuelle de chaque processus instrumenté est alloué. Pour ne pas être intrusif et limiter la consommation de mémoire virtuelle de l'espace utilisateur, une émulation de l'instruction par l'espace noyau dans le contexte utilisateur est effectuée. Cela évite l'allocation d'un tampon OLX dans chaque processus. Pour une raison de sécurité, uprobe ne permet pas à l'utilisateur de choisir sa propre fonction de rappel qui se trouve dans l'espace utilisateur. Cependant, une fonction de rappel peut être choisie en insérant un module dans le noyau et en créant un point de trace Uprobe en utilisant la fonction *uprobe_register()* [2]. À sa création, le point de trace est ajouté dans un arbre binaire rouge-noir. Lorsque le point de trace est rencontré, l'adresse de l'instruction d'interruption est utilisée pour le chercher dans l'arbre. Pour protéger l'arbre lorsque plusieurs insertions

2. implémenté dans *linux/kernel/events/uprobe.c* et exporté à l'aide de *EXPORT_SYMBOL_GPL()*

et exécutions de points de trace sont faites, un *spinlock* par Uprobe est déployé. Malgré le fait que RCU offre un faible surcoût et une meilleure évolutivité, il ne peut pas remplacer le *spinlock*. En effet, de la manière dont il est implémenté actuellement dans le noyau, RCU synchronisera tous les Uprobe du système. Pour la désallocation de point de trace, un compteur de référence est utilisé. Atomiquement incrémenté et décrémenté à chaque fois qu'il est atteint et à chaque fois qu'il n'est plus utilisé, le rôle du compteur est de savoir si un fil d'exécution exécute actuellement le point de trace. Lors de l'appel à *uprobe_unregister()* le point de trace est d'abord désarmé en remplaçant l'instruction d'interruption par l'instruction originale. Si le point de trace n'est pas en cours d'exécution, ses données peuvent être désallouées. Sinon, la tâche de suppression est ajoutée dans une file pour ressayer plus tard. Pour instrumenter l'entrée et la sortie des fonctions, Uretprobe a été bâti sur Uprobe et fonctionne comme Kretprobe (écrase l'adresse de retour dans la pile d'appels).

Comme Kprobes, le système de fichier *tracefs* permet d'insérer et d'activer des points de trace uprobe [3] [4]. Une fois le traçage fini, la trace est dans le fichier */sys/kernel/debug/tracing/trace* et peut être récupérée avec une simple lecture.

GDB : les points de trace d'interruption et les points de trace rapide

Les points de trace d'interruption utilisés dans GDB se basent sur l'instruction d'interruption logicielle [21]. Ils ont été créés pour pouvoir instrumenter n'importe quelle instruction d'un programme dans l'espace utilisateur. Puisqu'ils utilisent une interruption, les points de trace d'interruption partagent le même gestionnaire d'interruption que les points d'arrêt. Cela dit, lorsqu'une interruption logicielle est levée, le noyau vérifie d'abord si ce n'est pas un point de trace ou un point d'arrêt du noyau avant de commencer une série d'appels qui finissent par le gestionnaire d'interruption défini par l'espace utilisateur. GDB reçoit l'interruption du programme tracé à l'aide de ptrace et vérifie si c'est un point de trace ou un point d'arrêt avant d'effectuer le traçage. Cela ajoute un très grand surcoût, comparé à uprobe qui peut traiter l'évènement et le tracer sans retourner à l'espace utilisateur.

Les points de trace rapides de GDB sont, comme leur nom le mentionne, plus rapides que les points de trace d'interruption [21]. Ils utilisent des branchements relatifs et plus précisément une instruction de saut relative. L'utilisation de ce point de trace est limitée à l'instrumentation des instructions de taille supérieure à cinq octets, pour éviter la modification des frontières d'instructions. Cette limitation réduit considérablement les endroits où le point de trace peut être inséré. En effet, d'après une analyse statistique de 3 exécutables, presque 75% des instructions ont une taille inférieure à cinq octets [22]. Pour éviter les problèmes de XMC, l'écriture du branchement relatif se fait durant

3. /sys/kernel/debug/tracing/uprobe_events pour l'insertion
4. /sys/kernel/debug/tracing/events/uprobes/event/enabled pour l'activation

l'arrêt du processus. Contrairement à l'interruption logicielle, ptrace ne notifie pas GDB lorsqu'un point de trace rapide est rencontré. Il injecte alors une bibliothèque IPA (In-Process-Agent) dans l'espace d'adressage du processus tracé pour l'instrumenter et le tracer. L'IPA est une bibliothèque dynamique qui, une fois chargée, alloue une structure pour les trampolines ciblés par les sauts relatifs, pour enregistrer le contexte du fil d'exécution dans la pile, avant de rebondir vers la fonction de traçage. Dans l'architecture *x86_32* [5], peu importe où l'IPA se situe, un saut depuis le code statique du programme peut toujours atterrir sur le trampoline. Par contre, dans l'architecture *x86_64*, il faut s'assurer que le code statique du programme soit proche de l'IPA. Les points de trace rapides peuvent s'exécuter en simultané. Pour protéger le tampon de traçage, l'IPA écrit une prise et une libération d'un verrou exclusif dans le trampoline. Cela pénalise grandement la performance des processus avec plusieurs fils d'exécution. Si un point de trace rapide est inséré dans un gestionnaire de signal, et qu'un fil d'exécution reçoit un signal alors qu'il est déjà en train d'exécuter un point de trace, une situation d'interblocage peut se produire.

Pour insérer un point de trace, il faut interagir avec l'interface de GDB. La commande *trace* et *ftrace* prennent l'adresse de l'instruction et l'instrumentent respectivement avec un point de trace d'interruption et un point de trace rapide. Pour instrumenter un programme, le processus cible doit être arrêté par un point d'arrêt ou autre. Il n'est pas possible de choisir une fonction de rappel pour le point de trace à travers l'interface de GDB.

Dyntrace et Liteinst

Dyntrace est un outil d'instrumentation dynamique pour l'espace utilisateur. La technique d'instrumentation fait usage d'un branchement relatif qui embarque des instructions d'interruption à la tête de chaque instruction chevauchée. L'idée est que lorsqu'un fil d'exécution prend un saut qui cible une instruction remplacée, l'interruption fera le transfert vers l'instruction originale [23]. La figure 2.5 montre une situation où deux instructions (1 et 2) sont ciblées par des branchements. Une écriture normale d'un saut au point d'insertion 2.5b changera les bordures d'instructions et invalidera la cible des branchements existants. 2.5c montre le résultat d'instrumentation par Dyntrace. Les instructions d'interruption *INT3* sont fixées dans le déplacement et redirigeront les fils d'exécutions lorsqu'ils branchent vers cible 1 ou cible 2. L'instruction originale correspondante à l'endroit où l'interruption a eu lieu sera donc exécutée correctement dans le tampon OLX. En fixant une ou plusieurs valeurs dans le déplacement du saut, Dyntrace limite considérablement le nombre de régions où la structure du point de trace peut être allouée. Cela réduit le nombre de points de trace qui peuvent être insérés en même temps dans un programme. De plus, parce qu'un mapping dans l'espace d'adressage peut être fait pour chaque point de trace inséré, et parce que le dépla-

5. La taille totale de l'espace d'adressage virtuel d'un processus est 4 Go

cement doit respecter la contrainte des positionnements des interruptions, Dyntrace crée une forte fragmentation externe de la mémoire. Par défaut, le fichier */proc/sys/vm/max_map_count* contient la valeur maximale du nombre de mapping qu'un processus peut faire. Cette valeur est limitée à 65536. Cela réduit davantage le nombre de points de trace installés en même temps et peut même faire qu'un programme manque de mémoire.
Dyntrace insère le saut entier (avec les interruptions imbriquées) dans une seule écriture atomique (instruction *CMPXCHG8B/16B* avec préfixe *LOCK*). Même si cette opération est atomique au niveau de la mémoire centrale, elle risque de ne pas l'être dans le cas où deux lignes de cache sont chevauchées par le saut. Cette technique doit donc être combinée avec un protocole de XMC adéquat pour les environnements SMP.

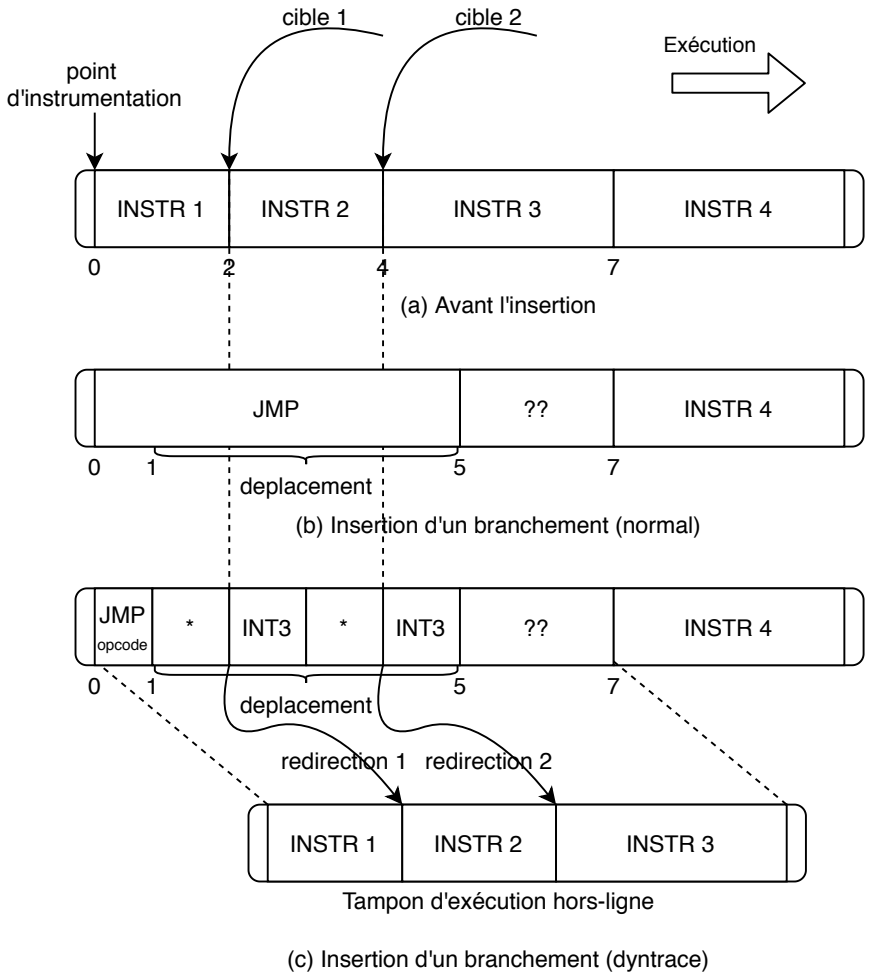


Figure 2.5 L'insertion d'un point de trace dyntrace. Les *INT3* sont fixés dans le déplacement. Le symbole "$\star$" est un octet libre qui peut prendre n'importe quelle valeur. Le symbole "??" représente un ou plusieurs octets inchangés durant l'instrumentation.

Liteinst est un outil d'instrumentation dynamique pour l'espace utilisateur qui partage le même concept que Dyntrace [4]. Au lieu de se limiter aux instructions d'interruption, Liteinst combine des instructions illégales pour avoir plusieurs choix concernant l'endroit où mettre la structure du point de trace. À l'aide d'un algorithme heuristique, ces combinaisons leur permettent de grouper les structures dans une seule région de mémoire. À cause de cela, leur consommation de mémoire est plus faible. Les combinaisons leur permettent aussi d'éviter les déplacements négatifs lorsque *INT3 (0xcc)* se trouve dans l'octet le plus significatif. Un déplacement négatif est problématique lorsque l'exécutable est chargé dans une adresse où rien ne peut être mappé avant. Liteinst utilise un algorithme d'insertion durant l'exécution qu'ils ont nommée *WordPatch++*. L'algorithme tire avantage du fait que les changements du code seront observés par les autres processeurs dans un nombre de cycles maximal T_{max} [24]. En effet, si une instruction chevauche deux lignes de cache, *WordPatch++* insère des points d'interruption au point d'instrumentation, verrouillant ainsi la région d'instrumentation pour qu'aucun fil d'exécution ne l'exécute. Ensuite, une attente de T_{max} cycles est effectuée avant de continuer l'insertion.

2.1.2 Modification (complète ou partielle) du programme instrumenté

Valgrind

Valgrind est une plateforme programmable qui modifie un programme dynamiquement dans le but d'insérer du code d'instrumentation ou du code de profilage. Il supervise l'exécution du programme à l'aide d'une technique de traduction dynamique du binaire (JIT) [25]. Ce rôle est notamment accompli par le coeur de Valgrind. Les *skin* peuvent utiliser l'interface du coeur pour implémenter des outils de profilage, de débogage ou de traçage.
Le coeur est une bibliothèque (*valgrind.so*) préchargée avant le programme supervisé. Le contrôle n'atteint jamais le programme client [26]. En effet, le code de Valgrind prend contrôle du CPU une fois la bibliothèque chargée. Valgrind crée un CPU virtuel qui supervise et pilote l'exécution du code traduit pas à pas. Ensuite, il décompile chaque bloc de base (Encodé en *x86*) qu'il rencontre en représentation intermédiaire IR (Encodé en *Ucode*). L'IR est optimisé et l'instrumentation des *skin* est ajoutée. L'IR est retransformé en bloc d'instruction *x86* avant d'être exécuté par le CPU physique. À la fin de l'exécution du bloc présent, le contrôle retourne à Valgrind et le processus recommence pour le prochain bloc. Les blocs de bases rencontrés sont ajoutés dans un cache pour accélérer la traduction si jamais ils doivent être exécutés de nouveau.
Les appels système sont interceptés par Valgrind parce qu'ils doivent s'exécuter dans le contexte du CPU réel. Les signaux sont aussi interceptés parce qu'ils sont envoyés par le noyau vers le CPU réel. Valgrind reçoit le signal, et l'envoi au programme client après qu'un certain nombre de blocs de base soit exécuté. Valgrind utilise une bibliothèque modifiée de *libpthread.so* pour que tous les

fils d'exécution créés par le programme client soient sérialisés en un seul fil d'exécution POSIX. Valgrind implémente donc une bibliothèque de fils d'exécution d'espace utilisateur qui s'occupe de les ordonnancer. L'approche de Valgrind offre un bon contrôle du programme client. Cependant, elle impose un très grand surcoût (5 à 100 fois plus lent) et ne peut donc pas être déployée dans un environnement de production [27].

Pin

Pin est une plateforme de modification dynamique de binaires [28]. La machine virtuelle de PIN se compose de quatre modules.

- Le compilateur JIT : le compilateur JIT compile le code du programme client à la volée et exécute le code généré à la place du code original. Contrairement à Valgrind, Pin fait une compilation ISA-ISA (x86-x86, ARM-ARM, etc.). Il n'utilise pas de représentation intermédiaire (IR).
- Le cache du code : Le code généré par le JIT est enregistré dans un cache de code pour une future exécution. Un changement de contexte est fait à chaque passage du code cache à la machine virtuelle et vice-versa.
- L'unité d'émulation : Tout comme Valgrind, les appels systèmes doivent être exécutés sur le CPU réel et non pas dans la VM. Ce module s'occupe de les intercepter et les émuler.
- Le répartiteur : Ce module repartit l'exécution à la *trace* (un ou plusieurs blocs de base en séquence) suivant ce qui se trouve dans le code de cache. Si la *trace* ne se trouve pas dans le cache, le JIT la compile.

Pour accélérer l'exécution, Pin essaye de relier les *trace* directement entre eux. Pour relier les *trace* qui se terminent avec des branchements indirects, Pin essaye de prédire dynamiquement la cible du branchement. Dans le cas où la prédiction échoue, une recherche de la cible est faite dans une table de hachage. Si aucune entrée n'est trouvée, le contrôle retourne à la machine virtuelle. Ce mécanisme qui relie les blocs de base est aussi utilisé par DynamoRIO [28]. Il réduit le nombre de changements de contexte de l'application vers la VM. Pin met en oeuvre des techniques qui le caractérisent des autres outils d'instrumentation du même type. Ces techniques le rendent 3.3 fois plus rapide que Valgrind et 2 fois plus rapide que DynamoRIO. L'insertion et l'optimisation *en ligne* de l'instrumentation sont une des techniques pour éviter des changements de contexte additionnels. Un autre avantage d'utiliser Pin est qu'il peut s'attacher à un programme durant l'exécution et prendre le contrôle à l'aide de *ptrace*. Le détachement enlève l'instrumentation et redonne le contrôle à l'application client exécutant ainsi le code original.

Dyninst

Dyninst offre une interface de programmation simple et écrite en C++ pour modifier dynamiquement un binaire [29]. L'utilisateur de l'interface peut insérer des *Snippet* dans des *Points* durant l'exécution. Un *Snippet* est une abstraction du code d'instrumentation inséré dans un *Point*. Un *Point* est un emplacement dans le programme. Il peut être l'entrée/sortie d'un bloc de base, l'entrée/sortie d'une fonction ou une adresse arbitraire dans un programme. Pour plus de flexibilité, Dyninst permet à l'utilisateur de choisir un *Snippet* écrit sous différentes syntaxes (code binaire, arbre syntaxique abstrait de Dyninst ou bien du code écrit en langage qui ressemble à C nommé DynC). L'utilisateur peut aussi choisir une fonction de rappel déjà compilée avec le programme comme *Snippet* [30].
Dyninst est une approche d'instrumentation dynamique basée sur la modification du GFC [31]. À l'aide de l'interface de programmation PatchAPI, Dyninst peut modifier le binaire d'un exécutable ou d'une bibliothèque en modifiant le GFC du programme. Dans Dyninst, un GFC est une représentation de la structure élémentaire qui relie les différentes composantes d'un programme entre elles : les fonctions, les boucles, les blocs de base, etc.. Pour instrumenter, un programme, du code (*Snippet*) est inséré entre les sommets du GFC, créant ainsi un GFC augmenté. Dyninst préfère utiliser un branchement relatif de cinq octets comme arrêtes pour connecter les nouveaux sommets ajoutés dans le GFC. Pour instrumenter l'entrée et la sortie d'une fonction, par exemple, Dyninst peut déménager toute la fonction vers une nouvelle position. Un branchement relatif vers la nouvelle fonction sera inséré à l'adresse originale de la fonction. Cependant, si par exemple la taille de la fonction est inférieure à cinq octets (ce qui est très rare), le branchement peut chevaucher une autre fonction ou un autre bloc de base non instrumenté. Dans ce cas, Dyninst utilisera une instruction d'interruption de un octet pour le transfert vers la nouvelle fonction. Avec l'analyse du GFC, Dyninst atteint un très grand taux de réussite d'instrumentation avec un branchement rapide. Cela vient bien sûr avec un grand surcoût d'analyse et d'écriture du GFC dans l'espace d'adresse virtuelle du processus cible (*mutatee*). Un temps d'analyse d'environ six seconds avec 64 coeurs utilisés a été remarqué. De plus, durant ce temps le *mutatee* est complètement arrêté.
Pour écrire le code binaire du GFC augmenté dans le *mutatee* après son exécution, *ptrace* est utilisé pour arrêter tous les fils d'exécution. Ensuite, le contexte de chaque fils d'exécution est inspecté pour transférer les fils d'exécution à l'extérieur de la région d'instrumentation. L'écriture est effectuée de façon sécuritaire avant de débloquer le *mutatee* et reprendre l'exécution.

Ksplice

Ksplice est un outil de modification dynamique du code du noyau [32]. Il peut installer des mises à jour dans le code du noyau sans recompilation et sans redémarrage. Il déménage entièrement des

fonctions dans la mémoire pour exécuter de nouvelles fonctions à la place. Pour ce faire, l'outil compile le code source avant et après la modification. Il compare ensuite les deux objets (ELF) générés pour trouver les fonctions affectées par la modification. Certaines fonctions peuvent être inchangées par la mise à jour, mais leurs compilations (avant et après le correctif) donnent un résultat différent. Par exemple, le compilateur peut décider d'ajouter des instructions NOP pour aligner du code dans une fonction. Ksplice peut détecter ces situations et ne pas les considérer comme un changement. De plus Ksplice essaye de faire la compilation de telle façon à minimiser ces différences indésirables. Ensuite, il interrompt l'exécution du système pour quelques microsecondes pour appliquer les modifications. Le coeur qui effectue la modification installe un branchement à l'adresse du début de chaque fonction affectée vers la nouvelle version de la fonction. Finalement, le système reprend l'exécution.
Les mises à jour qui ne modifient pas la sémantique des structures de données peuvent être appliquées automatiquement et sans l'intervention du programmeur pour les adapter à Ksplice. Le reste des mises à jour doit être modifié pour ajouter des lignes de code (17 lignes en moyennes) qui seront exécutées durant le processus de modification de code pour adapter l'ancienne structure à la nouvelle sémantique.

2.2 Outils de traçage

Pour récolter efficacement la trace de l'exécution d'un programme cible, les outils de traçages utilisent des points de trace rapides et performants qui peuvent enregistrer des évènements dans la trace. Dans cette section nous présenterons les différents outils de traçage existants.

2.2.1 GDB

GDB est principalement un outil de débogage, qui implémente un module d'instrumentation et de traçage dynamique [33]. Il utilise les points de trace d'interruption et les points de trace rapide expliqués précédemment. Pour tracer un programme, il faut utiliser l'architecture client-serveur de GDB. Le serveur *gdbserver* peut s'attacher ou créer un processus à tracer. Pour faire le traçage, la bibliothèque IPA doit être préchargée ou bien injectée par *gdbserver*. Cette bibliothèque reçoit les commandes du serveur(p. ex., installation des points de trace, démarrage du traçage, etc.) et les exécute. La même bibliothèque fait aussi le traçage et accumule les évènements dans un tampon local avant de les envoyer au serveur. Le client gdb peut se connecter au serveur en utilisant le type de connexion choisi par ce dernier (ssh, tcp, etc). À l'aide d'une interface de ligne de commande, l'utilisateur peut interagir avec le client pour envoyer des requêtes au serveur. Sous commande de l'utilisateur, le client reçoit les traces depuis le serveur pour les visualiser. Pour déboguer un programme, le client peut être utilisé directement sans le serveur.

GDB n'est pas très efficace pour faire du traçage. En effet, il récupère des informations prédéfinies sur le contexte d'exécution. Il n'offre pas aux utilisateurs la flexibilité de choisir leur propre fonction de rappel pour récolter les informations dont ils ont besoin. De plus, ses points de trace sont très lents ce qui fait qu'il ne peut pas être utilisé dans l'environnement de production. Même dans l'environnement de développement, il n'est pas très fiable pour cette tâche puisqu'il peut facilement cacher des problèmes de performance à cause de la latence de l'instrumentation.

2.2.2 DTrace

DTrace est un outil de traçage dynamique pour tracer le noyau Linux [34]. Il peut aussi instrumenter et tracer l'espace utilisateur. Dtrace permet à l'utilisateur de fournir du code dans un langage de script D qui ressemble à C. Ce code sera appelé (à chaque rencontre du point de trace) par DTrace pour filtrer les évènements à enregistrer et choisir seulement l'information pertinente à l'utilisateur. Dans le noyau, les modules qui exportent des points de trace dans l'application sont nommés des *provider*. DTrace se connecte à eux pour récupérer les évènements enregistrés par les points de trace. Dans l'espace utilisateur, la bibliothèque *libdtrace* chargée dans l'espace utilisateur du processus client s'occupe d'instrumenter et d'envoyer les évènements tracés au *démon* (module dans le noyau) de DTrace. DTrace utilise les points USDT/SDT pour les remplacer par des instructions d'interruption, ce qui le rend moins performant comparé aux techniques d'instrumentation par branchement relatif.
Pour enregistrer localement les évènements, le *démon* de DTrace utilise un tampon par consommateur par CPU. Deux tampons sont créés : un est actif et l'autre est de rechange [35]. Lorsque *libdtrace* demande de consommer le tampon actif, le *démon* demande au CPU correspondant de changer vers le tampon de rechange. Ensuite, le tampon de rechange (ancien tampon actif) est copié dans l'espace utilisateur. Les tampons sont par défaut non circulaires. Si jamais un tampon est plein, les données sont perdues. L'utilisateur peut décider de choisir un tampon circulaire à la place.

2.2.3 SystemTap

L'outil de traçage SystemTap peut tracer les deux domaines, espace utilisateur et noyau [36]. L'outil peut soit afficher la trace à la volée ou bien l'enregistrer sur disque pour la rejouer dans le futur. Pour insérer de l'instrumentation, un script en langage propre à SystemTap doit définir le point de trace ainsi que la fonction de rappel. Ce script sera compilé en langage C avant d'être compilé en un module noyau. Le module doit être compilé et chargé (avec un simple *insmod*) dans le noyau à chaque instrumentation, ce qui peut ajouter un surcoût à l'instrumentation. Le module généré instrumente le programme/noyau et communique avec SystemTap pour le traçage. Il peut se servir des points de trace statiques (*TRACE_EVENT*) et des points de trace dynamiques Kprobe pour instru-

menter le noyau. Pour instrumenter l'espace utilisateur, Uprobe, Dyninst et USDT sont disponibles. Dans le traçage de l'espace utilisateur, à chaque évènement déclenché, SystemTap fait un changement de contexte vers le noyau pour l'écriture dans la trace. Comparé à un outil de traçage qui fait la mise en mémoire tampon dans l'espace utilisateur (p. ex., LTTng et Uftrace), SystemTap peut être beaucoup plus lent (environ 289 fois lent) [37]. Un tampon par CPU est utilisé pour enregistrer les évènements. Pour lire la trace depuis le noyau, SystemTap peut déployer deux mécanismes. Le premier copie le contenu du tampon vers l'espace utilisateur, tandis que l'autre se base sur l'interface *socket* pour diffuser les données de la trace.

2.2.4 Strace

Strace est un outil pour tracer l'interaction entre l'espace utilisateur et le noyau [38]. Strace intercepte et enregistre deux types d'évènements, les appels systèmes faits par le processus et les signaux reçus par lui. Pour réussir sa tâche, Strace utilise l'appel système *ptrace* intérieurement [39]. À l'aide de la commande *PTRACE_ATTACH* et *PTRACE_SYSCALL*, Strace peut s'attacher à un processus et recevoir une notification à chaque fois que le processus tracé entre ou sort d'un appel système. Le processus tracé est arrêté avec un *SIGTRAP* jusqu'à ce que Strace examine la pile pour récupérer les paramètres ou la valeur de retour de l'appel système. Strace ralentit considérablement le processus tracé à cause des interruptions de *ptrace*. Des outils comme Perf ou Uftrace peuvent intercepter les appels système avec moins d'impact de performance (environ 60 fois plus rapide que Strace).

2.2.5 eBPF

eBPF ou *extended Berkeley Packet Filter* est un outil qui fait l'insertion et l'exécution du code dans le noyau [40]. BPF est la version originale qui permet d'analyser le contenu des paquets, de faire des opérations arithmétiques et des comparaisons sur eux ou bien de les filtrer [41]. eBPF est une extension de l'outil qui peut faire tout ce que BPF fait, en plus de faire du traçage. À l'aide des points de trace Uprobe, Kprobe, TRACE_EVENT et perf_events, l'utilisateur peut attacher du code eBPF qui sera appelé pour tracer le système et filtrer les données non pertinentes. En réalité, eBPF est une machine virtuelle implémentée dans le noyau Linux qui interprète le langage eBPF vers du langage machine. Dans certains programmes, eBPF est implémenté à l'aide d'un compilateur JIT pour augmenter la performance [42]. La performance du bytecode résulte du fait que les instructions du langage sont reliées de près à l'ISA du CPU physique [43]..

2.2.6 Perf

Perf implémenté dans le noyau Linux, est un outil de profilage à l'échelle du système [44]. Il peut aussi faire du traçage à l'aide de l'instrumentation et l'insertion de point de trace. Pour faire du profilage, l'échantillonnage temporel et l'échantillonnage par évènement sont deux options possibles. Les trois types d'évènements *perf_events* sont :

- Les évènements logiciels sont déclenchés lorsqu'une condition sur le compteur logiciel dans le noyau est vraie. Parmi ces évènements logiciels, on trouve : le compteur de changement de contexte, le nombre de fautes de page, le nombre de migrations de CPU, etc..
- Les évènements matériels sont déclenchés par les compteurs matériels de l'unité de surveillance de performance (PMU) du CPU. Ces compteurs peuvent instrumenter les évènements matériels de bas niveau du CPU (p. ex., nombre de cycle, cache miss, branch miss, etc.) [45].
- Les évènements de point de trace sont déclenchés par les points de trace statiques du noyau (*TRACE_EVENT*), les points de trace dynamiques (Uprobe et Kprobe) et USDT.

L'information de profilage peut être affichée sous forme d'évènement et de la valeur de son compteur d'occurrence. L'autre forme de présentation est l'évènement, ses paramètres et son horodatage. Perf peut aussi afficher des traces de graphe d'appel lorsqu'un évènement est déclenché. Pour cela, le programme client doit être compilé avec *-fno-omit-frame-pointer*. Dans le cas contraire, Perf peut utiliser LBR (Last branch records) pour les CPU de Intel pour analyser la pile d'appels. LBR est un ensemble de registres (accessible par l'anneau 0) qui permet de faire l'échantillonnage des branchements [46].
Si jamais la fonction qui enregistre l'évènement dans Perf est jugée inefficace en termes de performance ou en termes de pertinence de données tracées, l'utilisateur peut la remplacer. Pour cela, il peut écrire du code eBPF et le compiler à l'aide de Perf. Le code eBPF peut être attaché à un évènement et s'exécuter à la place.

2.2.7 Gcov

Gcov est un outil de profilage et d'analyses de couverture de code qui utilise l'instrumentation [47]. Il est pris en charge dans les deux espaces utilisateur et noyau. Il est utilisé pour savoir combien de fois un bloc de base a été exécuté. Pour pouvoir l'utiliser, il est nécessaire que le programme soit compilé avec l'option *-fprofile-arcs et -ftest-coverage*. Ces options lient le programme avec la libraire d'exécution *libgcov* qui initialise le profilage dans la fonction *__gcov_init()* appelée par le constructeur de la bibliothèque. À la compilation, pour chaque bloc de base du programme, un compteur (variable globale) qui compte le nombre d'exécutions est créé [48]. L'instrumentation consiste en un incrément de ce compteur et est ajoutée directement dans le bloc de base. La figure

2.6 montre le code d'instrumentation d'un bloc. D'abord, la valeur du compteur du bloc de base est chargée dans le registre *rax*. Ensuite, le contenu de *rax* est incrémenté avant d'être récrit dans le compteur du bloc.

```
mov     0x20405a(%rip),%rax        # 2054a8 <__gcov0._Z3fooi+0x8>
add     $0x1,%rax
mov     %rax,0x20404f(%rip)        # 2054a8 <__gcov0._Z3fooi+0x8>
```

Figure 2.6 L'instrumentation ajouté pour gcov

Lorsque le programme se termine, le destructeur de la bibliothèque d'exécution de gcov fait appel à *__gcov_exit()* qui sauvegarde les informations du profilage dans plusieurs fichiers. Pour analyser ces fichiers de sortie, l'exécutable gcov prend en paramètre le fichier source du programme et annote le code source avec le nombre de fois que chaque ligne de code a été exécutée.

2.2.8 Gprof

Gprof est un outil d'analyse de performance pour les programmes écrits en C/C++. Il fut créé et implémenté initialement par Susan L. Graham et al. Une implémentation de gprof est faite comme extension de l'outil prof [49] et ajoutée dans le projet GNU. Pour analyser un programme, gprof a besoin du soutien du compilateur pour l'insertion des appels à la fonction *mcount()* à l'entrée de chaque fonction. Le compilateur, à l'aide de la bibliothèque standard C, implémente une version de *mcount()* qui fera le traçage. Au démarrage du programme instrumenté, une table de hachage avec l'adresse de la fonction appelante comme clé primaire et l'adresse de la fonction appelée comme clé secondaire est créée. La valeur contenue dans cette table est un compteur qui représente le nombre d'occurrences d'un appel d'une fonction par une autre. Cette table de hachage est remplie par la fonction *mcount()*. La bibliothèque standard C utilise aussi une minuterie de basse fréquence (interruption générée chaque milliseconde) pour échantillonner le compteur de programme ainsi que l'horodatage. Ces informations sont écrites dans un histogramme durant l'exécution. Une fois le programme terminé, la table de hachage ainsi que l'histogramme sont écrits dans un fichier *gmon.out*.

Gprof prend le nom de l'exécutable tracé et le fichier *gmon.out* comme paramètre pour afficher le résultat de l'analyse. En effet, pour pouvoir afficher les résultats, les données encodées dans le fichier *gmon.out* doivent être reliées aux informations du binaire (symboles de débogage, adresses des symboles, etc). Gprof effectue du traitement pour calculer le nombre d'appels et le pourcentage du temps d'exécution de chaque fonction. Le graphe d'appel dynamique est aussi estimé depuis les données d'échantillonnage.

2.2.9 LTTng

LTTng est un outil de traçage du noyau et de l'espace utilisateur [7]. Il utilise les points de trace statiques de la composante LTTng-UST pour instrumenter l'espace utilisateur. La bibliothèque *lttng-ust-cyg-profile.so* peut être préchargée dans le programme tracé pour intercepter les appels ajoutés par l'option *-finstrument-functions* du compilateur. LTTng ne peut pas s'attacher à un processus client. La composante LTTng-UST contient deux sous-composantes, l'agent LTTng-UST-Java et l'agent LTTng-UST-Python, qui font l'instrumentation de Java et Python. La composante de LTTng qui effectue le traçage dans le noyau Linux est un module inséré dynamiquement. Elle fait aussi l'instrumentation à l'aide des points de trace statiques ajoutés par Ftrace (TRACE_EVENT) et Kprobe. À l'aide de la macro *LTTNG_TRACEPOINT_EVENT(),* le projet LTTng-modules offre à l'utilisateur l'option de définir un évènement qui sera inséré dans le noyau avec la composante de traçage. Cet évènement sera déclenché chaque fois que le point de trace correspondant dans le noyau est rencontré. Les deux modules (de l'espace utilisateur et du noyau) créent des tampons circulaires pour chaque CPU. Ces tampons, partagés avec le *démon* consommateur, enregistrent les évènements localement. Le *démon* peut les envoyer sur le réseau ou les enregistrer sur disque. LTTng se sert de URCU (user-space RCU) et RCU comme mécanismes de synchronisation dans l'espace utilisateur et dans le noyau [50, 51].

2.2.10 Ftrace

Ftrace est un ensemble d'outils de traçage implémenté dans le noyau Linux [52]. Le traceur de fonctions de ftrace trace le temps d'exécution de chaque fonction instrumentée. Le traceur de graphe trace les fonctions appelées et les fonctions appelantes pour afficher le graphe d'appel [53]. Ftrace exporte une interface pour recevoir les évènements de points de trace statiques (*TRACING_EVENT*) depuis l'espace utilisateur à l'aide du système de fichier *tracefs*. Ces évènements peuvent être utilisés par les traceurs depuis l'espace utilisateur (p. ex., Uftrace).
Ftrace peut utiliser les points de trace statique du noyau, Kprobes, et uprobes. De plus, il peut instrumenter le noyau Linux compilé avec l'option *-pg*. Cette option ajoute des appels à *mcount()* à l'entrée de chaque fonction. Ftrace écrase l'adresse résolue dynamiquement par l'éditeur de lien pour *mcount()* dans le but d'appeler sa propre version. Cela peut se faire dynamiquement sans corruption de l'exécution. Pour tracer le graphe d'appel, *mcount()* enregistre la fonction appelée ainsi que la fonction appelante. Pour tracer le temps d'exécution de chaque fonction, *mcount()* calcule et enregistre la différence temporelle entre l'entrée et la sortie de la fonction appelée. Même lorsque l'appel à *mcount()* ne fait pas de traçage, un surcoût de 13% est remarqué [54]. Pour cette raison, Ftrace prend aussi en charge l'instrumentation du noyau compilé avec l'option *mcount-mnop*. Cette option ajoute l'instruction *nopl 0(%eax, %eax, 1)* au début de chaque fonction. Le remplacement

du *NOP* par un branchement relatif est différent selon qu'il est fait avant ou après le démarrage du noyau. Durant le démarrage, un seul coeur du système *SMP* est actif. Les autres ne sont connectés que lorsqu'il n'y a plus de différence entre le coeur primaire et secondaire. Le remplacement peut donc se faire sans complexité [55]. Durant l'exécution, pour faire un remplacement atomique du *NOP* par un branchement relatif de la même taille, Ftrace utilise une technique similaire à celle de Kprobe. Il remplace le premier octet du *NOP* par une instruction d'interruption pour garantir qu'aucun processeur ne l'exécutera dorénavant. Ensuite, il écrit le déplacement (quatre derniers octets) du branchement avant d'envoyer une interruption inter processeurs (*IPI*) à tous les coeurs du système, dans le but de sérialiser leur exécution. [6] Finalement, l'instruction d'interruption est remplacée par le code d'opération (premier octet) du branchement, dans le but d'activer le point de trace.
Autres que le traceur de fonction et le traceur de graphe d'appel, Ftrace offre plusieurs traceurs qui utilisent tous les mêmes techniques d'instrumentation expliquées auparavant. Parmi ces traceurs, nous citons : *blk* pour tracer les opérations d'entrées/sorties de périphérique de bloc, et *mmiotrace* pour tracer les opérations d'entrées/sorties de mémoire virtuelle. Le traceur *Nop* remplace l'instrumentation par des instructions *NOP* qui ne font aucun traçage. Les traceurs de Ftrace se servent d'un tampon en anneau par processeur pour enregistrer les évènements. Si les évènements arrivent plus vite que leur consommation, les anciens évènements seront écrasés dans le tampon. Il est possible de changer la taille du tampon pour éviter ce genre de situation.

2.2.11 Uftrace

Uftrace est un outil de traçage des applications écrites en C/C++. Son utilisation principale est de tracer les applications dans l'espace utilisateur [56]. Cependant, il peut tracer le noyau à l'aide des points de trace statiques *EVENT_TRACING*. Il est inspiré de ftrace, surtout le module qui trace et affiche le graphe d'appel avec le temps que chaque fonction a pris pour s'exécuter. Uftrace ne peut pas s'attacher à un processus pour le tracer. À la place, il crée un nouveau processus (fork()) et utilise *LD_PRELOAD* pour précharger la bibliothèque d'instrumentation et de traçage *libmcount.so dans son espace d'adressage*. Pour instrumenter les appels aux fonctions qui se trouvent dans les bibliothèques dynamiques, Uftrace les intercepte à l'aide de la table *Procedure Linkage Table*. Il prend en charge les binaires compilés statiquement avec *mcount (-pg)*, *mcount-nop (-mnop-mcount)*, *cyg-profile (-finstrument-functions)* et *XRay (-fxray-instrument)*. Il peut aussi instrumenter dynamiquement les binaires [57]. Pour l'instrumentation, un branchement relatif de cinq octets est inséré à l'entrée des fonctions, si cela ne corrompt pas l'exécution du programme. Même si ce type d'instrumentation est considéré comme dynamique, Uftrace ne peut pas le faire durant l'exécution. En

6. Le retour de *IPI* fait appel à l'instruction *IRET* dans *x86*, sérialisant ainsi l'exécution de coeur.

effet, toute l'instrumentation est effectuée avant que le programme n'atteigne son point d'entrée. Uftrace a deux modules principaux, le traceur et le tracé. Le traceur est un processus qui s'occupe de recevoir les évènements et les enregistrer sur le disque. Les évènements sont envoyés depuis la bibliothèque *libmcount.so* chargée dans le processus tracé. Ces évènements sont envoyés à travers un ou plusieurs tampons de mémoire partagée pour accélérer l'exécution. La synchronisation de ces tampons se fait à l'aide de l'écriture/lecture atomique de drapeau. Une autre partie de la synchronisation qui coordonne le déroulement du traçage se fait à l'aide des *Unix socket*.

2.2.12 Conclusion du chapitre

Les différents outils de traçage utilisent l'instrumentation pour récolter et enregistrer les évènements dans une trace. Dans le cas où le code source est disponible, et en se servant de l'instrumentation statique, l'utilisateur peut appeler ses points de trace directement dans le code. Il peut aussi demander au compilateur d'ajouter automatiquement le code d'instrumentation au début de chaque fonction. Le code d'instrumentation peut être un appel à une fonction de traçage ou bien de l'espace sous forme d'instruction *no-operation*. Durant l'exécution, l'espace peut être modifié pour le remplacer par un saut vers le code de traçage. Certaines techniques de réservation d'espace ne préparent pas le terrain pour une insertion dynamique sécuritaire à la volée. Dans ce cas, les outils de traçage peuvent se servir d'un algorithme spécialisé pour le remplacement d'instruction.

Dans le cas où l'utilisateur évite de recompiler le code source, ou dans le cas où ce dernier n'est pas disponible, l'instrumentation dynamique est une alternative. L'insertion de l'instrumentation peut se faire dans un exécutable sur disque ou dans l'espace d'adresse virtuel durant l'exécution. Avant l'exécution (sur disque), les outils d'instrumentation qui modifient le binaire après l'avoir désassemblé peuvent être un bon candidat pour l'instrumentation. Cependant, durant l'exécution, ces outils doivent soit arrêter complètement le processus pour appliquer la modification, soit le ralentir significativement pour faire la compilation JIT. Pour pallier ce problème, un autre type d'instrumentation dynamique remplace seulement l'instruction à instrumenter par une instruction d'instrumentation. En faisant ainsi, des algorithmes de modification de code plus légers et moins intrusifs peuvent entrer en action. Cependant, ce type d'instrumentation ajoute souvent plus de surcoût, parce que les instructions originales doivent être soit émulées, soit déménagées pour être exécutées dans un tampon OLX

CHAPITRE 3 PROBLÉMATIQUE

Un point de trace statique est un appel à une fonction de traçage qui enregistre l'évènement associé. Il est souvent inséré manuellement par le développeur dans le code source. Une fois compilé, le programme instrumenté intègre le point de trace de manière stable et performante. En effet, le compilateur s'assure que l'insertion du point de trace est optimale au niveau de l'espace et du temps d'exécution. Pour enlever ou ajouter de nouveaux points de trace, la reconstruction du programme est nécessaire. À la compilation, une condition de désactivation du point de trace statique est ajoutée dans le code et peut être basculée à la volée. Une condition peut être un peu lourde sur le système. En effet, si le branchement de la condition n'est pas pris, et que la prédiction du branchement n'est pas correcte, le processeur doit vider le pipeline, ce qui réduit la performance. Le compilateur peut remplacer la condition par des instructions *nop*, à la place du point de trace statique. Le développeur ou la librairie de traçage peut donc remplacer ces instructions par un branchement vers la fonction de traçage. Pour instrumenter l'entrée et la sortie des fonctions, le compilateur peut ajouter statiquement des appels à une fonction de traçage liée dynamiquement dans une librairie partagée. L'utilisateur peut précharger une libraire qui contient sa propre version de la fonction de traçage et qui écrasera l'ancienne version. Toutes les techniques d'instrumentation statiques ont la même limitation, qui est la nécessité de recompiler le programme. Avec l'absence du code source, l'instrumentation statique est impossible à réaliser. D'où l'importance et l'utilité de l'instrumentation dynamique.

L'instrumentation de l'espace utilisateur est différente de celle du noyau. Le noyau a accès à des ressources et outils inaccessibles par l'espace utilisateur. Malgré le fait que le noyau essaye d'offrir une couche d'abstraction complètement raisonnable au besoin de l'espace utilisateur, certaines fonctionnalités sont très critiques au système et peuvent l'exposer à des problèmes de performance, ou encore pire corrompre son intégrité. Certains problèmes que l'instrumentation peut rencontrer sont résolus d'une manière simple et efficace dans le noyau, alors que ces problèmes peuvent constituer un grand défi dans l'espace utilisateur. Le changement du code exécutable pour insérer un point de trace durant l'exécution dans un programme à plusieurs fils requiert l'exécution d'une instruction de synchronisation sur tous les coeurs sur lesquels ces fils s'exécutent. Le noyau Linux a accès aux interruptions interprocesseur, un mécanisme de synchronisation matérielle de bas niveau avec un impact minime sur la performance, qui peut être utilisé pour interrompre n'importe quel coeur et le forcer a exécuter l'instruction de synchronisation. Malheureusement, ce mécanisme n'est pas directement disponible depuis l'espace utilisateur. En pratique, il est possible de charger un module d'instrumentation dans le noyau qui aurait un accès total à ces ressources. Cependant, cela requiert le privilège *root*. De plus, l'insertion d'un module dans le noyau dans le but de tracer

l'espace utilisateur est intrusive et n'est souvent pas souhaitable.

Deux types d'instructions sont connus pour insérer dynamiquement un point de trace : les instructions d'interruption logicielle et les branchements (sauts et appels de fonction). Ces deux types d'instruction permettent au fil qui les exécute de modifier son compteur de programme et de transférer son flot d'exécution vers la fonction de traçage. Les interruptions logicielles, lorsqu'exécutées, basculent le fil d'exécution en mode noyau pour les traiter. Une série d'appels de fonctions est faite avant de rebasculer en mode utilisateur pour appeler la routine de traitement de l'interruption qui est définie par l'utilisateur. Comparé à un saut ou un appel de fonction qui atterrit aussitôt sur la fonction de traçage, les interruptions logicielles ajoutent un grand surcoût d'exécution au point de trace. En dépit de la faible performance, les interruptions logicielles sont fréquemment utilisées pour leur simplicité d'insertion. En effet, avec une taille de un octet, elles peuvent remplacer n'importe quelle instruction sans chevauchement. De plus, une instruction de un octet peut être insérée dans un code en exécution à tout moment, sans que les coeurs observent un résultat incohérent au niveau du cache ou de la mémoire. Un branchement relatif a une taille de cinq octets, ce qui implique que le chevauchement de plusieurs instructions est possible. Si un branchement atterrit sur la région chevauchée et modifiée par le branchement, l'exécution du programme sera possiblement corrompue. En outre, l'écriture de deux octets ou plus dans un code durant l'exécution peut causer des incohérences de cache. Cela causera une défaillance de protection générale (GPF). Une instruction de sérialisation de coeur doit être exécutée sur chaque processeur avant d'exécuter le nouveau code. Intel définit un algorithme de XMC qui permet de faire la sérialisation de coeur. Néanmoins, il utilise un drapeau de synchronisation qui doit exister avant l'insertion. Puisqu'on a besoin du drapeau pour faire l'insertion et qu'on a besoin que l'insertion ajoute le drapeau, une situation paradoxale se manifeste.
Il est important de mentionner qu'il est possible d'arrêter un processus cible pour qu'il n'exécute pas le code inséré. Ensuite, la valeur du compteur de programme de chaque fil est vérifiée. Si un fil est en train d'exécuter une région à instrumenter, elle est transférée en dehors et le code d'instrumentation est inséré. Une sérialisation de coeur est forcée et le processus poursuit son exécution. Cette technique est utilisée par les débogueurs autant que les outils de traçage. Par contre, elle est trop intrusive et n'est pas très efficace pour la détection des problèmes de performance et les problèmes dus au parallélisme.

Dans le prochain chapitre, nous présenterons une technique d'instrumentation dynamique rapide de l'espace utilisateur nommée NOProbe. La technique combine différentes stratégies d'instrumentation pour atteindre un taux de succès très élevé. Dans les stratégies principales, nous utilisons des branchements relatifs de 2 et 5 octets. Lorsque le branchement de 5 octets chevauche plusieurs instructions et risque de corrompre le flot d'exécution, celui de 2 octets peut être considéré. En effet, deux octets est la taille minimale pour créer la situation de chevauchement. Pour arriver au code

de traçage, le saut de 2 octets n'est pas suffisant. Dans ce cas, NOProbe exploite les rembourrages ajoutés par le compilateur comme espace pour contenir un trampoline. De plus, un branchement de 2 octets chevauche rarement la mémoire cache, ce qui nous permet d'éviter le surcoût de la sérialisation reliée au problème de XMC.

Pour modifier le code simultanément sans interrompre l'exécution, un algorithme est présenté. L'algorithme dans une première étape nous permet d'aborder le problème d'exécution et modification simultané par différents fils d'exécution. Dans une deuxième étape, l'algorithme accomplit la sérialisation des coeurs avec un minimum d'intrusion pour pallier au problème de XMC.

CHAPITRE 4 ARTICLE 1 : NOPROBE : A FAST MULTI-STRATEGY PROBING TECHNIQUE FOR X86 DYNAMIC BINARY INSTRUMENTATION

Submitted to *ACM Transactions on Programming Languages & Systems*
Submition date : April 26, 2020
Anas Balboul
Ahmad Shahnejat Bushehri
Michel R. Dagenais

Abstract

Tracing is an effective technique to analyze the performance of complex multi-thread programs. A program trace consists of a series of events written by probes. The performance of a probe insertion and execution is very important for analyzing high-performance online systems. Existing techniques are either limited in scope, too intrusive or costly in overhead, limiting their applicability in many cases.

We introduce a new instrumentation technique for the x86 architecture, named NOProbe (NOP-based Probe), that exploits several complementary strategies in order to achieve a higher success rate in insertion of fast jump-based probes at different locations in a binary. For one of these strategies, we exploit NOP-paddings inserted by compilers to optimize the code alignment. To achieve this goal, two new algorithms are introduced to assign paddings to probes. One is greedy and emphasizes performance, while the other attempts to globally optimize the assignation of paddings to probes in a function. We also propose an improved algorithm (*Lock and Load*) used by NOProbe to dynamically insert probes on the fly. The algorithm can patch a whole basic block by locking it with a trap instruction before redirecting the thread out of it. Next, the basic block is patched and the cores are serialized before unlocking it. The experimental results demonstrate that our approach not only achieves a higher success rate for fast probes insertion and maintains a very low execution overhead, but outperforms competing techniques.

4.1 Introduction

To discover the target process execution state or to observe performance problems in a program, developers use tracing tools with minimum performance impact on the process execution. This is absolutely crucial since many issues cannot be captured if the tracing code significantly perturbates

the execution. With a tracing framework, one can figure out the internal state of a running process and extract valuable low-level information. Run-time binary modification is already used in various areas such as cache simulators [58], dynamic kernel patching [59], and JIT compilers [60]. Many tracing frameworks, including DTrace [61], explored the role of binary modification for tracing and debugging purposes.
To trace a program, probing code can be inserted into the binary, enabling to gather valuable information out of the execution. Instrumentation may be performed on the source code statically before the execution, or on the binary dynamically after the execution has started. A probe or tracepoint, is the code inserted by the instrumentation and in most cases, is a conditional function call or a trap. It could be inserted into the source code at compile-time, or into the binary before or after the execution has started.

During the execution of a target program, dynamic instrumentation is the process of manipulating and monitoring the code, at the instruction level granularity, to offer some functionality like tracing. There is no need to recompile the target program as dynamic instrumentation uses dynamic code injection mechanisms. Multiple techniques may be used for dynamic binary instrumentation, in order to insert probes and user-supplied analysis routines at run-time. As a separate process, the Instrumentation Code Injector program controls the target while injecting code for tracepoints and runtime support. This runtime support is an Instrumentation Agent, loaded in the same address space of the running target program. The Instrumentation Agent controls the tracepoints and receives commands from the Instrumentation Code Injector. For example, the GNU Debugger [62] can inject an In-Process Agent into a running program to insert low overhead conditional tracepoints. However, Dynamic Binary Instrumentation (DBI) is not limited just to debugging and tracing. Since DBI techniques are flexible and not dependent on source code or compiler support, they have been used in other domains like software security and software testing.

Here are definitions for the concepts illustrated in Figure 4.1 and used throughout the article :

— Probe site : The location at which a probe should be inserted.
— Original instruction(s) : One or more instructions, starting at the probe site, and being replaced by the probing instruction.

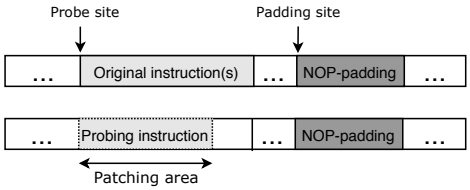
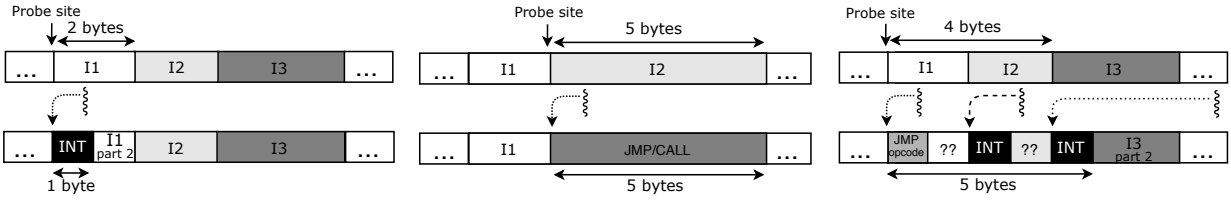


Figure 4.1 General definitions.



(a) Trap replacement. (b) Branch replacement. (c) Tranch replacement.

Figure 4.2 Original instruction(s) replacement by a probing instruction in Trap-based, Branch-based and Tranch-Based techniques.

— Probing instruction : The code branching instruction, replacing the original instruction(s) at the probe site, in order to transfer the control flow.
— Padding site : The location in which one or more no-operation instructions would be replaced by a code branching instruction (e.g., a 5-byte relative call) in order to reach the probe handler.
— Patching area : The region where the original instruction(s) will be replaced, partly or completely, by the probing instruction.

In this paper, we propose NOProbe, a user-space dynamic instrumentation technique for x86. We developed a multi-strategy algorithm to insert low overhead probes almost anywhere within a process. We propose a low-cost patching algorithm that can atomically patch a basic block [1] in SMP environments. Our instrumentation technique does not resort to a stop-the-world approach and installs the probe safely with limited intrusiveness. One of the strategies used is to exploit the significant padding space available in optimized x86 64-bit binaries.

Our main contributions in this paper are :

— We propose a multi-strategy user-space dynamic instrumentation technique that can insert a fast probe at almost any instruction location in a program.
— We developed an improved user-space algorithm that can safely and concurrently patch a basic block of instructions.
— We implemented our solution in a user-space tracing tool (*uftrace*), redesigning it to dynamically attach to a running process and inject instrumentation.
— We analyze and compare the probe performance of multiple existing instrumentation techniques.

The rest of this paper is organized as follows. Section 4.2 explains the different steps required for dynamic binary instrumentation and surveys existing approaches. In Section 4.3 we detail the

1. A sequence of instructions with no incoming branches, except to the first instruction, and no outgoing branches, except from the last instruction.

design and architecture of the proposed NOProbe technique. Section 4.4 presents our experimental results comparing NOProbe with existing probing techniques. Finally, section 4.5 concludes the paper and suggests improvements for future investigations.

4.2 Background and Related Work

The *x86* is one of the most popular platforms in production systems. However, dynamic instrumentation tools developers have been struggling with multiple challenges associated with this architecture. First, instructions in *x86* have variable sizes and thus not aligned to either word, cache line or even page boundaries, unlike most modern instruction sets. Common instructions in most cases have shorter and more complex encodings. This may shrink the code size, help reducing the footprint in cache memory, and saving clock cycles and energy. However, variable size instructions add complexity to dynamic instrumentation. A simple and effective DBI technique is to atomically replace the original instruction(s) with a code branching to the probe handler. With variable size instructions, if the probing instruction size is greater than the original probed instruction, it will overlap subsequent neighbor instructions. If the program branches to one of the overlapped instruction(s), the processor will try to interpret the bytes in the middle of the probing instruction as an instruction, and undefined behavior will result (Figure 4.3).

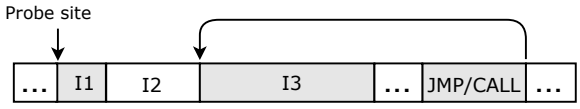
Another important issue in DBI is the concurrency in multi-thread systems. Furthermore, this issue compounded with variable size instructions adds significant complexity. For instance, when a probing instruction replaces several original instructions, further steps are required to be taken for a thread that is executing, preempted, or interrupted in the patching area. For example, if the Instrumenting Agent thread in Figure 4.3 writes the probing instruction starting at *I1*, while another thread is already interrupted at *I2*, it might cause the latter to see an invalid instruction when it is resumed. These issues, and the existing techniques to cope with them, are presented in the next subsections.

4.2.1 Instrumentation Types

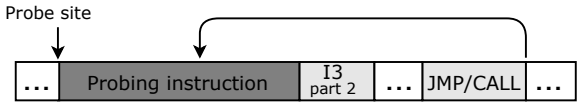
To insert a dynamic tracepoint at arbitrary instruction locations on x86 platforms, one of the three following approaches are typically used :

— Trap-based techniques
— Branch-based techniques
— Tranch-based techniques (Trap-Branch)

Trap-based techniques (e.g., uprobe and unoptimized kprobe [63]) use a trap as a probing instruction. In this case, as you can see in Figure 4.2a, the first byte of instruction *I2* has been substituted



(a) Before overwritting.



(b) After overwritting.

Figure 4.3 Undefined execution behavior.

with one byte *INT3* as a probing instruction. A trap instruction generates a *SIGTRAP* signal. It is also possible to use an illegal instruction instead of a trap to generate a *SIGILL* signal at execution time. However, in both cases, the software interrupt is processed by the operating system, which then calls the user-level signal handler. The target process running in user-space, executing a trap or illegal instruction causes several transitions between user-space and kernel space. The interrupt is handled in kernel space, then the user-space signal handler is called, and control returns to kernel space. After that, execution resumes in user-space to skip the trap, execute the overwritten instruction out-of-line and continue after the trap. This incurs a significant overhead which may be as high as 600 to 900 times slower than the Branch-based techniques on x86 platforms [62, 64].

Branch-based techniques are a collection of approaches that stay in user-space, to dynamically instrument binaries, without the overhead of Trap-based mechanisms. These techniques typically rely on 5-byte *jmp* or *call* instructions (e.g., GDB fast tracepoint). Therefore, they are unable to instrument a location if the size of the target instruction at the probe site is less than 5 bytes, which is the case for 75% of x86 instructions [62,65]. The branch in Figure 4.2b could be a 5-byte relative *jmp (0xe9)* or *call (0xe8)*, with a 4 bytes signed offset. This requires the branch target to be within a range of maximum 2 GB away from the branch source (before or after the probe site). Because of these constraints on the target instruction size and the probe handler address range, the insertion of a branch is much more difficult than of a 1-byte trap/illegal instruction.

When the target instruction is shorter than 5 bytes, Tranch-based techniques (e.g., liteInst and dyntrace) [66, 67] may be used that overwrite one or more neighbor instructions with a 5-byte branch, as depicted in Figure 4.2c. If one or more overlapped neighbor instructions are the destinations of branch instructions, the branching threads would find a possibly invalid sequence of bytes replacing the original instruction. However the relative offset of the 5-byte *jmp/call*, inserted by the Tranch-

based techniques, could be selected to contain a trap/illegal instruction at the branch destination. In that case, the signal handler could execute the overwritten instructions out-of-line and continue with the next instruction, as shown in Figure 4.2c. The Tranch-based techniques are interesting when it is very unlikely that a branch would target such a trap embedded in the relative offset. On the opposite, the embedded trap might be located on a hot code path like the beginning of a loop. In this case, the technique will perform significantly worse than putting a Trap-based probe on the first byte of the target instruction immediately preceded the loop. Another problem is that embedding traps or illegal instructions in the relative offset limits the possible locations for the probe handler, thus the available unused space in the process may not be reachable.

4.2.2 Intel Cross/Self Modification Challenges

Run-time cross thread code modification on the Intel x86 platform, in a multi-thread symmetric multiprocessing environment, is intricate, not very well documented, and may generate General Protection Faults (GPF). The instrumentation thread running on processor core *X* could modify some code, while the same code is concurrently executed by another thread running on processor core *Y* [2]. A first concern is cache straddling, when the size of the probing instruction is more than 1 byte. There is no problem for atomically inserting trap instructions, since they occupy a single byte. Longer instructions may sit across the cache-line boundary and cannot be written atomically [68]. Accordingly, the thread running on processor core *Y* might find an invalid partially updated instruction.

Further problems lie in the coherence of the instruction prefetch cache. If the thread on processor core *Y* prefetches the unmodified copy of the code, the micro-operations will be generated accordingly into the trace cache [69]. Since several optimizations such as reordering, fusion, and caching may affect the structure of micro-operations, no correlation would be noticed anymore between the original code and the optimized micro-operations. Therefore, the processor no longer keeps track of the corresponding start and end of prefetched instructions in the trace cache. Therefore, if the CPU invalidates the trace cache because of an unsynchronized cross-modification, the instruction execution will be terminated with a GPF exception.

According to Intel's erratum recommendation, issuing a *CPUID* instruction on each processor ensures that all modifications to memory regions, registers, and flags for executing instructions get finalized before the new instructions get executed. As shown in Figure 4.5, although invoking *CPUID* using Intel's protocols for cross-modification could solve the problem, it remains unsatisfactory. Indeed, the protocol not only requires a considerable space at the probe site, but the busy waiting mechanism imposes a remarkable latency that NOProbe cannot afford for performance reasons.

4.2.3 Safe Code Insertion Approaches

Modifying a program at run-time for instrumentation purposes, must be achieved without altering the normal process execution. As explained previously, dynamically modifying machine code on modern multi-core platforms can lead to many incoherent states in instruction code, instruction cache, instruction prefetch cache, and can cause a general protection fault. In this context, several techniques were proposed for safe cross thread code modification :

1. Stop-the-world approach :
   Stopping all the threads during dynamic instrumentation greatly simplifies the problem. This ensures that any concurrent thread running in the modified code section does not intervene during the patching. However, stopping all the threads in a process for the sake of instrumentation is highly intrusive and not practical in many performance-sensitive cases.
2. Before-the-sunrise approach :
   This approach seeks quiescence by pre-loading a library into the target program's address space. It instruments the program before its entry point, when no thread is executing the target code. Similarly, the tracepoints inserted before the execution starts cannot be removed once the execution has started. This is easier but even more constraining than the Stop-the-world approach. We have encountered many use cases where tracing tools are brought in to analyze a long-running process in production, that starts to misbehave. This approach cannot be used in these cases.
3. Atomic modification approach :
   The patching area of the target process could be atomically modified, such that it is unnecessary to stop all the threads, and it would be much less invasive. By writing atomically a probing instruction over the probe site, the control flow transfers to the probe handler. Special care would still be needed not only for instructions less than 5 bytes, but to ensure the safety of cross thread code modification.

4.2.4 No-Operation Instructions and Compiler Padding

Compiler Padding Modern processors are heavily optimized. They generally fetch a whole instruction cache line at a time, ranging in size between 16 and 64 bytes. In that context, it is interesting to align basic blocks (e.g., the start of a loop or a function) on cache line boundaries. This increases prefetching and cache effectiveness, thus improves performance. Compilers exploit this mechanism by inserting padding to align instructions on the next cache line boundary. This trade-off of space wasted in padding for improved execution performance is typically applicable to four critical regions : functions start, loops, labels, and jumps. They are respectively enabled in GCC with options

-falign-functions, *-falign-jumps*, *-falign-labels* and *-falign-loops*. All these options are enabled at the *-O2* optimization level and higher.

No-Operation Instructions (NOPs) The padding may consist of a single or multiple NOP instructions to fill the gap. Table 4.1 lists the various types of NOP-paddings and the corresponding assembly instruction. The NOP-paddings byte sequences consist of a set of instructions that perform no operation and have absolutely no effect on the content of their register operands or flags. They only increment the program counter (*PC*) to the next instruction. The first entry, encoded as a single-byte, *0x90*, is the most commonly used NOP-padding instruction in the x86 instruction set. Although *0x90* is identified as a NOP instruction, the same binary opcode also corresponds to *xchg %rax, %rax* (exchange *rax* with itself) in the x86 architecture.
The second entry indicates another *xchg* instruction using a redundant prefix (*0x66*) that does not affect the NOP behavior (*0x90*). In the case of the 9 bytes NOP-padding, the *0x66* takes the first byte as a redundant prefix to override the size of the operand. The neighboring two bytes, *0x0F 0x1F*, are the instruction opcode. The 4^{th} and 5^{th} bytes, *0x84* and *0x00*, represent the operands format specification, *Mod R/M* and *SIB*, and the last four bytes, *0x00000000* are used to hold the displacement of the memory operand.

Reachable and Unreachable NOP-Padding The NOPs in a program may or not be reachable. Reachable NOP-paddings could be executed by the program at some point. In Figure 4.4b, if the *Jump if not sign* condition is not satisfied, the branch will not be taken and the *nopl* instruction will be executed.
In Figure 4.4a, an unreachable padding is located at the end of the function, immediately after the *ret*. While unreachable NOP-paddings may safely be modified, reachable NOP-paddings should be modified very carefully because of possible concurrent execution.

Embedding Data as Code in NOPs For some NOPs, like the 9-byte one in Table 4.1, the no-operation behavior will remain unaffected, even if the *SIB* field and the 4-byte displacement are modified. This allows embedding data or code into the NOPs without impacting the program execution [70]. [71] exploit a similar technique to obfuscate and hide code using certain instructions in which several bytes are embedded. They mainly focus on using 9-byte no-operation instruction.

4.2.5 Memory Constraints

An instrumentation technique should have a minimal impact on the virtual space address of the program. Intrusive allocations could result in the inability of the program to carry out its execution.

The default maximum number of memory-mapped regions for a process is found in */proc/sys/vm/-max_map_count* on Linux (65536). Because Tranch-based techniques have some trap instructions embedded in the displacement of the probing instruction, they may have little choice in choosing the possible locations for the probe handler. In the worst case, one *mapping* may be required for each probe. As a consequence, instrumenting big programs (e.g., chrome, llvm-ar) could possibly consume every available mapping of the target process. A second concern is that excessive *mmap*-ing externally fragments the memory. The chunks of a highly fragmented virtual memory space may not be sufficient for larger future mappings. Since a successful Tranch-based probe insertion is highly dependent on the allocation of unused memory space, high fragmentation may degrade the scalability of probe insertions. Branch-based techniques, without embedded traps that constrain the displacement of the probing instruction, incur much less fragmentation as they could precisely choose their target.

Many existing DBI techniques on x86 use 5-byte *jmp*. With a jump, the probe handler must be specific to a tracepoint. It will call the tracing function before executing the replaced instruction(s) and jumping back to the instruction after the probing instruction. It is also possible to be used as probing instruction to transfer control to a shared probe handler. This requires maintaining a table of probing instructions locations and the associated replaced instructions to execute them out of line. However, with a single shared probe handler, memory fragmentation is reduced and the probe insertion success rate increased.

Other more invasive techniques may also be used that sidestep in part the cross thread code modification problem. One possibility is to generate a new instrumented copy of a function, add the new copy to the process address space, and replace the original function prologue by a *jmp* to redirect it to the new one. This is used in the Linux kernel live patching tool ksplice [32], which requires the source code to compile the new version of the function.
Dynamic binary translation (DBT) is another DBI technique that can move a whole function or basic block to a new memory location in order to insert instrumentation (e.g., dynInst) [64]. DynInst does not require the target program source code since it disassembles the binary code into a higher-level intermediate representation, inserts the instrumentation and recompiles the code. These systems, replicating large sections of code, incur a high memory overhead when multiple probes are inserted. Moreover, dynInst requires a considerable amount of memory for its program analysis, and is usually not applicable in memory-constrained systems such as embedded processors.

4.2.6 Out of Line Execution

The original instructions, the ones are overwritten by the probing instruction, must eventually be executed to keep the control flow of the program intact. DBI techniques save a copy of the true code in an out-of-line execution (OLX) buffer (Figure 4.6). Then, after the instructions in the OLX buffer have been executed, a *jmp* or a trap at the end of the buffer returns the control to the first valid instruction after the patching area.

Instructions that do not depend on the program counter can be copied to the OLX buffer without modification. However, the effective address of the memory operand in position-dependent instructions is actually established based on the current value of the instruction pointer. Therefore, the copied position-dependent code in the OLX buffer would not be executed correctly without relocation. It rectifies their relative addresses, in order to be executed with the same effect as the original instructions. Moreover, if an instruction is relocated far away from its target in memory, the addressing mode may not be able to reach it. Those instructions need to be manipulated, for instance with an addressing mode allowing more bytes for the offset, to be able to reach their target address.

4.2.7 Compiler-Assisted Probing Tools

Some Branch-based techniques limit the size of instrumented instructions to just 5 bytes or more. To relax this constraint, compiler-assisted techniques are useful, adding enough space at interesting probe sites at compile time. At run-time, the space is dynamically patched to insert the probe. Techniques like the GCC compiler *–mnop-mcount* option insert a 5 bytes no-operation instruction *nopl 0x0(%rax,%rax,1)* at the entry of functions. Therefore, patching the no-operation instruction with 5 bytes branch instructions is achieved more easily. However, cross modification problems should still be addressed carefully. For this purpose, in the Linux kernel ftrace locks the probe site with trap instructions before synchronizing the cores via a low overhead inter-processor interrupt. In user-space, uftrace patches the NOPs prior to the program execution. Once the probe injected in the code, uftrace can not safely remove or even dynamically modify it.

Some compiler-assisted techniques leave space in the probe site such that the cross-modification is simpler during run-time. For example, Google Xray inserts a 2-byte *jmp* at the probe site to jump over a reserved NOP sleds [72]. The 2-byte *jmp* plays the role of a lock and is aligned in memory not to straddle the cache. At run-time, the NOPs would be patched without a problem, since they are being jumped over and thus unreachable. The probe is finally unlocked by patching the non-straddler 2 bytes *jmp*, without worrying about cross-modification issues.

The major disadvantage of compiler-assisted techniques is the need to access the source code for

Table 4.1 Multi-byte sequence of NOPs used by GCC and recommended by Intel [1]

Assembly (AT&T Syntax)	Byte Sequence (Hexadecimal)
nop	90
xchg %ax,%ax	66 90
nopl (%rax)	0f 1f 00
nopl 0x0(%rax)	0f 1f 40 00
nopl 0x0(%rax,%rax,1)	0f 1f 44 00 00
nopw 0x0(%rax,%rax,1)	66 0f 1f 44 00 00
nopl 0x0(%rax)	0f 1f 80 00 00 00 00
nopl 0x0(%rax,%rax,1)	0f 1f 84 00 00 00 00 00
nopw 0x0(%rax,%rax,1)	66 0f 1f 84 00 00 00 00 00
nopw %cs :0x0(%rax,%rax,1)	66 2e 0f 1f 84 00 00 00 00 00

```
0000000000027a00 <free_parsed_cmdline>:
                 ...
27a09:  48 8b 7f f8       mov    -0x8(%rdi),%rdi
27a20:  f3 c3             repz retq
27a22:  0f 1f 40 00       nopl   0x0(%rax)
27a26:  66 2e 0f 1f 84    nopw   %cs:0x0(%rax,%rax,1)
        00 00 00 00 00
0000000000027a30 <absolute_dirname>:
                 ...
```

(a) Two NOP-paddings aligning *absolute_dirname* to 16 bytes.

```
0000000000015610 <do_replay>:
                 ...
15763:  0f 88 d7 00 00 00    js   15840 <do_replay+0x230>
                 ...
15833:  89 30 ff ff ff       jns  15769 <do_replay+0x159>
15839:  1f 80 00 00 00 00    nopl 0x0(%rax)
15840:  83 c4 01             add  $0x1,%r12d
```

(b) NOP-padding aligning branch target (at address *0x15840*) to 16 bytes.

Figure 4.4 NOPs aligning code.

instrumentation. Furthermore, the probing locations are defined at compilation time, and do not allow to instrument arbitrary probe sites thereafter. Finally, since instrumentation code exists in every probe site during run-time (e.g., function entries/exits), this approach may bring a higher memory and time overhead to the execution, even when tracing is disabled. [67] measured a total overhead of 6% added by the *-finstrument-functions* compiler flag. This could be reduced to 2% when smaller functions are left uninstrumented, as implemented in Google XRay.

4.3 NOProbe Design and Architecture

We are proposing a new multi-strategy technique for DBI called NOProbe. In this section, the different components of this DBI technique are detailed, starting with the three different complementary strategies used for instrumentation (Figure 4.6).

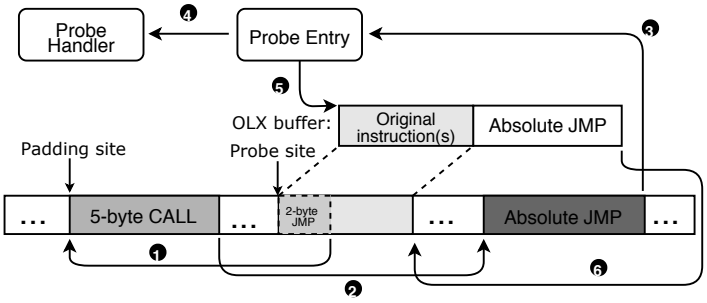
First strategy When the target instruction is at least 5 bytes long, or the target instructions form a basic block (no branches to internal instructions), a 5-byte relative *call* is used as probing instruction (Figure 4.6b). The *call* pushes the return address on the stack, which will be used as the probe identifier. If some code may branch to the overwritten instructions (except the first one), this strategy is not used and the second strategy will be attempted.

Second strategy The second strategy (Figure 4.6a) uses a smaller probing instruction (2-byte relative *jmp*). This reduces the chance of cache straddling and only requires a 2-byte basic block. The target probe location would be unsuitable only if the probed instruction is 1 byte long with possible branches targeting the following instruction. Having a single byte as offset, the 2-byte *jmp* can only reach target addresses from 128 bytes before to 127 bytes after the probe location. This

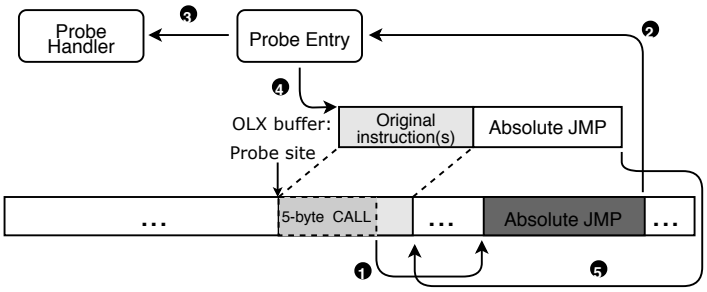
```
/* Action of Modifying Processor */
Memory_Flag ← 0;
Store modified code (as data) into code segment;
Memory_Flag ← 1;

/* Action of Executing Processor */
WHILE (Memory_Flag ≠ 1)
Wait for code to update;
ELIHW;
Execute serializing instruction; /* CPUID */
Begin executing modified code;
```

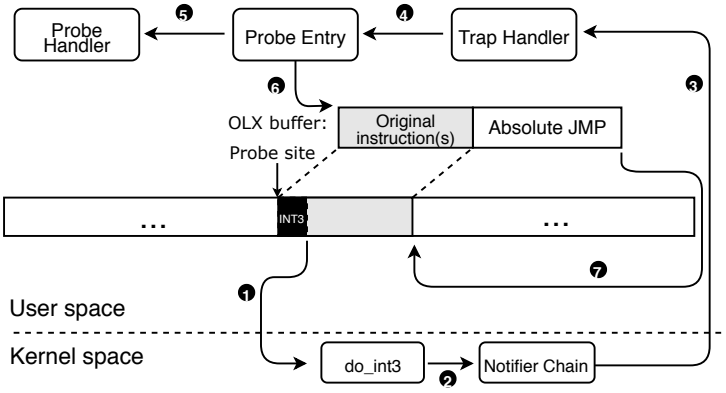
Figure 4.5 The algorithm recommended by Intel for cross-modification issue in SMP environment [2].



(a) NOProbe execution flow with a 2-byte *jmp* as probing instruction.



(b) NOProbe execution flow with a 5-byte *call* as probing instruction.



(c) NOProbe execution flow with an *int3* as probing instruction.

Figure 4.6 NOProbe execution flow.

is not sufficiently large to reach the probe handler, though. This is where NOP-paddings become useful. If an unreached NOP-padding is available within the 2 bytes jump range, it can be used to insert a trampoline with a larger jump, that will transfer control to the probe handler. The main challenge in this strategy is to exploit any NOP-padding, even if it seems unfitting at first. Moreover, if more than one NOP-padding is within range, we should globally optimize the assignation in order to maximize the probe insertion success rate. If we cannot ensure that we have a 2-byte basic block through static code analysis, or no suitable NOP-padding is within range, we have to resort to the third strategy.

Third strategy When neither of the first two strategies is applicable, we resort to a less efficient but always applicable trap (*INT3*) as probing instruction (Figure 4.6c). Once hit, the trap switches context to kernel mode and the kernel sends a signal to the process. The signal handler in the process is the probe handler which executes the instrumentation, the original instructions out of line, and returns to the kernel context with a *sigreturn*. The kernel then resumes the user-space execution.

The following proposed algorithm represents the aforementioned strategies in order :

Algorithm 1 Instrumentation routine

```
1:  procedure INSTRUMENTNOPROBE(addr)
2:      success ← False
3:      func ← getFunction(addr)
4:      if 5ByteStaticAnalysisPassed(func) then
5:          initCallProbe(addr)
6:          patchSiteCall5B(addr)
7:          success ← True
8:      else if 2ByteStaticAnalysisPassed(func) then
9:          nop ← findSuitableNop(addr)
10:         if nop ≠ None then
11:             initNopProbe(addr, nop)
12:             patchPaddingSite(nop)                ▷ with a 5-byte call
13:             patchSiteJmp2B(addr, nop)
14:             success ← True
15:     if success ≠ True then
16:         initTrapProbe(addr)
17:         *addr = trapInstruction
```

Shared trampoline The probe handler is part of a shared dynamic library injected into a free memory region of the target process, typically far from the main executable code section. On 32-bit x86 systems, the size of the virtual address space for each process is typically limited to 4GB. With a 5-byte probing instruction (1 byte opcode and 4 bytes offset), a 4-byte offset is sufficient to reach

the remote probe handler. However, on a 64-bit x86 system (x86_64), it may not be large enough to access the probe handler. For this reason, a shared trampoline is allocated somewhere within the program for instance in a long NOP padding. The probing instructions jump to this trampoline, where an indirect absolute *jmp* is located. This is sufficient to transfer control to anywhere in the memory address space, even on 64-bit machines. Using *call*-s as probing instructions to reach a single signal handler, and using a single trampoline if needed, significantly reduce memory usage, the number of mappings and external fragmentation.
When the control flow reaches the probe handler, directly from a call probing instruction or indirectly through the trampoline absolute jump (strategies 1 and 2), or through the trap handler (strategy 3), the context (registers, flags, etc..) is saved on the stack. Thereafter, the instrumentation is called (e.g., for tracing). Finally, the context is restored and the return address is examined, since it serves as a probe identifier. This identifier is used to lookup the corresponding original instructions in a lock-less Red-Black tree [2] that are executed out-of-line. Thereafter, the normal execution is resumed at the instruction following the patched area, through an absolute jump at the end of the out-of-line execution buffer.

Function boundary instrumentation Function boundaries instrumentation may be achieved by inserting a probe at the entry and another probe at the exit(s). If the function exit (typically a *ret*) is at the end of the function address space, the probing instruction can overlap the NOP-padding aligning the next function entry. If this NOP-padding is not sufficiently long, a different approach is taken. Upon function entry, the probe handler saves the return address and replaces it. The replacement value is the address of the function exit probe handler. This ensures that the function exit probe handler is called at function exit. At the end of the function exit probe handler, the saved return address is restored in the call stack and a normal return to the *caller* is performed.

Patching When patching multiple instructions, we must verify if any thread is currently executing, preempted or interrupted within the patching area. Even though existing techniques typically rely on the stop-the-world approach because of the difficulties involved, this is unsatisfactory for many online use cases. Instead, interrupts are used to verify if threads are in that area, and to move them away if it is the case. First, by inserting a trap at the probe site, we ensure that the patching area is locked and no new thread will enter it. If a thread reaches the trap, it will be handled as in Trap-based instrumentation. There may be some threads, however, already within the patching area. An interrupt is thus sent to all threads. The interrupt routine checks if the thread is at a location within the patching area, and if so replaces the address with that of the corresponding instruction in the out-of-line execution buffer.

2. Adapted from the Linux kernel version : linux/lib/rbtree.c

Thereafter, the probing instruction is written to the probe site, except for the first byte to keep the locking trap in place. Before unlocking the patching area, a serializing instruction needs to be executed on each thread, in order to achieve a valid cross-modification for cache straddling original instructions. Two alternative techniques have been implemented. The first is to send an interrupt to each thread with the interrupt handler executing the *CPUID* serializing instruction. As a more efficient alternative, when available, the *membarrier* syscall can synchronize with a serialization instruction all the cores on which the threads of the process are running. Finally, the first byte of the probing instruction is written, replacing the locking trap. As the insertion and removal of single-byte trap instructions are atomic and do not cause coherency issues on the x86 platform, it has been widely used in debuggers and other tools.

4.3.1 Assumptions and Constraints

Reusing the compiler NOP-paddings to insert probing instructions is convenient and frequently possible. Nonetheless, it relies on a few assumptions including :

No return address manipulation by the callee. To achieve function exit instrumentation, by replacing the return address pushed by the *caller* on the stack, we assume that the *callee* function does not interfere with the probe by overwriting the return address. Moreover, the *callee* is not expected to read and find the *caller* return address from the stack. Although it is rare, techniques used by virtual machines like on-stack replacement might cause such interference [73]. Stack unwinding in *C++* is another example where problems could occur; this could easily be solved by hooking the corresponding functions to restore the call stack when an exception is raised.

No data embedding in the NOPs operand. It is assumed that the target program does not use NOP-paddings to embed data. Clearly, manipulating those bytes to insert probing instructions would cause interference. While programs rarely do this, some anti-disassembly and obfuscation techniques touch the paddings to overlap instruction streams, in order to hide some portions of the code in the program [71].

Outside function branch. Injecting a probe for the purpose of dynamic binary instrumentation requires replacing instructions. We thus need to verify if any branching instruction would target one of the overwritten instructions, after the first byte at the probe site. To limit the scope of this verification, only the branching instructions within the instrumented function are examined. It is assumed that there is no branch from another function to the instrumented function, other than its

entry point. This assumption generally holds true in the C/C++ standard.

4.3.2 Static Analysis

Before inserting a probe in a function, a static analysis is performed to check whether a 5-byte *call*, 2-byte *jmp* or *int3* should be used. The multi-byte probing instructions can only be used when no branch is targeting the patching area (except its first byte). If the original instruction at the probe is five bytes long or more, it can easily be patched through the 5-byte *call*. Otherwise, the function is disassembled and the target of every branch is inspected. If no potential branch is found, it can be patched with the 5-byte *call*. Otherwise, if the detected branches do not land on the second byte, and a suitable NOP-padding is available in the vicinity, a 2-byte *jmp* can be patched at the probe site. As of last resort, the probe site can always be instrumented with trap instruction.
To identify the unreachable function paddings inserted between functions, the function entry address and size for each function are extracted from the Executable and Linkable Format (ELF) files to discover the gaps between functions. Moreover, jump paddings may also be unreachable if they follow an unconditional branch while no incoming branches exist.
When an indirect branch (with a register as an operand) is found in a function, we conservatively assume that it could branch anywhere in the function. Thus the search for possible incoming branches will always be affirmative for such functions.

4.3.3 Suitable NOPs

In the second strategy, with the 2-byte *jmp*, it is required to have at least 5 bytes in the padding site to insert a relative *call*. Thus, only 5 bytes or larger NOP-paddings are used. As an exception, there are cases in which the 5 bytes *call* at the padding site overwrites two consecutive NOP instructions

```
0:  0f 1f 84 00     nop  DWORD PTR [rax+rax*1+0x0]
    00 00 00 00
```

(a) Original reachable NOP-padding.

```
0:  0f 1f 84 e8     nop  DWORD PTR [rax+rbp*8+0x4030201]
    01 02 03 04
```

(b) Modified NOP-padding.

Figure 4.7 Embedding a 5-byte *call* in a NOP-padding.

```
0:  0f 1f 80 00 00 00 00    nop    DWORD PTR [rax+0x0]
7:  48 89 c8                mov    rax,rcx
```

(a) Original reachable NOP-padding.

```
0:  eb 04                   jmp    0x6
2:  e8 01 02 03 04          call   0x4030208
7:  48 89 c8                mov    rax,rcx
```

(b) A *jmp* bypassing the NOP-padding followed by an inserted *call*.

Figure 4.8 Inserting a 5-byte *call* in a 7-byte NOP-padding.

sized less than 5 bytes, forming a sufficiently large NOP-padding, as shown in Figure 4.4a. The first block consists of a 4-byte *nopl* and the second block is a 10-byte *nopw*. Therefore, the first entire block (4 bytes) plus the first byte of the second block would be patched.
Any unreachable padding of 5 bytes or more is considered suitable for patching. On the other hand, reachable paddings may only be exploited with extreme caution, since the execution flow may sooner or later reach that point, and overwriting it may alter the program behavior. For the 8, 9, and 10-byte reachable NOP-padding instructions, the probing instruction could be inserted within the unused memory operand, in the last 5 bytes. Therefore, it does not affect the operation (Figure 4.7). The 7-byte reachable padding must be treated differently. Placing a 5-byte *call* on the last 5 bytes of the 7-byte padding, will change the third byte of the NOP-padding (*0x80* indicating addressing mode *Mod R/M*). This modification changes the operation format. As a solution, the 7-byte NOP-padding should be patched with a 2-byte bypass relative *jmp* followed by a 5-byte relative *call* to keep the no-operation behavior untouched (Figure 4.8).

4.3.4 Search and Assignment of Suitable NOPs

Searching for a suitable NOP-padding in memory could be as simple as exploring instructions in the 256-byte range around the probe site. During this discovery phase, every NOP-padding found suitable is added to a list. The list is then fed to an algorithm for choosing the most suitable padding. The algorithm attempts to assign suitable paddings to every probe site in need of padding. This is not feasible in some cases, though. For instance, suppose that function boundaries should be probed and there are two functions *f1* and *f2* and two NOP-paddings *N1* and *N2*. *N1* may appear in the range of both *f1* and *f2*, while *N2* may be reached only by function *f2*. A non-optimal heuristic algorithm could assign *N1* to function *f2* and leaves *f1* unassigned. This underlines the importance of properly assigning NOP-paddings to probe sites. This combinatorial optimization assignment problem deals

with the allocation of multiple resources to the specified sectors on a 1-to-1 basis. This problem could be broadly defined as follows :
Assume that we have N distinct functions available to be instrumented through N suitable NOP-paddings. The list of all paddings reachable from each probe is known. The algorithm is expected to pair each function to one padding.
The definition of the linear assignment problem is formulated as follows :
Given a set of *P* functions and *Q* NOP-paddings, we have a weight matrix $\{c_{ij}\}_{P\times Q}$ where

$$c_{ij} = \begin{cases} 1 & \text{if NOP-padding } j \text{ is in range of function } i; \\ 0 & \text{otherwise.} \end{cases}$$

The algorithm should return a bijection such that the cost function z is maximized :

$$maximize \quad z = \sum_{i=1}^{P}\sum_{j=1}^{Q} c_{ij}x_{ij}$$

The resultant bijective matrix is a binary matrix defined as $\{x_{ij}\}_{P\times Q}$ where

$$x_{ij} = \begin{cases} 1 & \text{if NOP-padding } j \text{ is assigned to function } i; \\ 0 & \text{otherwise.} \end{cases}$$

Finding $\{x_{ij}\}_{P\times Q}$ that maximizes z is equivalent to pairing as many NOP-paddings to functions as possible. The algorithm used to solve this problem is called the Hungarian algorithm [74] that produces an optimal solution. It is a polynomial-time algorithm with a worst-case run-time complexity of $\mathcal{O}(n^3)$, where $n = P + Q$. Feeding the previous set of functions and paddings to this algorithm would result in the following :

	N1	N2
F1	(1)	0
F2	1	(1)

Due to its time complexity, the Hungarian algorithm is preferably used when the number of instrumented functions remains relatively small. The algorithm may be used on a small subset of functions, since each padding is only reachable from one or very few functions. For instance, the algorithm can be used on the instrumented function and its close neighbors to obtain a local optimal assignment.

Operating on all the functions in a binary, to achieve a globally optimal assignment, would be prohibitively expensive when the number of functions is large. For such cases, a greedy approach has

been developed as an alternative algorithm. The greedy algorithm takes as input the list of all the NOP-paddings in the range of the function being instrumented, and generates a non-optimal solution with linear time complexity.

As NOProbe instruments the function entry, any function alignment padding located directly above the entry point obviously lies within the range. If NOProbe uses a 2-byte *jmp* at the probe site, the target padding may be accessed within -128 to 127 bytes of the current location. Hence, the same function alignment padding would likely be out of range of the preceding function entry, unless its size is less than 127 bytes. As a result, the greedy algorithm sets out to first exploit the function paddings, knowing that they will always be within the range of the current function entry, and rarely in the range of other function entries. The list of NOP-paddings is sorted such that the closest ones to the instrumented function are visited first.

Algorithm 2 Greedy algorithm

Input $funcList$: A list of functions near the probe site. ▷
Input $nopList$: A list of potential NOPs sorted to visit the closest paddings to the site first. ▷
Output : A suitable nop if found else $None$ ▷

```
1:  procedure FIRSTSUITABLENOP
2:      for all nop ∈ nopList do
3:          if isFunctionPad(funcList, nop) then
4:              return nop
5:          else if isUnreachable(nop) then
6:              return nop
7:          else if canEmbedTrampoline(nop) then
8:              nop ← setEmbedFlag(nop)
9:              return nop
10:     return None
```

4.3.5 Out-of-Line Execution (OLX)

In the 32-bit x86 platform (x86_32), position-dependent instructions use *EIP* implicitly, without specifying it as an operand. For instance, a Jump Condition Code (JCC) instruction *jae 6* would jump to *IP+6* if the condition has been met. With RIP-relative addressing in the 64-bit x86 platform (x86_64), instructions can explicitly use *[RIP + displacement]* as an address. For example, instruction *mov DWORD PTR [rip+0x1], 0x2* will store the immediate value *0x2* in a double word starting one byte after the end of the *mov* instruction.

Instructions using the program counter (*PC*) implicitly, like unconditional branches (direct relative

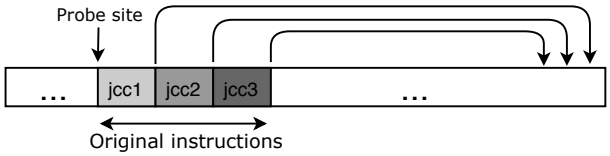
```
0:  e9 12 34 56 78          jmp     0x78563417
```

(a) Before relocation.

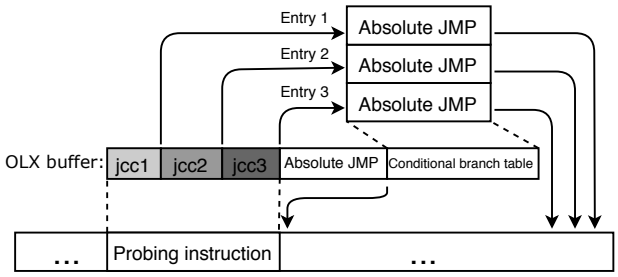
```
0:  ff 25 00 00 00 00       jmp     QWORD PTR [rip+0x0]
6:  00 00 00 00 17 34       data    target address
    56 78
```

(b) After relocation.

Figure 4.9 Relocation of a relative *jmp* instruction.



(a) Before relocation.



(b) After relocation.

Figure 4.10 Relocation of unconditional branches.

jmp in Figure 4.9a) are relocated out-of-line by replacing them with their absolute indirect equivalent instruction, capable of reaching anywhere in memory. The absolute target address of the branch is computed by adding its displacement and size to its location address. As seen in Figure 4.9b, the absolute indirect *jmp* at address *0x0* selects the next 64 bit (*QWORD*) as the target address. The quadword starting at address *0x6* holds the absolute RIP independent target address that is computed by adding the PC [3] value (*0x0* + *0x5*) to the *jmp* displacement (*0x78563412*).
Unlike unconditional branches, conditional branches do not have an equivalent absolute instruction and thus cannot be relocated in the same way. Instead, a branch table is added at the end of the OLX buffer (Figure 4.10). Each entry in the table defines an absolute jump that can reach everywhere. If the conditional jump is not taken, it slips into the next instruction. Otherwise, it would branch to the corresponding entry in the table, leading it to the original destination. The effective address of each entry in the table is computed as for unconditional branches.

In the case of the RIP-relative instructions, using the PC explicitly, the relocation is more complicated. They cannot be substituted as in the former cases. To relocate them, the *ModR/M* field of the original instruction must be modified to hold the computed absolute address, instead of the *PTR [rip + displacement]*. However, since there is already a memory operand, there is no room for the second one. The two-explicit-operand instructions are not allowed to perform memory to memory operations in x86. For this reason, a scratch register is used to hold the address computed by *[PC + displacement]* (Figure 4.11). To avoid clobbering the register, it needs to be saved on the stack prior to being used, and later restored.

```
site+0:  cmp    rdx, QWORD PTR [rip+0x5]
```

(a) Before relocation.

```
olx+0:  push   rax
olx+1:  mov    rax, 0x000000000000000F
olx+b:  cmp    rdx, QWORD PTR [rax]
olx+c:  pop    rax
```

(b) After relocation.

Figure 4.11 Relocation of an RIP-relative instruction.

A suitable general-purpose register, to serve as a scratch register holding the computed RIP independent address, should be chosen with care. Reusing a register that already contains valid data to be used in upcoming instructions may corrupt the program execution. To avoid that, the candidate

3. The program Counter in x86 always points to the next instruction to execute

register should not be used in the upcoming relocated instructions explicitly or implicitly. For instance, instruction *mulx rbx, rcx, QWORD PTR [rip+0x0]* explicitly accesses registers *rbx* and *rcx* and implicitly uses *rdx* as a source operand.

4.3.6 Simultaneous Execution Probing (Lock and Redirect)

As explained in section 4.3, when NOProbe overwrites more than one original instruction at the probe site with a probing instruction, a thread might be entrapped in the middle of the replaced instructions. That concurrent thread could experience undefined behavior if the patching takes place without special care. To prevent this scenario, a trap instruction is first inserted at the probe site, on the first byte of the patching area. Patching or removing a 1-byte trap instruction is a safe atomic operation and does not generate incoherent state on Intel x86 platforms.
Therefore, ahead of writing a multi-byte probing instruction to a patching area that contains two or more instructions, it is necessary to :

1. Ensure that no new thread will enter the patching area.
2. Move out of the patching area the threads that are already there.

Lock and Redirect To achieve the first objective, the original instructions overwritten by the multi-byte probing instruction are first copied to the OLX buffer (Figure 4.12). Then, a trap instruction (*int3*) is inserted into the first byte of the probe site, acting as a temporary Trap-based probe. The main role of the trap instruction is to lock the patching area to redirect all the incoming threads to the OLX buffer. By doing so, we avoid the invasive stop-the-world action that simultaneously stops all the threads.

The second concern is the threads that are currently executing, interrupted, or preempted in the patching area. If those threads resume execution after the area has been patched, they may execute invalid or incorrect instructions. Hence, those threads must not be allowed to remain there during the patching. With signals, the other threads can be interrupted from user-space to execute a signal handler. The instrumentation thread sends an asynchronous real-time POSIX signal to all the other available threads in the process. The signal handler verifies if the program counter of the receiver thread falls within the patching area. If it does, the thread changes its interrupt return address from the patching area to the corresponding original instruction in the OLX buffer. Finally, the signal handler clears a flag for the receiver thread to let the instrumentation thread know that its work was completed. Once all the receiver threads clear their flags, the instrumentation thread resumes execution. Now that no thread remains in the patching area, and no thread can enter into the patching area, the probing instruction can be inserted safely through the load and arm mechanism explained in the next subsection.

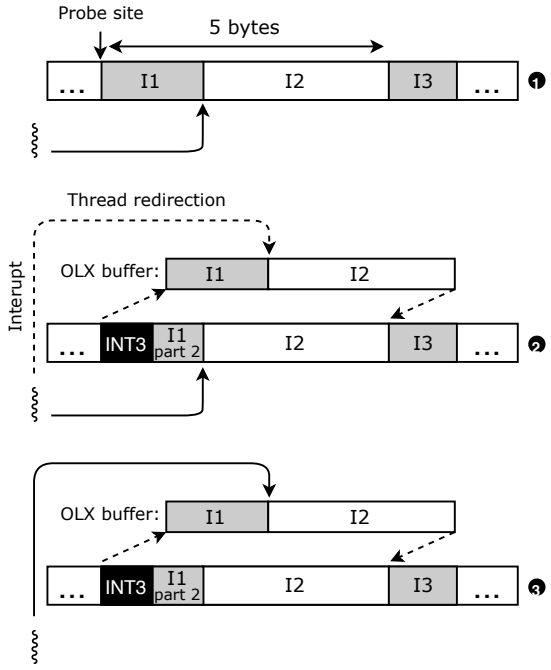


Figure 4.12 Lock the probe site and redirect threads to the OLX buffer.

4.3.7 Cross-modifying Probing (Load and Arm)

Patching several instruction bytes may straddle the cache. This is rarely the case for function entries, since those are typically aligned. For a 5-byte probing instruction, inserted at the function entry, our empirical studies of sample programs *nginx* and *llvm-ar*, compiled with optimizations (GCC option *-O2*), confirm that cache lines would not be straddled for 98.23% and 100% of cases, respectively. Since NOP-paddings are used to align the subsequent instruction (function entry, loop...), they typically do not straddle the cache lines either. However, even if some NOPs do straddle a cache line, there is no need to serialize the execution at this step. Indeed, the replaced NOPs are not

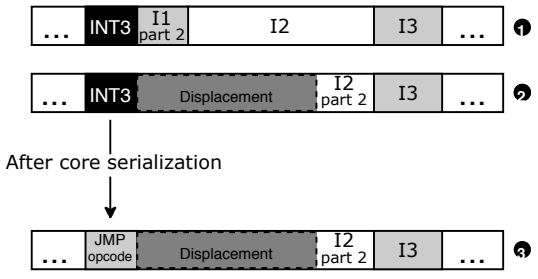


Figure 4.13 Patching the probe site together with serialization.

executed until the 2 bytes *jmp* is patched and serialized.
It is, therefore, a simpler process to patch the instructions that do not straddle the cache line. An atomic store can safely patch the probing instruction, without causing incoherence in the processor core. On the other hand, instructions straddling the cache must account for the Intel erratum, and issue a serialization instruction such as *CPUID, IRET, or RSM* on other processor cores. This ensures that any modifications of former instructions to flags/registers/memory are settled before the fetch and execution of the next instruction in the pipeline.

Load and arm Issuing the serialization instruction must take place before other threads execute the patching area. Thus, after redirecting the incoming threads to the patching area with a trap, and moving away the threads within, the offset of the *call* probing instruction can be patched (Figure 4.13). Then, the *CPUID* serialization instruction is issued either by sending a real-time POSIX signal or by using the *membarrier* syscall [4]. With the *membarrier* syscall, command *MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE* issues a smp_mb() memory fence, via Inter-Processor Interrupt (IPI), to every processor on which the threads of the process are currently running. When returning from the IPI, the Interrupt Return (IRET) instruction performs the needed serialization.
Then, once the pipeline, memory and cache lines of each involved processor core have been serialized, the 1-byte trap instruction at the head of the probe site can be atomically and safely replaced with a 1-byte *call/jmp* opcode. This activates the *call* probing instruction.

4.3.8 Signal Dispatching

Some of the programs to instrument may already be using signals for their own purposes, including *SIGTRAP*. Therefore, a shared signal dispatcher is used. NOProbe saves and replaces the original program trap handler with a shared dispatcher to receive *SIGTRAP* signals when a lock or probing trap is hit. Once a signal is received, the shared signal handler dispatches it to the corresponding handler. If the address of the instruction from which the trap has been raised corresponds to an address where a trap was inserted by the instrumentation program, the instrumentation handler is called. Otherwise, the original program handler is executed.
A shared real-time signal dispatcher is also registered to receive real-time signals, for serialization during the probe patching. Unlike standard signals, the real-time signals can be queued, which is important for signal dispatching purposes. There are two major concerns raised here. First, the target programs might replace the shared dispatcher for their own purposes. To disallow replacing the shared dispatcher calls, *sigaction()* and *signal()* have been intercepted. The intercepting function

4. *Membarrier* has been added recently to the Linux kernel (Linux 4.3). For this reason, signals are used for older kernels.

saves and feeds the probe handler argument to the dispatcher, instead of registering to replace the dispatcher. The second issue comes with the target programs blocking NOProbe from receiving the critical signals. To this end, *sigprocmask()* is intercepted as well. The intercepting function saves the signal mask and feeds it to the shared dispatcher to check if the signal was already blocked or not, before deciding to dispatch it to the program handler. To dispatch the real-time signals, the thread identifier of the sender is verified. If the signal comes from the instrumentation thread, the instrumentation signal handler is invoked. Otherwise, the original program handler is called.

4.4 Results

In this section, the performance overhead of NOProbe (instrumentation and execution cost) is measured by running a specific code of interest repeatedly, and averaging the total execution time over the total execution count. To measure the execution time, we avoided using the *POSIX timer* API. Although it offers good precision, it relies on sequential locks, which is problematic for our multi-thread environment. As a result, the Time Stamp Counter (TSC) was used, along with a memory fence instruction [75], which carries a negligible overhead compared to the total execution time.

Experimental setup The uftrace tracing tool was chosen as an efficient user-space prototyping vehicle for the new algorithms in NOProbe. We partially redesigned uftrace such that it could attach to an existing process and dynamically insert instrumentation. Before that, uftrace preloaded (*LD_PRELOAD*) an instrumentation library called *libmcount* and then forked the target program. This instrumentation library takes care of all the initialization steps, from reading symbol table to creating shared buffers, hooking into functions, instrumenting the target program, and starting the target program execution while tracing. Although the *LD_PRELOAD*-ing technique is a simple and effective approach, it could not enable uftrace to attach to an existing process and dynamically instrument it.
On-the-fly instrumentation of the *tracee* requires *libmcount* to be injected into its address space. To this end, the *ptrace* system call is used for a safe and race-free injection. Once the modified uftrace attaches to a running *tracee*, it pauses the program with the *ptrace* system call. To reduce the time during which the program is paused, a minimal library named *libloader* is injected. That library can thereafter setup and load other libraries without pausing the *tracee* and using the *ptrace* system call. Once injected, *libloader* creates a dormant thread that waits on a Unix socket to receive commands. The tracer program then sends commands to the *libloader* thread to load (*dlopen*) the *libmcount* library, in order to insert instrumentation and start tracing (*start_tracing*).

Table 4.2 displays the specification of the environment in which our experiments were conducted.

Table 4.2 Evaluation Machine Specifications.

Processor	4x Intel Xeon E7-8867 v3 @2.50GHz
# Cores	64 with Hyper-Threading disabled
Operating System	Debian GNU/Linux 9.0 and Linux 4.19.0-6-amd64
Compiler & Libraries	GCC/G++ 8.3.0 and GLIBC 2.28-10
Virtualization	Kernel-based Virtual Machine
Memory	256GB

4.4.1 Instrumentation Cost

Through this section, we conduct three micro-benchmarks to test the performance of the probe instrumentation mechanism. We compare the insertion procedure cost of dyninst, uprobe and NOProbe based on the number of threads, probes, and the function size. Each experiment was setup to measure different aspects of the performance.

Cost vs. #threads In this test, the instrumentation cost is measured while a single thread performs the instrumentation while multiple threads are executing another task. This task is an infinite loop to successively *read* from a file and *write* its content to the standard output. Since the *read* and *write* syscalls are serialized with a lock on the file descriptor, only one I/O operation on a file is executed at a time [76]. This allows us to measure the worst-case patching cost of NOProbe caused by the *Lock and Load* synchronization overhead. the instrumentation thread insert probes in only 1000 function entries and exits to keep the number of probes and function size constant while we vary the number of threads. The cost per instrumentation point is computed by dividing the total time it takes to instrument all the functions by the number of instrumented functions. The test was repeated 100 times to minimize the jitter, and the results were averaged.
We found in our experiment that dyninst attaches very slowly to a target, taking almost 6 seconds. Since it relies on *ptrace* for attaching, all the threads running in the target process will be paused and only resumed after the instrumentation is inserted. Moreover, dyninst consumes all the core resources available in the system (64 cores here) to analyze and instrument the target. During the probe instrumentation measurement, the attachment cost for dyninst was excluded. However, we chose the immediate instrumentation mode over the batch mode. It might be slightly less effective, due to the inherent caching capability of the batch mode that reduces the Inter-Process Communication (IPC) cost [29].
For NOProbe, we break the instrumentation cost into 3 distinct parts :

— The static analysis cost of the greedy algorithm, which includes the NOP-padding search and assignment cost.

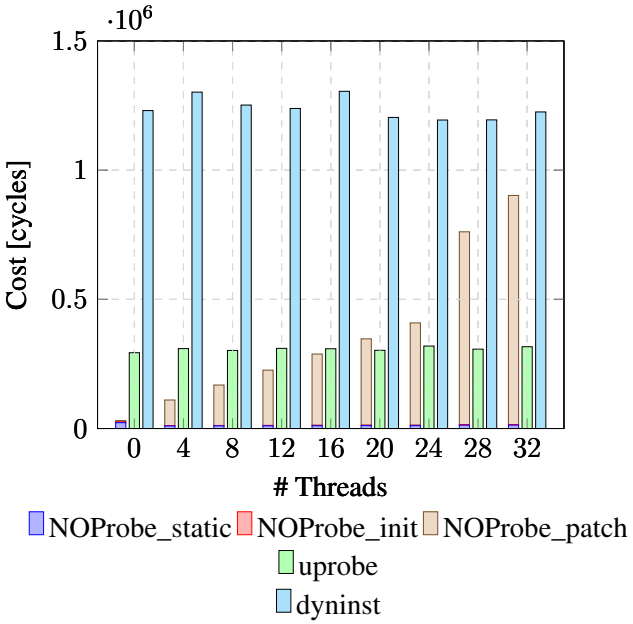


Figure 4.14 Probe insertion cost versus the number of running threads.

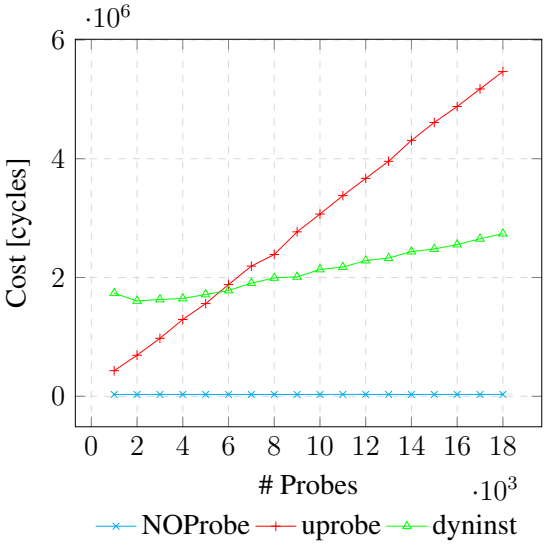


Figure 4.15 Probe insertion cost for uprobe, NOProbe and dyninst versus the number of probes inserted.

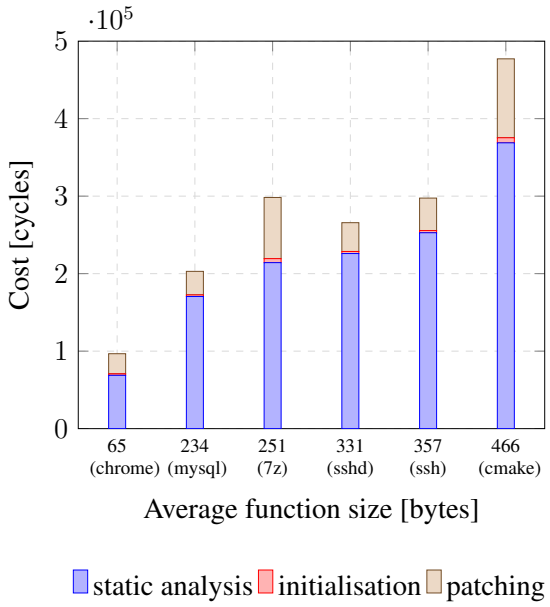


Figure 4.16 Probe insertion cost versus the average function size of multiple binaries.

— The OLX buffer initialization and Red-Black tree insertion cost.
— The *Lock and Load* patching cost (using *membarrier*).

Figure 4.14 shows the result for the probe injection cost. In NOProbe, the instrumentation thread sends an interrupt signal to all threads at the time of patching. It then waits for all the threads to acknowledge the signal before proceeding with the patching. For instance, If a thread is blocked waiting for a resource to be available, the instrumentation thread will spend more time waiting for the acknowledgment. Handling a queued POSIX signal happens once a thread returns from kernel space. Since the locks (used by *read/write* syscalls) that protect kernel resources may keep the signal receiving threads waiting in kernel space, they would not be able to proceed quickly with the signal handler. This case will be even worse when the number of threads increases.
Although the dyninst instrumentation cost is high, it scales well with the number of threads. This efficient scaling is related to the fact that *ptrace* is able to stop all the running threads before patching. The uprobe instrumentation cost is rather stable and outperforms NOProbe when the number of threads exceeds 20. Uprobe operates in the kernel and is not blocked waiting for threads to come back to user-space.

In this experiment, with a recent Linux kernel, we used *membarrier* for serialization. The real-time signals would be used on older kernels. Based upon a modified *lmbench* benchmark, *membarrier* is three times faster than sending and receiving a signal.

Cost vs. #probes The goal of this test is to evaluate the scalability of the instrumentation procedure as the number of inserted probes grows. Figure 4.15 demonstrates the insertion cost of uprobe, NOProbe, and dyninst. The instrumentation cost was assessed for the greedy algorithm for NOProbe, while excluding the attaching cost for dyninst. Here we only used one instrumentation thread to exclude the impact of multithreading from this experiment. For each batch size, from 1k to 18k probes, we generate a program with the same number of empty functions as probes. As in the previous experiment, we instrumented the function boundaries once. The instrumentation cost of all the functions is then divided by the batch size to determine the cost of a single instrumentation. The test is repeated 100 times and the average computed.

Both uprobe and dynisnt scale poorly with the batch size. In the case of uprobe, the *tracefs* filesystem insertion interface adds to the cost and may explain the poor result. In the case of NOProbe, by using a lock-less Red-Black tree to store the probe and the efficient greedy algorithm, the cost of probe insertion scales well with the number of probes.

Cost vs. function size This test evaluates how NOProbe scales with the size of a function in which the probe is inserted. To this end, we measure the average time taken to insert a probe into every function of six typical large applications. Then, we measure the average size of all instrumented functions in each application. Figure 4.16 shows the average insertion cost for NOProbe versus the average function size.

The average function size of all the benchmarked large applications is slightly larger than 200 bytes. For Chrome, however, we filtered out all the functions longer than 100 bytes. All the functions in the other applications were instrumented and the average function size for those is thus unaffected. This allows us to have sample cases with low average function sizes for this experiment.
The static analysis cost rises with the average function size. The static analysis is conducted on every instruction in the functions in order to discover branches to the patching areas.

4.4.2 Execution Overhead

For multiple instrumentation tools, we measured the probe execution overhead, defined as the extra CPU cycles imposed on the normal execution without the probe. We vary the number of simultaneous threads executing the probe and then measure how the overhead scales with the thread count. The entry and exit of an empty function were probed.

Because gdb does not support entry/exit instrumentation natively, we forced the compiler to insert a 5-byte instruction at the entry of the empty function. This enables us to accomplish only function entry instrumentation for gdb. The probe handler used for dyninst, NOProbe and dyntrace is a user-space function performing no operation, since we want to measure separately the cost of having a

probe and not the cost of tracing or other tasks. The situation by default would be different for gdb and uprobe. In gdb, the user cannot override its probe handler, and in uprobe it is not feasible to replace the probe handler from user-space. Therefore, we recompiled gdb and statically disabled the call to its probe handler. We left uprobe intact and the results thus include the extra overhead of saving an event in a per-CPU ring buffer.

We first ran the probed function for each number of threads and for $r = 1,000,000$ times. The total execution time is divided by $r \times \#threads$ to get the average execution time per thread. For each number of threads, the experiment is repeated 100 times to get the mean execution time per thread. Finally, the overhead is obtained by subtracting the execution time of the probe-less empty function from that of the probed one. Figure 4.17 illustrates the execution overhead when multiple threads execute a probe simultaneously.

As expected, the Trap-based approaches in gdb, uprobe, and dyntrace perform poorly compared to the others. In dyntrace, with the Tranch-based approach, the probability of using the trap is approximately 14% [67], affecting its performance but to lesser extent. Uprobe clearly outperforms the two other tools.

It is worth mentioning that gdb-server uses *ptrace* to receive the trap signal and handles it while the target is stopped. Then, gdb resumes the target only after the probe is finished. All the 3 Trap-based tools scale similarly with the number of threads. The increased overhead may be explained by the fact that each tool uses some synchronization mechanism to ensure a coherent execution. To transfer a trap signal to user-space, the kernel code acquires spin-locks and updates atomic reference counters in order to protect the signal handlers' structure.

The Branch-based dyntrace and gdb probes perform better than the Trap-based techniques. However, gdb fails to scale nicely with the number of threads because it acquires an exclusive lock in the trampoline (jump pad), in order to protect the tracing buffer from concurrent *write*-s [77]. Dyninst probes do not need any lock to protect their structure, because the patching procedure is not performed on-the-fly. Indeed, dyninst stops all the threads using a *ptrace* syscall and transfers the running threads out of the critical regions before patching the code.

Although *write*-s to the lock-less Red-Black tree must use costly atomic (lock) instructions, *read*-s are not required to. The NOProbe execution overhead thus scales very well with the number of threads. However, the *read*-s may miss an entire sub-tree of a node that is being atomically modified. Alternatively, a lighter-weight synchronization mechanism like user-space Read-Copy-Update (RCU) could be used which offers low overhead and good scalability. A writer-side will wait until all currently pre-existing reader-side critical sections are completed, before attempting any modification to the tree. Although the reader-side avoids any waiting, a minimal overhead may be added, depending on the library implementation [78].

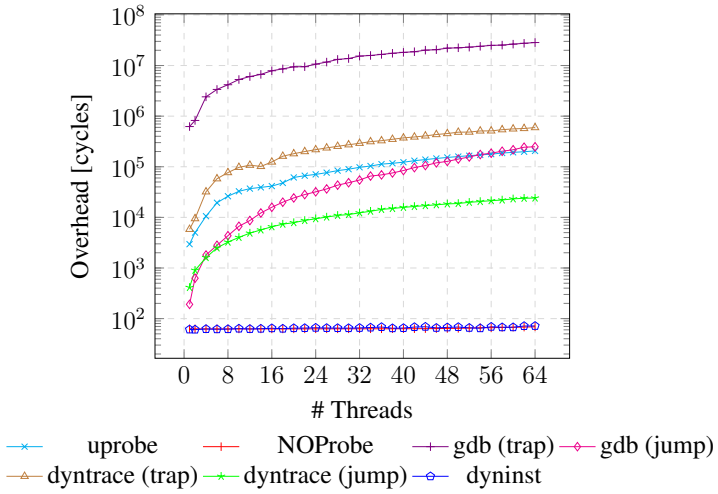
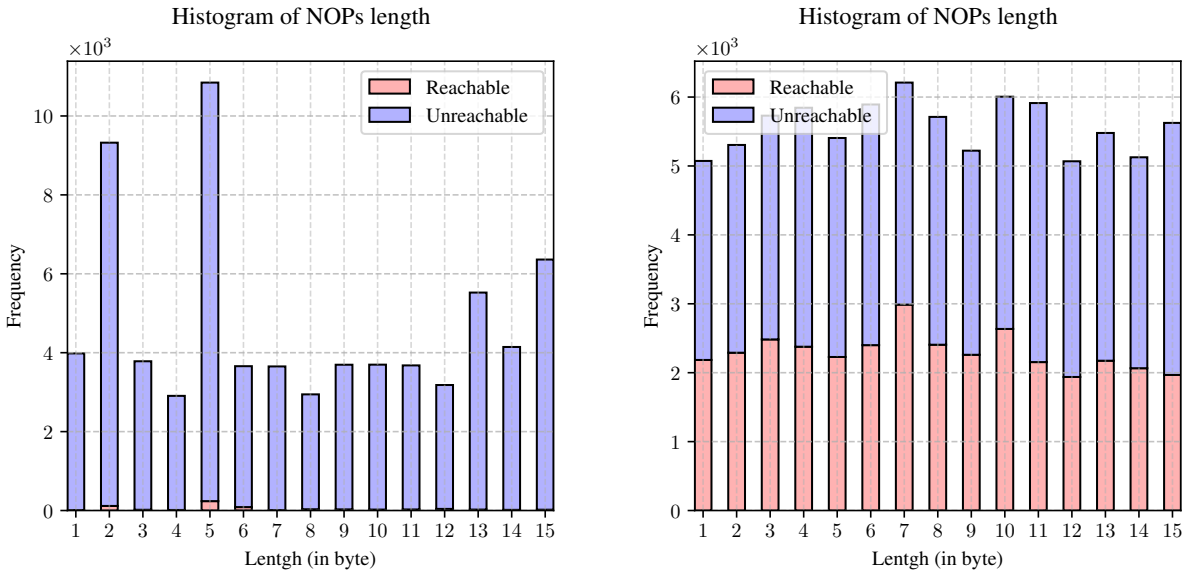


Figure 4.17 Probe execution overhead for uprobe, NOProbe, gdb (trap and fast tracepoint), dyntrace (best case and worst case) and dyninst.

4.4.3 NOPs Distribution Analysis

To demonstrate the interest of exploiting NOP paddings in a binary, we analyzed the distribution of NOPs in two popular large software packages, *llvm-ar* and *clang*, when compiled with *gcc* and *clang*. Figure 4.18 presents the padding length histogram and the proportion of reachable and unreachable paddings. It shows that the paddings in *llvm-ar* and *clang* can be identified as unreachable in 99% and 58.67% of cases, respectively. Such variability is expected since many factors (compiler design, optimization level, coding style, etc..) can affect this rate. NOProbe can use any unreachable padding within the range of the probe site and any reachable padding larger than 6 bytes. Figure 4.18 illustrates the uniform probability distribution of NOPs. In Figure 4.18a, about 51k out of 71k NOPs are 5 bytes long or more. This means that approximately $51k \div 71k = 71.83\%$ of NOPs in *llvm-ar* and $61k \div 83k = 73.74\%$ of NOPs in *clang* can be used by NOProbe. For a binary aligned on 16 bytes, the expected average length can be estimated by computing the length of NOP-paddings based on the number of bytes that are left to reach the alignment boundary. Assuming that the probability distribution for the number of remaining bytes until the block boundary is uniform, the probability of having a 5-byte or longer NOP-padding, that could hold the 5-byte *call*, can be computed as follows :

$$\begin{aligned} P(len \geq 5) &= P(len = 5) + P(len = 6) + \cdots + P(len = 15) \\ &= 1/15 + 1/15 + \cdots + 1/15 \\ &= 11/15 = 0.7333 \end{aligned} \tag{4.1}$$



(a) *llvm-ar* compiled with *gcc -O2*. (b) *clang* compiled with *clang -O3*.

Figure 4.18 The proportion of reachable and unreachable NOPs at different NOPs lengths.

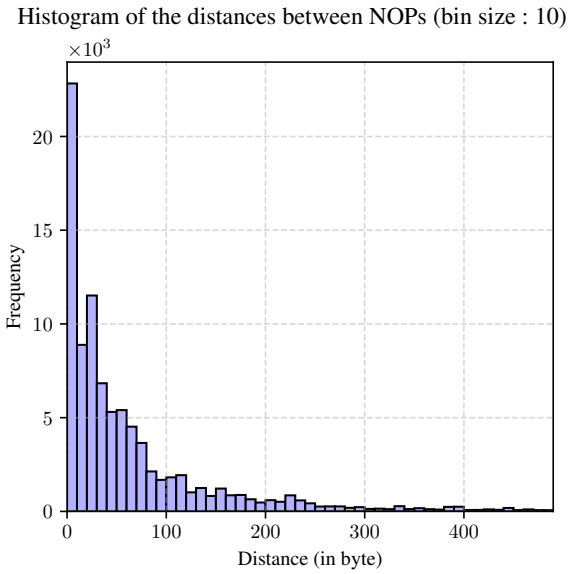


Figure 4.19 The distribution of the distances between NOPs in *llvm-ar*.

Figure 4.19 presents the distribution of distances between NOPs for llvm-ar. Notice that the density of [0-60] bytes interspace is much higher than the others. Indeed, 94.21% for llvm-ar and 77.63% for clang of the distances between NOPs in *llvm-ar* are shorter than 256 bytes. Knowing that a 2-byte *jmp* with a 1 byte displacement has a reachable range of 256 bytes, there is a high chance of finding one of the NOPs close to the probe site.

4.4.4 Probe Effectiveness

In this subsection, we compared the instrumentation success rate of different tools (Branch-based gdb, dyntrace, dyninst, and NOProbe), defined as the number of successful Branch-based (fast) inserted probes divided by the total number of target probe locations. To conduct the experiment and compute the instrumentation success rate, we attempted to insert Branch-based probes at every function entry of five large popular applications (Chrome, LLVM, Git, VIM, and Nginx).
As probes structures might interfere with each other in memory, we decided to insert the probes one by one, keeping them installed until the end of the experiment. We only instrumented the functions residing in the code segment of the main executable, leaving the shared library functions untouched. Table 4.3 describes the instrumentation effectiveness based on the number of successful insertions and the total number of instrumentable locations.
Compilers typically add a prologue to most functions, preparing the registers and stack for the function execution. The prologue typically consists of a sequence of 1 to 3 bytes long instructions. Therefore, the probability of finding a 5-byte instruction at the function entry is very low. This greatly affects the success rate of gdb because it requires at least a 5 bytes instruction in the prologue to employ a Branch-based technique.
Gdb injects a shared dynamic library (In-Process Agent (IPA)) into the target. The IPA initializes all the trampolines (jump pads) and offers an infrastructure for tracing from within the target. In most cases, once a binary has been loaded in Linux, its *.text* section is placed in the lower addresses, far from the shared dynamic libraries. Consequently, in x86_64, a 5-byte *jmp* at a probe site in the *.text*

Table 4.3 Number of successful fast probing across different benchmarks. Success is the number of fast probes inserted successfully. Total is the total number of probe sites and is equivalent to the total number of functions in the instrumented program. Rate is effective fast instrumentation rate computed by dividing Success by Total.

Bench	GDB			NOProbe			Dyntrace			Dyninst		
	Success	Total	Rate	Success	Total	Rate	Success	Total	Rate	Success	Total	Rate
chrome	28982	326205	8.88%	310497	326205	95.18%	-	-	-	320216	326205	98.16%
llvm-ar	523	19940	2.2%	19104	19940	95.80%	9318	19940	46.73%	75539	75539	100%
git	1	9901	0.01%	9815	9901	99.13%	4291	9900	43.34%	9908	9908	100%
vim	0	6396	0%	6279	6396	98.17%	5506	6396	86.08	6403	6403	100%
nginx	137	1148	11.93%	1109	1148	96.60%	1054	1148	91.81%	1154	1154	100%

section may not reach the trampoline in the IPA. While this could be fixed at link-time, by asking the compiler to load the section *.text* close to the IPA, it would defeat the purpose of not needing to rebuild/recompile the executable.
Dyntrace suffers from a similar limitation whenever the *.text* section is loaded in the lowest mappable memory area. If a trap instruction (*0xcc*) was embedded in the fourth byte of a *jmp*'s displacement at the probe site, the offset is negative and dyntrace would be forced to map the probe structure before the *.text* section, which by default is not feasible. This is what happened to many git and llvm-ar functions, resulting in poor instrumentation effectiveness, as revealed in table 4.3.
The probe instrumentation success rate in DynInst is excellent. This was expected since DynInst entirely rewrites and relocates functions. Indeed, for function boundaries instrumentation, DynInst does not deploy the Trap-based technique as it can relocate the entire function to a new location. This success rate and flexibility come at a very high price in terms of attaching cost and memory usage.

4.5 Conclusion

We proposed a new and efficient method to probe and instrument a program entirely from user-space. Without any direct kernel involvement, our probe execution overhead is remarkably low compared to existing techniques. This opens up interesting possibilities for several applications needing dynamic instrumentation and for which the existing tools brought an unacceptable overhead in terms of time or memory, for example in high-performance real-time embedded systems. We exploited NOP-paddings to insert instrumentation code. Moreover, two algorithms for searching and assigning NOP-paddings to probes have been proposed. One emphasizes the performance while the other focuses on the instrumentation success rate. Finally, our improved algorithm based on the *membarrier* system call and the *Lock and Load* mechanism offers a safe code patching technique with reasonable overhead. It can be used for inserting a branch into the code or to replace a whole basic block with another one. In the future, we aim to increase the instrumentation success rate by enhancing NOProbe with other high-performance instrumentation strategies. Further work is also required to achieve safe, scalable and efficient probe removal from user-space.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Retour sur les résultats

Lorsqu'utilisée pour faire le traçage, l'instrumentation du système tracé doit être rapide et performante. Un système tracé doit être observé lors de son exécution normale. Il ne doit donc pas être dérangé par l'exécution du code de l'instrumentation ou par le code du traçage. Malheureusement, le processus de l'instrumentation et du traçage doit interagir avec le système tracé dans le but de mesurer et récolter des informations sur son exécution. NOProbe, en tant que solution d'instrumentation dynamique pour l'espace utilisateur en x86, permet au code de traçage de s'exécuter quelques nanosecondes après l'entrée du point de trace. Cela réduit largement l'impact de l'instrumentation sur la trace du système. Le temps d'exécution de l'instrumentation n'est pas la seule métrique importante. En effet, le cout d'insertion de l'instrumentation peut aussi déranger le système tracé. Certains outils d'instrumentation arrêtent complètement le système tracé pour quelques seconds, et le relancent une fois l'instrumentation terminée. NOProbe peut insérer l'instrumentation durant l'exécution et sans arrêt. Cela est très utile pour instrumenter des systèmes qui ne doivent pas être arrêtés (p. ex. serveur, solution de virtualisation, système d'exploitation, etc.). L'insertion ou la modification de code durant l'exécution, avec le moins d'intrusion possible, pourraient aussi être utilisées pour installer des mises à jour de sécurité, sans redémarrer un système informatique

5.2 Limitations de la solution proposée

Dans le cas où il existe un saut vers la région modifiée par l'instruction d'instrumentation de 5 octets, la deuxième stratégie est vérifiée. Si un saut existe vers la région modifiée par l'instruction d'instrumentation de 2 octets, ou si aucun rembourrage n'existe aux alentours, une interruption logicielle doit être utilisée. Malgré le fait que cette situation ne se produise seulement dans 5% des cas de nos expériences, une interruption logicielle dans un chemin répétitif (comme au début d'une boucle) risque d'impacter sérieusement la performance.

Notre solution peut modifier un rembourrage utilisé par une autre technique, interférant ainsi avec elle. Cela peut causer une altération du flot de contrôle, ou pire une corruption des données. Des techniques d'obscurcissement de code peuvent se servir des instructions NOP pour cacher du code dans leurs opérandes. Cependant, ce cas survient rarement.

Durant l'utilisation de l'algorithme d'insertion simultané (*lock and load*), le fil d'exécution qui fait l'instrumentation envoie un signal temps réel POSIX à tous les autres fils du processus. Il attend ensuite une réponse de chaque fil avant de continuer l'instrumentation. Si un fil d'exécution prend

trop de temps à répondre, il peut ralentir l'instrumentation. Cette situation peut se produire lorsqu'un fil passe trop de temps dans le noyau et passe rarement en mode utilisateur pour traiter le signal. Un autre exemple peut se produire lorsqu'une application crée et supprime des fils d'exécution fréquemment. En effet, si un fil est supprimé juste après l'envoi du signal, il ne pourra pas répondre. Le fil d'instrumentation perdra du temps en attente avant de se rendre compte que le fil en question a été supprimé.

CHAPITRE 6 CONCLUSION

Pour conclure ce mémoire, nous présenterons une synthèse des contributions apportées au traçage et à l'instrumentation dynamique. Nous parlerons aussi de certaines limitations de notre technique et nous discuterons de quelques pistes pour des améliorations futures.

6.1 Synthèse des travaux

Nos contributions et notre recherche ont permis d'avancer l'état des connaissances dans le domaine du traçage et de l'instrumentation dynamique. La contribution générale de ce mémoire est une nouvelle technique d'instrumentation dynamique rapide et efficace pour l'espace utilisateur en *x86*. Les détails de la contribution sont :

— *Lock and Load*. Un algorithme d'insertion de code simultané qui peut modifier partiellement ou complètement un bloc de base sans avoir à arrêter l'exécution du programme.
— *NOProbe*. Une technique d'instrumentation dynamique à multiples stratégies. Elle permet d'exploiter les rembourrages ajoutés par le compilateur comme espace ou insérer du code. Elle utilise l'algorithme *Lock and Load* et elle évite les mécanismes de synchronisation lourds. Cela la rend très rapide, comparée aux techniques existantes.
— Deux algorithmes pour maximiser l'exploitation des rembourrages, l'un est utilisé pour sa performance et l'autre pour son optimalité.

En conclusion, le surcoût de notre point de trace est considérablement plus bas que celui des techniques existantes. Notre solution peut instrumenter les programmes à plusieurs fils d'exécution de façon sécuritaire et sans impacter la performance. Elle peut notamment être utilisée dans le traçage et la surveillance des systèmes parallèles. Avec son poids léger et sa faible consommation de mémoire, notre technique pourrait même être examinée pour une utilisation dans des systèmes avec des ressources limitées (systèmes embarqués). L'utilisation de l'algorithme *Lock and Load* ne se limite pas à l'insertion du code d'instrumentation. En effet, des techniques de modification de code sont souvent utilisées dans le domaine de la sécurité (obscurcissement du code). Dans un cas moyen, il offre un surcoût très bas comparé aux techniques existantes.

6.2 Améliorations futures

L'enlèvement d'un point de trace dynamique est aussi important que l'insertion. Parmi les améliorations futures, nous cherchons à enlever un point de trace et libérer sa structure efficacement.

Pour cela, nous devons nous assurer qu'il n'est pas sur le point d'être exécuté. La solution standard se base sur un compteur atomique pour savoir si un fil est en train d'exécuter le point de trace. Cependant, effectuer une opération CAS à chaque exécution du point de trace est trop coûteux. Des mécanismes de synchronisation légers, tels que RCU pour l'espace utilisateur pourraient être examinés.

Dans le futur, nous aimerions ajouter plus de stratégies d'instrumentation rapide. Cela diminuera la nécessité d'utiliser les interruptions logicielles comme dernier recours. On pourrait, par exemple, utiliser un saut relatif de deux octets pour sauter vers un bloc de base dans les alentours. Le bloc de base serait déménagé pour l'exécuter en mode hors-ligne. Il serait ensuite remplacé par une instruction de saut absolu ou relatif de cinq octets qui peut atteindre le code de traçage. Cela requiert un mécanisme pour identifier la source du branchement qui atterrit sur le code du traçage (le saut de deux octets du point de trace ou une exécution normale).

Il est aussi possible d'améliorer l'algorithme *lock and load*. En effet, à la place d'utiliser le signal pour faire sortir les fils d'exécution de la zone à modifier, on pourrait attendre qu'ils sortent par eux-mêmes. Pour cela, l'appel système *membarrier* pourrait être utilisé dans le noyau Linux préemptif. Avec la bonne commande, cet appel système appelle *synchronise_sched()* dans le noyau Linux préemptif. Cependant, cette solution risque d'avoir un surcoût similaire à la solution proposée.

RÉFÉRENCES

[1] "Intel® 64 and ia-32 architectures optimization reference manual (2019)." [En ligne]. Disponible : https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[2] "Intel® 64 and ia-32 architectures software developer's manual," vol. 3A : System Programming Guide, Part 1, 2016. [En ligne]. Disponible : https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[3] "Re : Hitachi djprobe mechanism," 2005. [En ligne]. Disponible : https://sourceware.org/legacy-ml/systemtap/2005-q3/msg00208.html

[4] B. Chamith, B. J. Svensson, L. Dalessandro et R. Newton, "Instruction punning : lightweight instrumentation for x86-64," 06 2017, p. 320–332.

[5] S. Rostedt, "Using the trace_event() macro," 2020, en ligne : https://lwn.net/Articles/379903/.

[6] M. Desnoyers, "Low-impact operating system tracing," Thèse de doctorat, École Polytechnique de Montréal, 2009.

[7] LTTng, "The lttng documentation," 2019, en ligne : https://lttng.org/docs/v2.11/#doc-instrumenting-linux-kernel.

[8] D. Pacheco, "Usdt providers redux," 2011, en ligne : https://dtrace.org/blogs/dap/2011/12/13/usdt-providers-redux/.

[9] M. Fleming, "Using user-space tracepoints with bpf," 2018, en ligne : https://lwn.net/Articles/753601/.

[10] Dtrace, General Commands Manual, 2019, tiré des pages de manuel de Linux.

[11] T. Iskhodzhanov, R. Kleckner et E. Stepanov, "Combining compile-time and run-time instrumentation for testing tools," n°. 3, 2013.

[12] LLVM, "Xray instrumentation," 2016, en ligne : https ://llvm.org/docs/XRay.html.

[13] E. A. N. H. Alistair Veitch, Dean Berris et N. Wang, "Xray : A function call tracing system," 2016.

[14] GNU, "Implementation of profiling," 2020, en ligne : https ://sourceware.org/binutils/docs/gprof/Implementation.html.

[15] S. Goswami, "An introduction to kprobes," 2020, en ligne : https://lwn.net/Articles/132196/.

[16] M. Hiramatsu et S. Oshima, "Djprobe—kernel probing with the smallest overhead," 2007.

[17] M. Hiramatsu, “The enhancement of kernel probing-kprobes jump optimization,” 2016.

[18] P. S. P. Jim Keniston et M. Hiramatsu, “Kernel probes (kprobes),” 2020, en ligne : https://www.kernel.org/doc/Documentation/kprobes.txt.

[19] J. Sun, Z.-h. Li, X. Zhang, Q.-l. He et H. Wang, “The study of data collecting based on kprobe,” dans 2011 Fourth International Symposium on Computational Intelligence and Design, vol. 2. IEEE, 2011, p. 35–38.

[20] S. Dronamraju, “Uprobe-tracer : Uprobe-based event tracing,” 2020, en ligne : https ://www.kernel.org/doc/Documentation/trace/uprobetracer.txt.

[21] S. Shebs, “Gdb tracepoints, redux,” Polytechnique, Montréal, Quebec, Canada, 2009, p. 105 – 112. [En ligne]. Disponible : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.739.1074&rep=rep1&type=pdf

[22] P. Kankowski, “x86 machine code statistics,” 2009, en ligne : https ://www.strchr.com/x86_machine_code_statistics.

[23] C. Harper-Cyr, M. R. Dagenais et A. S. Bushehri, “Fast and flexible tracepoints in x86,” Software : Practice and Experience, vol. 49, n°. 12, p. 1712–1727, 2019. [En ligne]. Disponible : https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2746

[24] B. Chamith, B. J. Svensson, L. Dalessandro et R. R. Newton, “Living on the edge : Rapid-toggling probes with cross-modification on x86,” dans Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’16. New York, NY, USA : Association for Computing Machinery, 2016, p. 16–26. [En ligne]. Disponible : https://doi.org/10.1145/2908080.2908084

[25] N. Nethercote et J. Seward, “Valgrind : A framework for heavyweight dynamic binary instrumentation,” dans Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI ’07. New York, NY, USA : Association for Computing Machinery, 2007, p. 89–100. [En ligne]. Disponible : https://doi.org/10.1145/1250734.1250746

[26] ——, “Valgrind : A program supervision framework,” Electronic notes in theoretical computer science, vol. 89, n°. 2, p. 44–66, 2003.

[27] Valgrind, “About valgrind,” 2020, en ligne : https://valgrind.org/info/about.html.

[28] D. Bruening et S. Amarasinghe, “Efficient, transparent, and comprehensive runtime code manipulation,” Thèse de doctorat, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[29] B. Buck et J. K. Hollingsworth, “An api for runtime code patching,” The International Journal of High Performance Computing Applications, vol. 14, n°. 4, p. 317–329, 2000.

[30] Paradyn, “Patchapi programmer’s guide,” Rapport technique, 2016.

[31] A. R. Bernat et B. P. Miller, “Anywhere, any-time binary instrumentation,” dans Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, 2011, p. 9–16.

[32] J. Arnold et M. F. Kaashoek, “Ksplice : Automatic rebootless kernel updates,” dans Proceedings of the 4th ACM European conference on Computer systems, 2009, p. 187–198.

[33] F. S. Foundation, “Debugging with gdb,” 2015, en ligne : https://sourceware.org/gdb/current/onlinedocs/gdb.html.

[34] B. Cantrill, M. W. Shapiro, A. H. Leventhal et al., “Dynamic instrumentation of production systems.” dans USENIX Annual Technical Conference, General Track, 2004, p. 15–28.

[35] B. Cantrill et M. Shapiro, “Dtrace : Dynamic tracing for solaris,” 2002, en ligne : http://dtrace.org/resources/bmc/dtrace_ktd.pdf.

[36] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston et J. Chen, “Locating system problems using dynamic instrumentation,” dans 2005 Ottawa Linux Symposium. Citeseer, 2005, p. 49–64.

[37] M. Desnoyers, “Lttng-ust vs systemtap userspace tracing benchmarks,” 2011, en ligne : https://sourceware.org/legacy-ml/systemtap/2011-q1/msg00247.html.

[38] Strace, “Linux programmer’s manual,” 2020, tiré des pages de manuel de Linux.

[39] A. N. Burton et P. H. Kelly, “Workload characterization using lightweight system call tracing and reexecution,” dans 1998 IEEE International Performance, Computing and Communications Conference. Proceedings (Cat. No. 98CH36191). IEEE, 1998, p. 260–266.

[40] B. Gregg, “Linux extended bpf (ebpf) tracing tools,” 2020, en ligne : http://www.brendangregg.com/ebpf.html.

[41] S. McCanne et V. Jacobson, “The bsd packet filter : A new architecture for user-level packet capture.” dans USENIX winter, vol. 46, 1993.

[42] iovisor, “Userspace ebpf vm,” 2020, en ligne : https://github.com/iovisor/ubpf.

[43] M. Fleming, “A thorough introduction to ebpf,” 2017, en ligne : https://lwn.net/Articles/740157/.

[44] G. Brendan, “Perf,” 2019, en ligne : http://www.brendangregg.com/perf.html.

[45] P. Guide, “Intel® 64 and ia-32 architectures software developer’s manual,” Volume 3B : System programming Guide, Part 2.

[46] A. Kleen, “Advanced usage of last branch records,” 2016, en ligne : https://lwn.net/Articles/680996/.

[47] GNU, “a test coverage program,” 2020, en ligne : https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[48] H. Blasum, F. Görgen et J. Urban, “Gcov on an embedded system,” GCC for Research in Embedded and Parallel Systems, 2007.

[49] J. Fenlason, “Gnu’s bulletin, vol. 1 no. 5, june, 1988,” 1988, en ligne : https ://www.gnu.org/bulletins/bull5.html.

[50] D. Mathieu, M. Paul, E., S. Alan, S., D. Michel, R. et W. Jonathan, “User-level implementations of read-copy update,” 2010.

[51] P. E. McKenney et J. D. Slingwine, “Read-copy update : Using execution history to solve concurrency problems,” 2002.

[52] S. Rostedt, “Ftrace linux kernel tracing,” dans Linux Conference Japan, 2010.

[53] T. Bird, “Measuring function duration with ftrace,” dans Proceedings of the Linux Symposium. Citeseer, 2009, p. 47–54.

[54] S. Rostedt, “Ftrace kernel hooks : More than just tracing,” 2015, en ligne : https://blog.linuxplumbersconf.org/2014/ocw/system/presentations/1773/original/ftrace-kernel-hooks-2014.pdf.

[55] R. Steven, “Debugging the kernel using ftrace,” 2009, en ligne : https://lwn.net/Articles/365835/.

[56] N. Kim, “A function (graph) tracer for c/c++ userspace programs,” 2020, en ligne : https://github.com/namhyung/uftrace.

[57] K. Namhyung, “Uftrace updates,” 2019, en ligne : https://tracingsummit.org/ts/2019/files/Tracingsummit2019-uftrace-kim.pdf.

[58] A. Jaleel, R. S. Cohn, C.-K. Luk et B. Jacob, “Cmp $ im : A pin-based on-the-fly multi-core cache simulator,” dans Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA, 2008, p. 28–36.

[59] J. Poimboeuf, “Introducing kpatch : dynamic kernel patching,” Red Hat, Rapport technique, 2014.

[60] T. Lindholm, F. Yellin, G. Bracha et A. Buckley, The Java Virtual Machine Specification, Java SE 7 Edition, 1er éd. Addison-Wesley Professional, 2013.

[61] B. Gregg et J. Mauro, DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD, 1er éd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2011.

[62] S. S., “Gdb tracepoints, redux,” Proceedings of the GCC Developers’ Summit.

[63] A. Mavinakayanahalli, P. Panchamukhi, J. Keniston, A. Keshavamurthy et M. Hiramatsu, “Probing the guts of kprobes,” dans Linux Symposium, vol. 6, 2006, p. 5.

[64] A. R. Bernat et B. P. Miller, “Anywhere, any-time binary instrumentation,” dans Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, ser. PASTE ’11. New York, NY, USA : Association for Computing Machinery, 2011, p. 9–16. [En ligne]. Disponible : https://doi.org/10.1145/2024569.2024572

[65] K. Peter, “x86 machine code statistics,” 2009. [En ligne]. Disponible : https://www.strchr.com/x86_machine_code_statistics

[66] C. Harper-Cyr, M. R. Dagenais et A. S. Bushehri, “Fast and flexible tracepoints in x86,” Software : Practice and Experience, vol. 49, n°. 12, p. 1712–1727, 2019. [En ligne]. Disponible : https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2746

[67] B. Chamith, B. J. Svensson, L. Dalessandro et R. R. Newton, “Instruction punning : Lightweight instrumentation for x86-64,” SIGPLAN Not., vol. 52, n°. 6, p. 320–332, juin 2017. [En ligne]. Disponible : http://doi.acm.org/10.1145/3140587.3062344

[68] M. Hiramatsu et S. Oshima, “Djprobe kernel probing with the smallest overhead,” dans Linux Symposium, 2007, p. 189.

[69] “[patch] linux kernel markers,” 2006. [En ligne]. Disponible : https://linux.kernel.narkive.com/FDc9TB0d/patch-linux-kernel-markers

[70] C. LeDoux, M. Sharkey, B. Primeaux et C. Miles, “Instruction embedding for improved obfuscation,” dans Proceedings of the 50th Annual Southeast Regional Conference, ser. ACM-SE ’12. New York, NY, USA : Association for Computing Machinery, 2012, p. 130–135. [En ligne]. Disponible : https://doi.org/10.1145/2184512.2184543

[71] C. Jämthagen, P. Lantz et M. Hell, “A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries,” dans 2013 Workshop on Anti-malware Testing Research, Oct 2013, p. 1–9.

[72] D. M. Berris, A. Veitch, N. Heintze, E. Anderson et N. Wang, “Xray : A function call tracing system,” Rapport technique, 2016, a white paper on XRay, a function call tracing system developed at Google.

[73] S. J. Fink et Feng Qian, “Design, implementation and evaluation of adaptive recompilation with on-stack replacement,” dans International Symposium on Code Generation and Optimization, 2003. CGO 2003., March 2003, p. 241–252.

[74] H. W. Kuhn, “The hungarian method for the assignment problem,” Naval research logistics quarterly, vol. 2, n°. 1-2, p. 83–97, 1955.

[75] Gabriele Paoloni, Intel Corporation, How to Benchmark Code Execution Times on Intel® IA-32 and IA-6
2010, nº. 324264-001.

[76] corbet, “write(), thread safety, and posix.” [En ligne]. Disponible : https://lwn.net/Articles/180387/

[77] D. Nadeau, N. Ezzati-Jivan et M. R. Dagenais, “Efficient large-scale heterogeneous debugging using dynamic tracing,” Journal of Systems Architecture, vol. 98, p. 346 – 360, 2019. [En ligne]. Disponible : http://www.sciencedirect.com/science/article/pii/S1383762118301838

[78] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead,” ACM Comput. Surv., vol. 51, nº. 2, p. 26 :1–26 :33, mars 2018. [En ligne]. Disponible : http://doi.acm.org/10.1145/3158644