POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Points de Trace Dynamiques, Portables et Extensibles en Espace Utilisateur avec un Surcoût Minimal et une Couverture Maximale

OLIVIER DION
Département de génie informatique et génie logiciel

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

Avril 2023

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Points de Trace Dynamiques, Portables et Extensibles en Espace Utilisateur avec un Surcoût Minimal et une Couverture Maximale

présenté par **Olivier DION**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

François-Raymond BOYER, président
Michel DAGENAIS, membre et directeur de recherche
Tarek OULD BACHIR, membre

DÉDICACE

Je dédicace ce mémoire à ma mère qui a su prendre soin de moi.

Je dédicace ce mémoire à ma soeur et son ami pour m'avoir encouragé dans mes choix académiques.

Je dédicace ce mémoire à ma conjointe qui a su être patiente avec mes longues heures de travail le soir.

Je dédicace ce mémoire à la communauté du logiciel libre pour le partage de leur savoir.

REMERCIEMENTS

Un remerciement pour Michel Dagenais qui a été un excellent directeur de recherche. Il a su me conseiller, m'orienter et m'écouter lorsque j'en ai eu besoin. Je le remercie aussi grandement de m'avoir offert une recherche stimulante.

Un remerciement à Mathieu Desnoyers avec qui j'ai eu de longues discussions techniques qui ont propulsé ma recherche.

Un remerciement à Jean-Olivier Dalphond qui a fait la correction de ce mémoire et l'a rendu lisible à tous.

Un remerciement à tout ceux et celles avec qui j'ai discuté et qui ont su m'écouter. Je pense notamment à Clément et Mohammad et bien d'autres.

Enfin, un gros remerciement à ceux et celles qui prennent de leur temps pour lire ce mémoire.

RÉSUMÉ

L'instrumentation d'un logiciel fait partie de la boîte à outils de l'ingénieur informatique. Ses principales utilisations sont l'évaluation des performances d'un logiciel et la génération d'évènements, dérivés de l'état du programme, en minimisant l'effet de sonde. Deux types d'instrumentation existent. L'instrumentation dite statique est déterminée au moment de la compilation du programme et requiert une recompilation de ce dernier pour être modifiée. Elle contraste avec l'instrumentation dite dynamique qui permet d'être modifiée sans recompilation du programme. Cela a le grand avantage de pouvoir instrumenter un logiciel en production, sans avoir accès à son moteur de production.

Or, l'instrumentation dynamique – quoique présente dans le noyau de Linux – est peu présente en espace utilisateur. L'état de l'art actuel propose plusieurs algorithmes – tels que le calembourage d'instructions – permettant d'effectuer une instrumentation dynamique qui minimise les coûts associés à celle-ci sur le programme instrumenté, tout en maximisant sa couverture. Cependant, il n'existe pas d'implémentation robuste basée sur de tels algorithmes qui puisse être utilisée sur des systèmes réels. L'implémentation la plus sérieuse se nomme Dyninst et n'implémente pas ces algorithmes, son paradigme d'instrumentation dynamique étant différent.

L'objectif de ce travail est donc de mettre au point de nouveaux algorithmes améliorés, et d'implémenter ceux-ci dans une bibliothèque en espace utilisateur qui unifie, améliore et complète les algorithmes existants en matière d'instrumentation dynamique, en offrant une API qui abstrait les détails architecturaux. Les sous-objectifs sont de maximiser la couverture de l'instrumentation, minimiser le surcoût spatio-temporel associé à l'instrumentation sur le programme, et de garantir l'exactitude de l'instrumentation, c.-à-d. que celle-ci ne brise pas le programme.

L'implémentation qui en résulte, nommée Libpatch, a été comparée à Dyninst. Sur le plan de la couverture, Dyninst est en théorie capable d'atteindre 100%, car celui-ci déplace et augmente des flots de contrôle complets du programme. Libpatch, quant à lui, offre une couverture globale de 99.1%, tout en étant beaucoup moins intrusif. Cela favorise Libpatch sur le plan du surcoût spatial en utilisant beaucoup moins de mémoire physique – près de 10 fois moins – que Dyninst, un aspect critique pour les sytèmes embarqués. De plus, le surcoût temporel lié à la modification du programme pour installer ou enlever l'instrumentation est beaucoup moins important pour Libpatch, car ce dernier n'interrompt pas l'ensemble des fils d'exécutions du programme, contrairement à Dyninst. Enfin, les surcoûts temporels, liés

à l'exécution de l'instrumentation, sont du même ordre de grandeur et Libpatch offre une grande capacité de mise à l'échelle, par rapport au nombre de fils d'exécution.

Finalement, Libpatch serait en mesure de bien s'intégrer au sein d'outils tels que GDB et Uftrace, pour augmenter le taux de couverture de leur instrumentation, mais aussi les accélérer. D'autres applications sont aussi possibles, telles que la vérification des accès en mémoire et l'application de correctifs liés à des failles de sécurité.

ABSTRACT

Software instrumentation is a technique increasingly used in software development to determine performance bottlenecks and problems related to the execution of a program. Instrumenters are optimized for generating events derived from the state of the program, while minimizing their impact. Developers often use static instrumentation, where its specification is determined at compile time and is fixed in time. This contrasts with dynamic instrumentation which offers a more flexible specification that can be remodeled after the compilation, allowing the instrumentation of production ready software, without the need for recompilation.

Dynamic instrumenters, like Kprobe, are already extensively used in the Linux kernel. There is a lack of such flexible and efficient instrumenters in userspace. There are of course tools like Uprobe that can be used, but their runtime impact on the program are often too important. Recent advances have been made in the literature, presenting new techniques such as instruction punning, which can help in doing userspace dynamic instrumentation. There is, however, no real robust implementation available. The most notable one is Dyninst and it does not implement the latest algorithms, such as instruction punning, since its instrumentation paradigm differs.

Therefore, the objective is to design new improved algorithms for efficient and flexible dynamic instrumentation, implementing a userspace library that unifies the latest algorithms in the literature while offering a portable API, abstracting away the architecture dependent details. The sub-objectives are maximizing the instrumentation coverage, minimizing the runtime overhead imposed by the instrumentation, and insuring its portability and correctness.

The resulting artifact is a new library named Libpatch and was compared against Dyninst. Theoretically, Dyninst can achieve 100% coverage, since it rewrites the control flow graph of the program. Libpatch is able to achieve near 99.1% global coverage while being less intrusive. This also means that Libpatch uses way less physical memory than Dyninst, a considerable advantage on embedded systems. Furthermore, the modifications made by Lipbatch impose much less time overhead on the program than Dyninst, thanks to its cross-modifying protocol. Moreover, the runtime overhead of executing Libpatch instrumentation is similar, if not better than Dyninst.

Finally, Libpatch can be used to improve many tools that rely on dynamic instrumentation, such as GDB and Uftrace, for increasing their instrumentation coverage and reducing their

runtime overhead. Other applications are possible as well, such as runtime verification of memory accesses, or hot patching of software vulnerabilities.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xi

LISTE DES FIGURES xii

LISTE DES SIGLES ET ABRÉVIATIONS xiv

CHAPITRE 1 INTRODUCTION 1
1.1 Définitions 3
1.2 Objectifs de recherche 4
1.3 Plan du mémoire 6

CHAPITRE 2 REVUE DE LITTÉRATURE 7
2.1 Points de trace par interruption logicielle 7
2.1.1 Uprobe 7
2.2 Accélération des points de trace dynamiques 8
2.2.1 Réécriture du flot de contrôle du programme 8
2.2.2 Branchements dynamiques 10

CHAPITRE 3 DOMAINE DE LA PROBLÉMATIQUE 28

CHAPITRE 4 ARTICLE 1 : DYNAMIC PATCHING OF BINARIES IN USERSPACE 30
4.1 Introduction 30
4.2 Related Work 31
4.2.1 Dyninst 32
4.2.2 Kprobe 32
4.2.3 Dyntrace 32

4.2.4 Liteinst . . . 33
4.2.5 E9Patch . . . 34
4.2.6 NOProbe . . . 34
4.3 Problem Domain . . . 35
4.4 Design and Implementation . . . 35
4.4.1 Patch flow . . . 36
4.4.2 Jump pattern algorithms . . . 37
4.4.3 Patching protocol . . . 38
4.4.4 Trampoline allocator . . . 42
4.4.5 Trampoline instructions . . . 44
4.4.6 Handlers . . . 46
4.4.7 OLX buffers . . . 50
4.4.8 Memory reclamation . . . 51
4.4.9 Runtime cooperation . . . 52
4.5 Interaction with security measures . . . 52
4.6 Experiments/Results . . . 53
4.6.1 Reproductibility . . . 53
4.6.2 Coverage . . . 54
4.6.3 Memory usage . . . 56
4.6.4 Throughput . . . 57
4.6.5 Micro-benchmarks . . . 61
4.7 Conclusion . . . 66
4.8 Acknowledgements . . . 67
4.9 Appendix . . . 68
4.9.1 Artefact . . . 68

CHAPITRE 5 DISCUSSION GÉNÉRALE . . . 69

CHAPITRE 6 CONCLUSION . . . 71
6.1 Synthèse des travaux . . . 71
6.2 Limitations des solutions proposées et améliorations futures . . . 72

RÉFÉRENCES . . . 73

LISTE DES TABLEAUX

Table 4.1	Coverage results.	55
Table 4.2	Coverage averages and standard deviations.	55
Table 4.3	Algorithm chosen (sorted by attempt).	56
Table 4.4	Top 20 unpatchable regions.	56

LISTE DES FIGURES

Figure 2.1 Accès à une variable globale en C. 12
Figure 2.2 Accès à une variable globale en x86-64. 12
Figure 2.3 Programme vide en C. 14
Figure 2.4 Programme vide en assembleur. 14
Figure 2.5 Programme vide compilé avec -pg. 15
Figure 2.6 Programme vide compilé avec -pg -mnop-mcount. 15
Figure 2.7 Programme vide compilé avec -finstrument-functions. 16
Figure 2.8 Programme vide compilé avec -fxray-instrument. 16
Figure 2.9 Accès en mémoire sur x86-64. 18
Figure 2.10 Accès en mémoire relocalisé sur x86-64. 19
Figure 2.11 Accès en mémoire relocalisé et optimisé sur x86-64. 19
Figure 2.12 Région à modifier de cinq octets composée de deux instructions. 20
Figure 4.1 Patch flow. 37
Figure 4.2 Patching of the prologue of an unoptimized procedure. 39
Figure 4.3 Pools placement example on a program compiled with -pie. . . 43
Figure 4.4 Pseudo C structure of a trampoline. 43
Figure 4.5 C structure of a trampoline descriptor. 44
Figure 4.6 Generic trampoline instructions on x86-64. 44
Figure 4.7 C structure of a probe context. 47
Figure 4.8 Generic handler for x86-64. 49
Figure 4.9 Emulation of an indirect jump. 50
Figure 4.10 Generic emulation of a relative instruction. 51
Figure 4.11 RSS usage against the number of patches. 57
Figure 4.12 Throughput sampling base scenario. 59
Figure 4.13 Throughput sampling with more threads. 59
Figure 4.14 Throughput sampling with more patches. 60
Figure 4.15 Throughput sampling with deeper stack. 60
Figure 4.16 Throughput sampling with unbounded frequency. 61
Figure 4.17 C powmod function. 62
Figure 4.18 Object dump of C powmod function. 62
Figure 4.19 Average execution time of powmod (function entry). 63
Figure 4.20 Average execution time of powmod (function entry without Uprobe and GDB). 64

Figure 4.21 Average execution time of powmod (function entry/exit). . . . 65
Figure 4.22 Average execution time of powmod (function entry/exit without Uprobe). 66
Figure 4.23 Shell script that reproduce the experiments. 68

LISTE DES SIGLES ET ABRÉVIATIONS

ABI	Application Binary Interface
CAS	Compare And Swap
CET	Control-flow Enforcement Technology
CFA	Canonical Frame Address
CFG	Control-Flow Graph
CISC	Complex Instruction Set Computer
ELF	Executable and Linkable Format
GC	Garbage Collector
JIT	Just In Time
OLX	Out of Line eXecution
PIE	Position Independent Executable
PLT	Procedure Linkage Table
RISC	Reduce Instruction Set Computer
RSS	Resident Set Size
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multi-Processor
TLS	Thread Local Storage
URCU	Userspace Read Copy Update
WXE	Write Xor Execute

CHAPITRE 1 INTRODUCTION

Les logiciels modernes demandent de plus en plus de puissance de calcul par la nature et la complexité de leurs tâches. On peut penser par exemple à un plus gros volume de traitement de données ou bien à des algorithmes plus sophistiqués. L'approche moderne est souvent de paralléliser ou de distribuer sur plusieurs nœuds les calculs d'un logiciel, ce qui augmente le nombre de communications entre les processus qui le composent. Cela a des répercussions sur la complexité du développement du logiciel, notamment lors du débogage. En effet, une des plus grandes parties de l'effort de l'ingénieur est mis sur cette phase du cycle de vie du logiciel. Toutefois, la nature de certains bogues modernes – comme une course critique – les rend parfois difficiles à résoudre avec les outils traditionnels tels qu'un débogueur ou l'enregistrement chronologique des données (*logging*). On parle dans le jargon d'un *Heisenbug* ou plus formellement d'effet de sonde. En d'autres mots, le fait même d'observer le comportement d'un logiciel – par exemple dans le but de le déboguer – affecte le logiciel et donc le résultat observé. Cela s'avère extrêmement problématique, car l'ingénieur est confronté à une situation où les outils qu'il emploie pour trouver la source du problème contribuent, au contraire, à brouiller cette source. Ce phénomène affecte aussi le profilage, analyse essentielle en production pour identifier les goulots d'étranglement ou encore des portions de logiciel qui consomment trop d'énergie. Cette analyse est d'autant plus primordiale de nos jours qu'il ne sera plus possible de pallier les mauvaises performances d'un logiciel en le migrant sur une nouvelle génération de matériel, ce qui est pratique courante dans l'industrie. De plus, l'utilisation des systèmes embarqués à grande échelle et le développement durable sont aussi des raisons pour lesquelles il est souhaitable de réduire la puissance électrique consommée. Ainsi, les outils spécialisés dans l'instrumentation logicielle conçus précisément dans le but de résoudre ce type de problématiques ont vu leur intérêt croître dans les dernières années, par opposition aux outils dits généralistes. Enfin, le laboratoire DORSAL – où cette recherche a été menée – est en relation avec plusieurs partenaires industriels qui ont des besoins similaires à ceux des systèmes embarqués. Ce contexte industriel justifie les besoins de dévelopement pour des outils de traçage de haute performance.

L'instrumentation d'un logiciel désigne un ensemble de techniques permettant l'analyse de celui-ci. Elle permet notamment le profilage du logiciel, par exemple en surveillant l'entrée de chaque fonction comme `gprof`, ou bien en vérifiant son exécution, par l'instrumentation de tous les accès en mémoire comme `ASAN`. Conceptuellement, elle peut être vue comme un sous-programme qui est greffé au programme principal et qui effectue des analyses sur ce dernier lors de son exécution, et ce de façon transparente. Ce sous-programme peut être représenté

par un ensemble de points de trace qui forment une spécification de l'instrumentation. Les points de trace sont quant à eux composés d'un emplacement (une adresse) et d'une procédure qui effectue un calcul d'analyse lorsque le programme atteint cet emplacement pendant son exécution. Ils peuvent parfois être activés ou désactivés pour modifier le comportement de l'instrumentation.

Elle peut être définie avant la compilation du programme. On la dit alors statique, puisque définie sous forme de points de trace à même le code source du programme ou émis par le compilateur. L'exécutable résultant contient ainsi les instructions nécessaires pour effectuer l'instrumentation désirée et habituellement une condition d'activation. Pour ajouter ou enlever des points de trace et donc modifier le comportement de l'instrumentation, il est nécessaire de recompiler le programme. Il est toutefois possible d'activer et désactiver des points de trace et donc de modifier le flot de contrôle de l'instrumentation, mais il est impossible de l'étendre. Sous cette forme, les surcoûts temporels associés à l'instrumentation sont minimisés. En contrepartie, les tailles en mémoire et sur disque du binaire sont augmentées. De plus, la flexibilité de modification de l'instrumentation est limitée. En effet, il est nécessaire d'avoir accès au code source du programme, à ses dépendances et de bien comprendre son moteur de production avant de pouvoir obtenir une nouvelle version du logiciel avec une nouvelle instrumentation.

L'instrumentation dite dynamique, quant à elle, permet des modifications au-delà du contrôle des points de trace. Dans la littérature, on fait souvent référence à ce type lorsqu'un binaire est modifié sur le disque pour en produire un nouveau avec une instrumentation ajoutée ou modifiée. Cette forme d'instrumentation dynamique peut être considérée comme une fonction pure prenant en entrée une spécification ainsi qu'un fichier binaire et émet en sortie un binaire transformé qui inclut les instructions correspondant à la spécification de l'instrumentation désirée. Ce type est plus flexible que le type statique présenté plus haut, et ne demande pas de recompiler le logiciel. Cependant, une analyse approfondie du programme complet – y compris ses dépendances – est nécessaire, ce qui requiert une grande quantité de mémoire et possiblement de temps.

Il existe une autre forme d'instrumentation dynamique, dite à temps réel, où la modification du binaire se fait lors de son exécution. Ainsi, l'instrumentation est construite à même l'espace d'adressage virtuel du processus qui exécute le programme. Elle peut par la suite être renversée pour retrouver la forme originale du programme. Ce type est le plus flexible de tous et minimise l'utilisation de la mémoire, mais en contrepartie, les performances sont affectées. À moins d'une mention contraire, c'est à cette forme d'instrumentation dynamique que ce mémoire fera référence. L'autre forme sera plutôt nommée instrumentation dynamique hors

ligne.

1.1 Définitions

La nomenclature en instrumentation dynamique est variable entre les auteurs et travaux. Il est néanmoins impératif d'en définir une qui est claire et regroupe tous les concepts qui seront abordés. Les définitions suivantes regroupent des concepts communs aux différents travaux en matière d'instrumentation dynamique, mais aussi de nouveaux concepts introduits par cette recherche. Elles ne viennent pas d'ouvrages, mais bien de l'auteur de ce mémoire et constituent donc le reflet de sa compréhension du sujet.

Divergence du programme Lorqu'il y a instrumentation dynamique d'un programme, il y a pertubation du flot de contrôle naturel de celui-ci. On dit que le calcul courant est divergé. La reprise du calcul courant est appelée la continuation du programme.

Sonde Les sondes (*probes*) sont des procédures de rappel déterminées par l'instrumenteur. Elles ont pour but d'effectuer une analyse du programme. Une sonde peut être pure lorsqu'elle ne fait qu'échantillonner l'état du programme. Par exemple, un instrumenteur ayant pour but d'analyser les performances d'un logiciel enregistre des points dans le temps. À l'inverse, une sonde est dite impure lorsqu'elle modifie le comportement du programme, par exemple en modifiant un registre ou en interrompant prématurément ce dernier. Dans tous les cas, une sonde peut indirectement modifier le résultat d'un programme par ses coûts en temps de calcul et en espace mémoire.

Origine de la sonde L'origine de la sonde est l'adresse virtuelle – dans l'espace d'adressage du processus instrumenté – où l'instrumentation doit avoir lieu. On dit que l'instruction se trouvant à l'origine de la sonde est instrumentée. La sonde elle-même ne se trouve pas à son origine, mais ailleurs dans l'espace d'adressage virtuel. On parle aussi de l'origine du correctif ou de l'emplacement de la sonde.

Trampoline Un trampoline est un des nœuds de transition entre l'origine de la sonde et celle-ci, dans le flot de contrôle qui appartient à ce qu'on appelle le correctif. Le trampoline permet entre autres de faire un branchement vers le gestionnaire de correctif tout en lui passant les informations nécessaires à l'appel de la sonde et le retour au tampon d'exécution relocalisée.

Gestionnaire de correctif Un gestionnaire de correctif (*handler*) s'assure de sauvegarder l'état du calcul courant avant d'exécuter la sonde. Celui-ci restaure par la suite l'état original – ou modifié par la sonde – avant d'effectuer un branchement au tampon d'exécution relocalisée.

Tampon d'exécution relocalisée Un tampon d'exécution relocalisée (Out of Line eXecution (OLX) buffer) contient les instructions originales – possiblement émulées – à l'origine de la sonde ainsi qu'une instruction de branchement vers la suite du programme.

Correctif Un correctif (*patch*) est l'ensemble des informations qui permettent de diverger le calcul courant d'un programme pour exécuter une sonde avant de retourner à celui-ci. Le correctif contient aussi l'ensemble des informations permettant d'enlever la sonde, c.-à-d. de renverser une partie de l'instrumentation. L'ensemble des correctifs forment la spécification de l'instrumentation dynamique.

Flot de contrôle du correctif Le flot de contrôle du correctif (*patch flow*) est l'ensemble des nœuds alloués – à ce correctif – qui permettent la diversion du calcul courant et sa reprise à la suite de l'exécution de la sonde. Un fil d'exécution entre dans un flot de contrôle d'un correctif au moment où la première instruction de branchement à l'origine de la sonde est exécutée et en ressort au moment où une instruction de branchement dans le tampon d'exécution relocalisée est exécutée et sort de ce dernier.

1.2 Objectifs de recherche

L'objectif principal de la recherche est de combiner les algorithmes les plus récents dans la littérature en matière d'instrumentation dynamique, afin d'obtenir le meilleur taux de couverture d'instrumentation et les mettre à la disposition d'outils sous la forme d'une bibliothèque – nommée **Libpatch** – en espace utilisateur. Pour ce faire, l'objectif principal est divisé en sous-objectifs.

1. Maximiser la couverture des sondes, c.-à-d. les emplacements où une sonde peut être insérée dans un programme.
2. Minimiser le surcoût associé aux sondes lors de leur installation/déinstallation ainsi que lors de leur exécution.
3. Minimiser le surcoût en mémoire.
4. Garantir l'exactitude et la portabilité de l'instrumentation dynamique.

5. Offrir une interface logicielle qui abstrait les détails d'implémentation et qui est portable sur plusieurs architectures matérielles.

1.3 Plan du mémoire

Ce mémoire est divisé en chapitres.

Chapitre 2

Présente l'état de l'art en matière d'instrumentation dynamique sous forme d'une revue de la littérature scientifique.

Chapitre 3

Explique en détail le domaine de la problématique qui entoure l'instrumentation dynamique.

Chapitre 4

Est le verbatim d'un article scientifique soumis à `Software: Practice and Experience` et qui présente les détails d'implémentation de la bibliothèque `libpatch` qui résout la problématique.

Chapiter 5

Discussion de l'article scientifique.

Chapitre 6

Conclusion de ce mémoire.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art en matière d'instrumentation dynamique. Les sections du chapitre sont ordonnées de telle sorte que les techniques présentées dans une section sont soit fondées sur les techniques des sections précédentes ou bien viennent les compléter. Les points forts et faibles de chaque technique sont présentés sous la forme d'une critique constructive et de proposition pour leur amélioration. L'union de ces techniques sera le sujet de l'article au Chapitre 4 tandis que l'ensemble des problèmes relevés par cette unification est le domaine de la problématique détaillée au Chapitre 3.

2.1 Points de trace par interruption logicielle

Traditionnellement, les points de trace dynamique en espace utilisateur s'implémentent à l'aide d'une instruction d'interruption logicielle. Dans ce modèle, il y a deux processus : le traceur et le tracé. Par exemple dans le cas d'un débogueur, celui-ci prend le rôle de traceur et, à l'aide l'appel système `ptrace(2)`, interrompt l'exécution de tous les fils d'exécution du tracé pour y insérer des instructions d'interruption logicielle aux emplacements désirés. Le traceur relance les fils d'exécution du tracé et écoute pour des évènements. Lorsqu'un fil d'exécution exécute une interruption logicielle, son exécution est interrompue et un évènement est généré par le noyau du système d'exploitation qui l'achemine au traceur. Ce dernier consomme l'évènement et prend une décision. Pour un débogueur, la réaction classique à une interruption logicielle est d'interrompre tous les autres fils d'exécution du traceur. Le débogueur détient alors une vue globale de l'état du tracé et détermine comment la reprise de l'exécution se fera. Cette technique est très flexible, car elle permet d'insérer des points d'instrumentation partout dans la cible à instrumenter. Elle a aussi l'avantage d'être indépendante – à quelques détails près – de l'architecture matérielle grâce à l'interface offerte par l'appel système `ptrace(2)`. Notons que certaines architectures offrent des registres pouvant servir à interrompre un programme lorsque son compteur de programme est égal à une des valeurs dans ceux-ci [1]. C'est ce qu'on appelle une interruption matérielle. On peut donc généraliser le concept par un point de trace par interruption.

2.1.1 Uprobe

`Uprobe` [2] est un exemple d'outil qui utilise des points de trace par interruption logicielle. Son traçage se fait sur des processus en espace utilisateur, mais sa logique et l'instrumentation

sont faites dans le noyau Linux. L'insertion des points de trace dans des programmes se fait à partir du pseudo-système de fichiers `debugfs` [3]. Avec ce dernier, la spécification d'instrumentalisation se fait au niveau d'un fichier binaire, ce qui représente un avantage. Il est donc capable de faire une forme d'instrumentation hors ligne sur un binaire, où les processus qui l'exécuteront obtiennent au moment de leur exécution les points de trace déjà installés. Il est également possible d'instrumenter les processus qui exécutent déjà le binaire. Cependant, il a de nombreux désavantages. En effet, l'interface utilisée – `debugfs` – entre l'utilisateur et le noyau du système d'exploitation n'a pas une ABI (Application Binary Interface) qui garantit une stabilité dans le temps. Les programmes ou scripts qui utilisent cette interface sont donc susceptibles de cesser de fonctionner dans le futur. De plus, l'interface n'est pas facile d'utilisation : par exemple, la spécification d'un point de trace demande impérativement de passer une adresse à instrumenter et non un symbole avec un décalage. Il incombe donc à l'instrumenteur de résoudre les symboles du programme. Aussi, l'instrumentation se fait dans le noyau du système d'exploitation. Pour étendre les fonctionnalités de l'instrumentation, il faut impérativement utiliser un module de noyau qui sera chargé dynamiquement dans celui-ci et qui s'enregistre auprès de `Uprobe`. Enfin, les changements de contexte entre l'espace utilisateur du processus et le noyau entraînent un surcoût de temps considérable. En résumé, `Uprobe` est un outil pratique pour faire de l'instrumentation dynamique rapidement, mais pour toute instrumentation plus complexe ou robuste à long terme, il ne s'agit pas d'une solution optimale.

2.2 Accélération des points de trace dynamiques

Les points de trace par interruption ont comme désavantage d'induire un fort surcoût durant l'exécution, en particulier en espace utilisateur à cause des changements de contexte entre le noyau du système d'exploitation et l'espace utilisateur [4]. Pour diminuer ces surcoûts, il existe de nombreuses solutions. Les deux plus prometteuses sont la réécriture du flot de contrôle du programme et l'insertion de branchements dynamiques.

2.2.1 Réécriture du flot de contrôle du programme

Une façon de faire de l'instrumentation dynamique est de procéder à l'analyse du CFG (Control-Flow Graph) du binaire et de l'augmenter. Cette technique peut s'appliquer aussi bien hors ligne qu'en temps réel. L'idée est de reconstruire le flot de contrôle naturel du programme pour déterminer le comportement du programme et ainsi faire une instrumentation dynamique adaptée.

Dyninst

L'outil `Dyninst` [5,6] implémente une telle technique. L'instrumentation peut se faire à n'importe quel moment dans ce que les auteurs appellent le *continuum* de l'exécution. En d'autres mots, `Dyninst` supporte l'instrumentation dynamique hors ligne et en temps réel. Une force de `Dyninst` réside dans sa spécification de l'instrumentation qui se définit par deux primitives : les *instpoints* et les *snippets*, soit respectivement les origines des sondes et les sondes elles-mêmes. Cependant, l'origine d'une sonde n'est pas forcément une adresse virtuelle, mais peut être un nœud dans le flot de contrôle d'une fonction. De plus, une sonde n'est pas forcément une fonction de rappel, mais peut être des instructions incorporées à même le tampon d'exécution relocalisée. Une autre de ses forces réside dans la manipulation du CFG pour raffiner la spécification. Par exemple, une spécification qui demande d'instrumenter l'entrée d'une fonction débute, en théorie, l'instrumentation sur la première instruction possible. Par conséquent, cette instrumentation peut se faire exécuter plus de fois que le nombre d'appels de la fonction, par exemple si un branchement intraprocédural est fait vers l'origine de la sonde. En modifiant le CFG, `Dyninst` s'assure que l'instrumentation est exécutée uniquement lors d'un appel interprocédural.

Construction du CFG Il y a toutefois des désavantages considérables à son utilisation. Pour reconstruire le CFG – même sans table de symboles –, `Dyninst` désassemble l'entièreté du programme pendant que celui-ci est totalement arrêté grâce à l'appel système `ptrace(2)`. Cela induit un haut pic de consommation de mémoire proportionnel à la taille du programme et une chute nette de l'utilisation des processeurs par le programme, et ce pour une durée allant jusqu'à quelques secondes. Ce pic d'utilisation de la mémoire peut être problématique sur les systèmes avec une faible capacité tels que les systèmes embarqués. De plus, l'arrêt complet du programme pendant un temps aussi considérable peut être inacceptable en production. Enfin, `Dyninst` dit détecter toute nouvelle région instrumentable, générée par exemple avec un compilateur JIT (Just In Time). Cela implique nécessairement une autre phase de désassemblage et arrêt du monde, mais aussi une forme d'écoute d'évènements – venant du système d'exploitation par l'appel `ptrace(2)`. Cette écoute n'est pas gratuite, car tout signal ou appel système est intercepté par le traceur, ce qui ralentit davantage le programme.

Modification du programme Pour modifier le programme, `Dyninst` détermine une région – des instructions ou un sous-ensemble du CFG – à relocaliser. C'est ce qu'on peut appeler l'augmentation du CFG original. Pour connecter cette extension, des branchements

sont ajoutés avec l'appel système `ptrace(2)` dans la fonction originale. Par exemple, si l'on souhaite instrumenter l'entrée d'une fonction, la première instruction de celle-ci sera remplacée par un saut vers une nouvelle fonction relocalisée et augmentée avec l'instrumentation. Les fils d'exécutions qui exécutent la fonction sont relocalisés au nœud correspondant dans la fonction relocalisée. Pour enlever l'instrumentation, les branchments sont retirés de la fonction originale puis la pile de chaque fil d'exécution est vérifiée pour déterminer s'il est possible de recycler la fonction relocalisée. Le tout se fait pendant que l'ensemble des fils d'exécution du programme sont arrêtés. Néanmoins, leur méthode de modification présente plusieurs problèmes. Premièrement, il est dit que la relocalisation des fils d'exécution se fait en comparant leur compteur de programme à celui des blocs de base du CFG. Cependant, qu'en est-il lorsqu'un fil d'exécution exécute un signal asynchrone ? Le fil d'exécution ne se trouve pas dans un des blocs de base qui s'apprêtent à être relocalisés, mais se prépare à y retourner. Deuxièmement, la relocalisation d'une région plus grande que nécessaire pour un branchement induit un surcoût supplémentaire en mémoire et en temps. Par exemple, on peut penser à la relocalisation complète d'une fonction au lieu d'uniquement les instructions à l'origine de la sonde. Enfin, l'arrêt du monde ajoute à son tour un surcoût en temps lors de l'insertion des points de trace. Finalement, l'utilisation de l'appel système `ptrace(2)` avec `PTRACE_POKETEXT` peut modifier un mot à la fois [7], ce qui rallonge le temps pendant lequel le programme est à l'arrêt.

2.2.2 Branchements dynamiques

La modification du CFG d'un programme est puissante, mais elle impose un surcoût considérable en mémoire et temps. De plus, son analyse et sa modification sont faites par un processus externe. Il est cependant possible de rester totalement dans l'espace d'adressage virtuel du processus, de ne pas construire de CFG et de ne pas utiliser d'interruption logicielle. Pour ce faire, il faut insérer une instruction de branchement au lieu d'une instruction d'interruption pour diverger le calcul courant vers un trampoline. Le trampoline contient généralement une séquence d'instructions qui permet l'exécution de la sonde avant d'exécuter les instructions originales à l'emplacement de la sonde, et retourner à la continuation du programme. C'est d'ailleurs une optimisation faite par l'outil `Kprobe` dans l'espace noyau. La technique est similaire à la réécriture du CFG du programme, mais est moins invasive et demande moins de ressources.

Cependant, elle amène d'autres problèmes majeurs. Premièrement, un branchement relatif impose qu'un trampoline soit proche de l'origine de la sonde à cause de sa limite en termes de distance. Par exemple sur `x86-64`, il existe plusieurs types de branchement relatif. Si on

considère uniquement les branchements relatifs inconditionnels, il en existe deux. Soit un à 2 octets et un à 5 octets qui ont respectivement un décalage encodé sur 8 bits et 32 bits. Dans le premier cas, un trampoline doit se trouver dans un voisinage de -128 *octets* à 127 *octets* autour de l'emplacement de la sonde. Dans le second cas, le trampoline doit se trouver dans un voisinage de -2 *Gio* à 2 *Gio* $- 1$ *octets* autour de l'emplacement de la sonde. Cela peut devenir problématique pour des programmes très larges, car une grande partie du voisinage atteignable par un branchement est possiblement déjà alloué par le programme.

Deuxièmement, sur les architectures de type CISC (Complex Instruction Set Computer) comme `x86-64`, les instructions ont des tailles variables. C'est un problème si l'on veut, par exemple, utiliser un branchement de 5 octets sur cette architecture à l'emplacement de la sonde, où il faut que l'instruction qui s'y trouve – dite primaire – soit d'au moins 5 octets. Sinon, il faut utiliser les instructions suivantes – dites secondaires – pour former un bloc d'au moins 5 octets. Cela introduit aussi de nouveaux problèmes quant à la cohérence des accès en mémoire. Notons que ce problème n'est pas présent sur les architectures RISC (Reduced Instruction Set Computer) qui ont des instructions de taille fixe.

Troisièmement, il peut y avoir une incohérence des accès en mémoire si la région à modifier est exécutée par d'autres fils d'exécution [8–10]. Cela est hautement dépendant de l'architecture et même de la microarchitecture du matériel. Il est donc impératif de suivre les protocoles de modification croisée des fabricants.

Quatrièmement, certaines instructions sur plusieurs architectures peuvent être relatives au compteur de programme. Cela est hautement problématique, car il est nécessaire d'exécuter les instructions remplacées par le branchement ailleurs dans le programme. Il faut donc être en mesure de pouvoir relocaliser les instructions dans un tampon d'exécution relocalisée, quitte à les émuler dans celui-ci. L'exemple le plus trivial qui s'applique sur toutes les architectures matérielles est celui d'un branchement relatif qui, par définition, utilise et modifie implicitement le compteur de programme. Il faut donc recalculer son décalage à une nouvelle position. Un autre exemple, cette fois sur `x86-64`, est présenté sur la Figure 2.1 où le programme effectue un accès en mémoire à une variable globale. L'assembleur généré par le compilateur (sans optimisation) est visibile sur la Figure 2.2. On constate que cet accès en mémoire utilise le registre `rip` qui est le compteur de programme. Ce type d'accès en mémoire est en effet souvent utilisé par les compilateurs pour les variables globales dans le but de faciliter la relocalisation de l'exécutable par le chargeur-éditeur de liens.

FIGURE 2.1 Accès à une variable globale en C.

```c
int global;
int main(void) { return global; }
```

FIGURE 2.2 Accès à une variable globale en x86-64.

```
        .text
        .globl  global
        .bss
        .align 4
        .type   global, @object
        .size   global, 4
global:
        .zero   4
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    global(%rip), %eax
        popq    %rbp
        ret
```

Cinquièmement, certains systèmes ou architectures ont des règles dans leur ABI qui comportent des impacts majeurs sur l'instrumentation dynamique. Par exemple sur `x86-64`, l'ABI autorise les fonctions à utiliser les 128 octets au-dessus de la pile d'exécution sans modifier le pointeur de la pile, c.-à-d. sans faire d'allocation [11]. C'est ce qu'on appelle la zone rouge (*red zone*), une optimisation souvent utilisée par les compilateurs. Les bibliothèques telles que *libc* et le noyau du système d'exploitation s'assurent de respecter ces conventions. Un non-respect résulterait en un comportement indéfini.

Sixièmement, certains systèmes ou architectures matérielles ont des mesures de sécurité pour prévenir la modification d'instructions ou bien pour détecter un flot de contrôle non prévu par le compilateur. Par exemple, les pages d'un processus peuvent être exclusivement en mode d'accès pour écriture ou pour l'exécution. On appelle ce modèle de sécurité WXE (Write Xor Execute). Il n'est donc pas possible de modifier des pages exécutables. Il faut d'abord changer le mode de protection à l'aide de l'appel système `mprotect(2)` avec `PROT_WRITE`, modifier les pages et refaire l'appel système avec `PROT_EXEC` [12]. Il est cependant possible qu'entre les deux appels système, un fil d'exécution tente d'exécuter la région à modifier, ce qui générera une faute de segmentation. Un autre exemple de mesure de sécurité est celui d'Intel CET (Control-flow Enforcement Technology) [13, 14] où les branchements indirects

doivent obligatoirement atteindre une instruction `endbr64` avant toute autre exécution, sans quoi le processeur génère une faute. Cette technologie utilise aussi une pile dite fantôme (*shadow stack*) qui assure que le flot d'appels d'un fil d'exécution ne soit pas modifié autre que par les instructions `call` et `ret`. Si, par exemple, la pile est modifiée par l'instrumentation dynamique, ce n'est pas le cas de la pile fantôme et il y aura une incohérence entre les deux qui sera détectée au prochain `ret`.

Kprobe

L'outil `Kprobe` [15–17] permet de faire de l'instrumentation dynamique dans le noyau du système d'exploitation. Il n'est pas destiné à faire de l'instrumentation dans un processus utilisateur. Cependant, il est intéressant d'observer ses techniques d'instrumentation dynamique. D'abord, l'outil, par défaut, utilisera des interruptions logicielles pour tous ses points de trace. Celles-ci imposent un moindre surcoût en temps qu'en espace utilisateur, puisque les changements de contexte sont moins coûteux. Ensuite, après avoir inséré une interruption logicielle, une optimisation est effectuée – uniquement sur les architectures `i386` et `x86-64` – si le contexte de l'exécution le permet. Par exemple, si l'instruction où se trouve l'origine de la sonde fait au moins cinq octets, alors l'interruption logicielle est remplacée par un branchement de cette taille. L'instruction originale à l'origine de la sonde est relocalisée dans un tampon d'exécution relocalisée. Si l'instruction dépend du compteur de programme, son décalage est recalculé. `Kprobe` est aussi capable d'optimisation si l'instruction à l'origine de la sonde fait moins de cinq octets. Dans ce cas, les instructions suivantes – dites secondaires – doivent former une région d'au moins cinq octets avec l'instruction primaire. Elles doivent toutes faire partie de la même fonction et seront aussi relocalisées avec l'instruction primaire. On dit que celles-ci forment une région à optimiser. Néanmoins, si la fonction contient au moins une instruction qui fait un branchement dans la région à optimiser, si la fonction peut causer une exception, ou si la fonction contient un branchement indirect, ou encore si une des instructions de la région à optimiser ne peut être relocalisée, alors une approche conservatrice est adoptée et l'optimisation n'est pas appliquée.

Uftrace

`Uftrace` [18, 19] est un traceur en espace utilisateur inspiré du traceur noyau `Ftrace` [20, 21]. Son but est de tracer les entrées et sorties des fonctions instrumentées et il existe plusieurs façons d'y parvenir. Il faut comprendre avant tout que `Uftrace` lance le programme à instrumenter et précharge ses bibliothèques dans celui-ci à l'aide de `LD_PRELOAD`. Il s'agit donc d'une instrumentation pseudo-dynamique, car le programme ne s'exécute pas pendant l'ins-

tallation des points de trace. La Figure 2.3 montre un code en C et la Figure 2.4 le code assembleur résultant, ce sera le cas d'étude pour comprendre comment `Uftrace` réalise son instrumentation.

FIGURE 2.3 Programme vide en C.

```c
int main(void)
{
        return 0;
}
```

FIGURE 2.4 Programme vide en assembleur.

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        xor     %eax, %eax
        popq    %rbp
        ret
```

mcount La première façon est d'utiliser l'option `-pg`, lors de la compilation du programme, qui permet d'installer un appel à la fonction `mcount` après chaque prologue d'une fonction [22], comme montré à la Figure 2.5. Il est aussi possible d'utiliser l'option `-mfentry` pour que l'appel soit avant le prologue. Par défaut, cet appel est fait à l'implémentation dans la bibliothèque standard C. Cependant, `Uftrace` usurpe le symbole grâce à son préchargement pour y installer son implémentation. Cette dernière effectue l'instrumentation et change la valeur de retour sur la pile pour qu'une seconde fonction soit appelée au retour. Ainsi est obtenue la différence entre l'entrée et la sortie d'une fonction. Le problème avec l'option `-pg` est que toutes les fonctions – à l'exception de celles marquées avec l'attribut `no_instrument_function` [23] – dans le programme sont instrumentées. Cela peut grandement ralentir le programme. Il est donc possible d'utiliser l'option `-mnop-mcount`. Dans ce cas, on utilise plutôt une instruction NOP de cinq octets qui, dynamiquement, est remplacée par un appel vers `mcount`. Cette technique est montrée à la Figure 2.6.

FIGURE 2.5 Programme vide compilé avec -pg.

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         call    mcount
5         xor     %eax, %eax
6         popq    %rbp
7         ret
```

FIGURE 2.6 Programme vide compilé avec -pg -mnop-mcount.

```
1 main:
2         pushq   %rbp
3         movq    %rsp, %rbp
4         ;; nopl (%rax, %rax)
5         .byte   0x0f, 0x1f, 0x44, 0x00, 0x00
6         xor     %eax, %eax
7         popq    %rbp
8         ret
```

cyg_profile De façon similaire à `mcount`, il est possible d'utiliser l'option `-finstrument-functions` pour insérer une paire d'appels à `__cyg_profile_func_enter` et `__cyg_profile_func_exit` pour toutes les fonctions, y compris celles qui ont été optimisées `inline` [22]. Ces fonctions d'instrumentation acceptent des arguments, d'où la plus grande intrusion à la Figure 2.7. De façon analogue, `Uftrace` installe sa propre version des deux fonctions lors du chargement du programme.

FIGURE 2.7 Programme vide compilé avec -finstrument-functions.

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        pushq   %rbx
        subq    $8, %rsp
        movq    8(%rbp), %rax
        movq    %rax, %rsi
        movl    $main, %edi
        call    __cyg_profile_func_enter
        xor     %ebx, %ebx
        movq    8(%rbp), %rax
        movq    %rax, %rsi
        movl    $main, %edi
        call    __cyg_profile_func_exit
        movl    %ebx, %eax
        movq    -8(%rbp), %rbx
        leave
        ret
```

XRay Il est possible d'insérer des instructions NOP avant le prologue et après l'épilogue des fonctions dans un programme à l'aide de l'option `-fxray-instrument` [24] (clang uniquement), comme indiqué dans la Figure 2.8. Cette technologie appelée XRay [25] permet de remplacer dynamiquement ces instructions par des appels ou branchements pour faire de l'instrumentation. `Uftrace` va typiquement installer des appels vers ses procédures internes lors du chargement du programme. Le grand avantage est que l'instrumentation est très sélective. Cependant, la taille du binaire est plus grande.

FIGURE 2.8 Programme vide compilé avec -fxray-instrument.

```
main:
        nopw    512(%rax,%rax)
        pushq   %rbp
        movq    %rsp, %rbp
        xor     %eax, %eax
        popq    %rbp
        retq
        nopw    %cs:512(%rax,%rax)
```

PLT Pour instrumenter les fonctions publiques d'une bibliothèque utilisée par un programme, `Uftrace` modifie la PLT (Procedure Linkage Table) – une section d'un fichier ELF (Executable and Linkable Format) – des modules à instrumenter pour appeler une procédure d'instrumentation avant d'appeler la fonction originale. Cela fonctionne de façon similaire à un pré-chargement de fonctions avec `LD_PRELOAD`, sauf que la PLT est locale à chaque module et que la modification de la PLT peut se faire de façon dynamique [26].

Branchement dynamique Dans le cas où il n'est pas possible d'utiliser les autres techniques, `Uftrace` se résout à faire de l'instrumentation dynamique en remplaçant les instructions au début de la fonction par un branchement vers une fonction interne. Cette technique est grandement limitée dans son implémentation, car il n'y a pas de relocalisations des instructions dépendantes du compteur de programme. Il y a aussi une approche conservatrice comme `Kprobe` où tout branchement dans la zone à modifier ou branchement indirect dans la fonction annule l'instrumentation. Enfin, tous les points de trace dynamique partagent un trampoline, ce qui implique une recherche dans une table de hachage pour retrouver les instructions originales à exécuter.

Dyntrace

L'outil `Dyntrace` [27] implémente une alternative pour l'instrumentation dynamique en offrant quelques pistes pour résoudre les principaux inconvénients. Cependant, plusieurs détails font que cet outil n'effectue pas une instrumentation dynamique rapide ni correcte, c.-à-d. que l'application de l'instrumentation dynamique sur le programme résulte en un comportement indéfini de ce dernier. Cette section détaille donc l'implémentation de `Dyntrace` et discute de ses qualités et lacunes.

Premièrement, le branchement se fait à un trampoline alloué avec l'appel système `mmap(2)` et l'option `MAP_FIXED`. Le trampoline contient un second branchement – absolu cette fois – vers la sonde. Cependant il faudrait que cette allocation soit faite avec `MAP_FIXED_NOREPLACE` pour éviter d'utiliser des pages qui sont déjà allouées par le processus [28, 29]. De plus, cette méthode est très lente, car elle peut demander plusieurs appels système avant de trouver une solution valide. En effet, prenons en exemple le simple cas d'une instruction de 5 octets qui est remplacée par un branchement de la même taille sur `x86-64`. Il y a alors 2^{32} possibilités, soit 2^{20} pages en supposant que chaque page fait 2^{12} octets. Il faudra donc faire dans le pire des cas un peu plus d'un million d'appels système pour trouver une solution. Il serait plus performant de faire un seul gros bloc d'allocation une fois et un allocateur mémoire qui utilise ce bloc. Cela présente trois avantages : le nombre limité d'appels système, la réservation de

l'espace mémoire virtuel et la faible fragmentation de la mémoire. Notons que même si un gros bloc en mémoire est alloué avec l'appel système `mmap(2)`, celui-ci n'utilise aucune mémoire physique, sauf les structures nécessaires à l'interne dans le noyau du système d'exploitation. En effet, la plupart des systèmes d'exploitation tels que Linux font de l'allocation paresseuse, c.-à-d. que la mémoire physique associée à une page est allouée uniquement après la première écriture dans celle-ci.

Deuxièmement, les instructions relatives au compteur de programme sont émulées dans un tampon d'exécution relocalisée. La Figure 2.9 montre une instruction qui dépend du compteur de programme (`rip`) et la Figure 2.10 montre cette instruction émulée une fois relocalisée. On constate que le registre utilisé comme base pour l'instruction `mov` est maintenant `rbx`. Ce dernier est sauvegardé sur la pile avant le début de la séquence d'émulation et restauré à la fin. La valeur `ORIGINAL-RIP` qui est mise à l'intérieur de `rbx` correspond au compteur de programme où se trouve l'instruction originale. D'autres techniques existent pour d'autres cas, par exemple ceux des branchements relatifs ou bien encore des branchements conditionnels. Avantageusement, certaines de ces techniques permettent une relocalisation absolue, ce qui n'est pas le cas par exemple de `Kprobe` qui recalcule uniquement le déplacement (*displacement*) des instructions. En effet, une relocalisation relative restreint l'endroit où doit se trouver le tampon d'exécution relocalisée dans l'espace d'adressage virtuel du programme. Cependant, certaines techniques ne respectent pas l'ABI en touchant la zone rouge de la fonction. De plus, il y aurait avantage à faire une analyse des registres de la fonction où se trouve l'origine de la sonde pour optimiser les instructions émulées dans le tampon d'exécution relocalisée. Cela demande l'aide d'un désassembleur, mais aussi une connaissance de l'ABI du système. Par exemple, la Figure 2.11 montre qu'il est possible d'optimiser la relocalisation de l'instruction précédente, à condition de prouver que le registre `rbx` n'a pas besoin d'être préservé à l'emplacement de la sonde.

FIGURE 2.9 Accès en mémoire sur x86-64.

```
1        mov DISP(%rip), %rax
```

FIGURE 2.10 Accès en mémoire relocalisé sur x86-64.

```
1        push %rbx
2        movabs ORIGINAL-RIP, %rbx
3        mov DISP(%rbx), %rax
4        pop %rbx
```

FIGURE 2.11 Accès en mémoire relocalisé et optimisé sur x86-64.

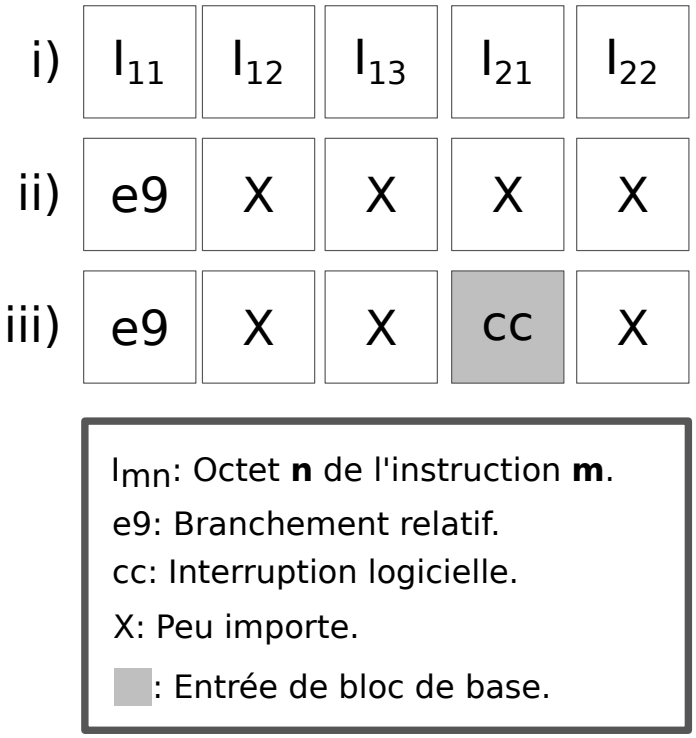
```
1        movabs ORIGINAL-RIP, %rbx
2        mov DISP(%rbx), %rax
```

Troisièmement, la modification de la région en mémoire se fait de façon atomique à l'aide d'une instruction CAS (Compare And Swap), plus précisément l'instruction `cmpxchg8b` sur `x86-64` qui modifie de façon atomique jusqu'à 8 octets. Toutefois, cela n'est pas suffisant, car il est nécessaire d'effectuer une instruction de sérialisation – telle que `cpuid` – des processeurs, avant d'exécuter des instructions qui ont été modifiées sur l'architecture `x86-64` [8,9]. Or, cette instruction n'est pas forcément présente avant la région à modifier. Il faudrait donc remplacer l'instruction précédant la région à modifier par une instruction de sérialisation, soit une nouvelle instruction. Le problème se chaîne ainsi et on se retrouve face à un paradoxe de l'oeuf ou la poule où l'on demande une sérialisation avant une modification, mais une sérialisation demande une modification. De plus, d'autres architectures telles que ARM sont plus exigeantes et demandent d'interrompre totalement l'exécution de tous les fils d'exécution du programme avant la modification, puis une purge de la cache des instructions et une sérialisation des processeurs [10], sans quoi le résultat est indéfini. Ces détails microarchitecturaux font en sorte qu'il est possible que le remplacement atomique proposé par `Dyntrace` fonctionne sur un modèle de microarchitecture, mais pas nécessairement sur un autre.

Quatrièmement, des instructions d'interruption logicielle sont utilisées pour la tête (premier octet) de chaque instruction secondaire pour éviter l'exécution d'instructions invalides. Cela est nécessaire uniquement pour les instructions qui sont un point d'entrée d'un bloc de base du flot de contrôle du programme, mais cette stratégie est appliquée partout de façon conservatrice sur l'ensemble des instructions. Cela a comme conséquence de réduire grandement l'espace de recherche d'un trampoline. L'exemple à la Figure 2.12 montre un tel scénario

avec un branchement de 5 octets sur une région composée d'une instruction primaire de deux octets suivie d'une instruction secondaire de trois octets (point i). En théorie, ce branchement a 256^4 possibilités de décalage qui peuvent être encodées (point ii). Cependant, puisque l'instruction secondaire est l'entrée d'un bloc de base, il est possible pour le programme de l'exécuter sans passer par l'instruction primaire. La tête de l'instruction secondaire est donc forcée à une valeur qui correspond à une interruption logicielle (point iii). Ainsi, si un fil d'exécution tente d'exécuter l'instruction secondaire sans passer par l'instruction primaire, celui-ci va générer une interruption logicielle. En enregistrant un gestionnaire de signal à l'aide de l'appel système `sigaction(2)` pour le signal `SIGINT` [30], il est possible de modifier le compteur de programme du fil d'exécution à l'emplacement correspondant dans le tampon d'exécution relocalisée. Or, cette technique réduit à 256^3 le nombre de décalages possibles à encoder dans l'instruction de branchement, soit une diminution d'environ 99.61%. Pour remédier à cela, il serait possible d'étendre l'espace de recherche en reconstruisant le flot de contrôle de la fonction où se trouve l'origine de la sonde pour déterminer quelles instructions sont des points d'entrée des blocs de base. On peut donc éviter l'approche conservatrice de considérer toutes les instructions comme étant une entrée d'un bloc de base. Cette technique ne s'applique cependant pas dans le cas où la fonction a un branchement indirect qui peut uniquement être résolu au moment du branchement.

FIGURE 2.12 Région à modifier de cinq octets composée de deux instructions.



Cinquièmement, la réclamation de la mémoire d'un point d'instrumentation est contrôlée par un compteur de référence atomique, ce qui induit un surcoût notable sur les processeurs SMP (Symmetric Multi-Processor). Il y a tout de même une mention intéressante de l'utilisation de URCU [31–33] (Userspace Read Copy Update) pour diminuer ce surcoût. Cela s'explique par la bien plus grande fréquence en lecture d'un point de trace de par son exécution, par rapport à la fréquence en écriture lors de son installation et de son retrait. Néanmoins, puisque la ressource à réclamer est une région exécutable, et que c'est cette même ressource qui se protège elle-même, il existe forcément des sous-régions – avant et après l'incrémentation et la décrémentation, respectivement, du compteur de référence – de celle-ci qui sont non protégées. Pour ce faire, `Dyntrace` interrompt les fils d'exécution du programme et vérifie leur compteur de programme. Cependant, cela n'est pas performant et même incorrect. En effet, il serait préférable de vérifier un seul fil d'exécution à la fois pour réduire l'impact sur l'exécution du programme. De plus, si on prend le cas où un fil d'exécution est en train d'exécuter un signal asynchrone, et que celui-ci s'apprête à retourner dans une des sous-régions non protégées, la vérification du compteur donnera lieu à un faux négatif. Il est donc obligatoire de dérouler la pile du fil d'exécution pour vérifier tous les compteurs de programmes enregistrés sur celle-ci.

Sixièmement, les trampolines respectent l'ABI de `x86-64` en préservant la zone rouge à l'aide de l'instruction `lea -128(%rsp), %rsp` qui soustrait 128 au pointeur de la pile. Il n'est en effet pas possible de faire uniquement une soustraction avec l'instruction `sub` sur le pointeur de la pile, car cela modifierait le registre à drapeaux `rflags`. De plus, ce dernier ne peut être simplement sauvegardé sur la pile avec `pushf`, puisque cela écraserait ce qui se trouve au début de la zone rouge. Il faut donc impérativement préserver la zone rouge sans affecter le registre à drapeaux. Il existe aussi une autre façon de préserver la zone rouge, avec 16 instructions `push -8(%rsp)` consécutives, mais la première solution a l'avantage d'être courte et rapide.

Finalement, il n'y a aucun support pour les systèmes ayant une protection WXE ou Intel CET, rendant l'instrumentation dynamique sur ceux-ci simplement impossible.

Instruction calembourée

L'une des critiques de `Dyntrace` est la réduction de l'espace de recherche pour un trampoline, due à l'utilisation d'instructions d'interruption logicielle pour les têtes des instructions secondaires. C'est un concept fondamental et il est impératif de le décrire formellement pour bien le comprendre.

On représente le degré de liberté dl de l'origine d'une sonde par le logarithme en base 2 du nombre de branchements possibles à partir de cet emplacement. Autrement dit, 2^{dl} donne le nombre de branchements possibles. Dans le cas d'une instruction faisant au moins cinq

octets, et réservant quatre octets pour le déplacement associé au branchement, le degré de liberté est égal à $log_2(256^4) = 32$ qui est le degré de liberté maximal. En contrepartie, si une région est composée d'une séquence de cinq instructions d'un octet chacune et qu'on considère chaque instruction potentiellement comme une entrée d'un bloc de base (approche conservatrice), le degré de liberté est de $log_2(1^4) = 0$ qui est le degré de liberté minimal. Le degré de liberté est simplement un indicateur sur les chances qu'a une adresse en mémoire pour être l'origine d'une sonde. Plus il est haut, plus il y a de branchement possibles et plus hautes sont les chances de pouvoir atteindre un trampoline.

Liteinst Pour augmenter le degré de liberté d'une région à instrumenter, il faut augmenter le nombre de branchements possibles. Pour ce faire, il existe des instructions – sur `x86-64` – d'un octet qui peuvent faire office de remplacement d'une interruption logicielle, car elles sont illégales et génèrent une interruption similaire [34, 35]. C'est ce que fait l'outil `LiteInst` [36] par une technique appelée le calembourage d'instructions (*instruction punning*). Cette technique utilise 14 codes d'instructions d'un octet qui sont illégales sur `x86-64`. En réalité il en existe 22, soit 8 oubliés par ses auteurs. L'utilisation de cette technique de calembourage permet d'augmenter le degré de liberté minimal à $log_2(23^4) \approx 18.09$. Une amélioration considérable, mais insuffisante pour pouvoir instrumenter à grande échelle un logiciel. Par exemple, si l'on applique cette technique à la région présentée à la Figure 2.12, on obtient un degré de liberté de $log_2(256 \times 256 \times 23 \times 256) \approx 28.52$ comparativement à $log_2(256^3) = 24$ avec uniquement une interruption logicielle. Néanmoins, comme pour `Dyntrace`, il y a des lacunes dans l'implémentation de `Liteinst`. Cette section vise à mettre ces lacunes en évidence.

Premièrement, exactement de la même façon que `Dyntrace`, l'allocation des trampolines ne se fait pas avec `MAP_FIXED_NOREPLACE`. L'application de l'instrumentation dynamique sur le programme a donc un effet indéfini sur ce dernier [29]. Cependant, à l'instar de `Dyntrace`, l'appel système est minimisé en allouant des gros blocs de mémoire qui sont par la suite utilisés par un allocateur de mémoire.

Deuxièmement, il n'y a aucune mention du respect des règles de l'ABI du système – telles que la zone rouge – ou des mesures de sécurité telles que WXE. Leur implémentation [37] ne semble pas non plus inclure de détails par rapport à ces sujets.

Troisièmement, la modification en temps réel du binaire se fait selon un protocole [38] qui est fortement couplé à la microarchitecture matérielle et qui utilise des métriques de temps empiriques. Ce protocole est détaillé davantage plus loin.

E9Patch La technique de calembourage de `Liteinst` est étendue par l'outil `E9Patch` [39] de trois façons. Bien que ces techniques soient appliquées pour une instrumentation dynamique hors ligne dans le cas de `E9patch`, celles-ci s'avèrent applicables pour une instrumentation dynamique en temps réel.

La première tactique – appellée branchement rembourré (*pad jump*) – est d'utiliser un préfixe redondant pour déplacer le début du branchement qui remplace l'instruction primaire. Cela a pour effet de réduire la taille de l'instruction primaire et de potentiellement étendre le nombre d'instructions secondaires ou d'aller chercher des octets libres supplémentaires, et ainsi d'augmenter le degré de liberté. Par exemple, si l'on considère que l'instruction primaire fait 2 octets et que les deux instructions secondaires suivantes font 2 octets, le degré de liberté de départ est de $log_2(256 \times 23 \times 256 \times 23) \approx 25.05$. En utilisant cette tactique on peut utiliser un préfixe redondant de 1 octet sur l'instruction primaire pour aller chercher un degré de liberté de $log_2(23 \times 256 \times 23 \times 256) \approx 25.05$. Les degrés de liberté ici sont identiques, car en utilisant un préfixe redondant, on perd un octet de degré de liberté de huit sur l'instruction primaire, mais on introduit un nouvel octet avec le même degré de liberté dans la seconde instruction secondaire. Cependant, les emplacements en mémoire qui peuvent être atteints peuvent différer. À degrés de libertés égaux, les solutions ne sont pas forcément les mêmes et c'est donc leur union qui est importante.

La seconde tactique – appelée éviction du successeur (*successor eviction*) – effectue des branchements vers des trampolines pour l'instruction primaire et la première instruction secondaire (le successeur), le but étant toujours d'obtenir un plus grand degré de liberté. On dit que l'instruction qui succède l'instruction primaire est évincée. La recherche du trampoline de l'évincée se fait à l'aide de la première tactique et le résultat est appliqué pour l'instruction primaire. Le plus gros problème de cette tactique est la dépendance entre les trampolines des deux instructions. La complexité de la recherche d'une solution est donc mise au carré, car toute solution trouvée pour le premier trampoline doit être vérifiée par la recherche d'un second trampoline. Cela n'est pas un problème pour l'outil `E9Patch` qui fait de l'instrumentation dynamique hors ligne, mais c'est un problème pour de l'instrumentation dynamique en temps réel. Le second problème est que l'augmentation du degré de liberté n'est pas significative. Prenons par exemple le cas d'une instruction primaire de 2 octets, suivie d'une instruction secondaire de 5 octets. Le degré de liberté initial est de $log_2(256 \times 23 \times 256^2) \approx 28.52$. En utilisant la tactique, la tête de l'instruction secondaire peut prendre une valeur supplémentaire (celle du code d'opération d'un branchement), ce qui donne un degré de liberté final de $log_2(256 \times 24 \times 256^2) \approx 28.58$ soit une augmentation d'environ 0.215%.

La troisième tactique – appellée l'éviction du voisin (*neighbour eviction*) – reprend le *modus*

operandi de la seconde tactique, mais l'applique à une région différente de l'origine de la sonde. L'objectif est de faire un petit branchement de deux octets vers cette région à partir de l'origine de la sonde. L'idée est bonne mais elle a comme inconvénient majeur que la sonde peut s'exécuter même si le programme ne passe pas par son origine, ce qui n'est pas forcément désirable pour un instrumenteur. De plus, il n'est pas possible de déterminer quel chemin a été emprunté et donc si des instructions ont besoins d'être émulées dans un tampon d'exécution relocalisée. Il est cependant possible de corriger le tir en utilisant une région dans le programme qui n'est jamais exécutée. La difficulté est donc de trouver une telle région qui est proche de l'origine de la sonde. Il est possible de trouver de telles régions en profitant du remplissage de fonction, une optimisation que font les compilateurs – tel que `GCC` (`-falign-functions`) [22] – en alignant les fonctions pour optimiser leurs accès en mémoire. Ces alignements se font à l'aide d'instructions NOP qui ne sont jamais exécutées par le programme et qui sont donc d'excellents candidats pour cette tactique [40].

NOProbe Une recherche précédente [40] – appelée ici `NOProbe` – montre l'utilisation des remplissages entre les fonctions pour faire un saut supplémentaire entre l'origine de la sonde et le trampoline. La technique fonctionne bien et étonnamment encore mieux avec les binaires optimisés. Cependant, l'implémentation de `NOProbe` a des lacunes qui font qu'elle ne fait pas une bonne instrumentation dynamique ou que celle-ci n'est pas optimale.

Premièrement, toutes les sondes partagent un trampoline commun. Cela a l'avantage de réduire l'utilisation de la mémoire. En contrepartie, il est nécessaire de faire une recherche dans une structure de données – un arbre bicolore dans ce cas-ci – pour trouver le tampon d'exécution relocalisée. Cela affecte grandement les performances de l'instrumentation dynamique.

Deuxièmement, `NOProbe` instrumente la sortie d'une fonction de deux façons. La première façon est de modifier – de l'entrée de la fonction – l'adresse de retour sur la pile par l'adresse d'un gestionnaire. La deuxième façon est de placer une sonde à la dernière instruction de la fonction qui est habituellement une instruction de retour `ret`. Cela assume que les fonctions terminent par une telle instruction et que celles-ci n'ont pas plusieurs points de sortie. Or, les compilateurs font souvent des optimisations pour les appels de récursion terminale pour réutiliser le bloc d'activation courant. Ainsi, certaines fonctions peuvent ne jamais retourner. Les compilateurs sont aussi libres de générer plusieurs `ret` dans une même fonction. Cette deuxième façon est donc à prohiber.

Troisièmement, lors de la modification du binaire, une synchronisation est faite avec l'ensemble des fils d'exécutions par un signal système asynchrone, pour déplacer ceux qui sont

dans des régions critiques en train d'être modifiés vers les instructions correspondantes dans un tampon d'exécution relocalisée. Or, il est possible que des fils d'exécutions – par exemple d'une bibliothèque tierce – ne se synchronisent jamais, car ils ont bloqué le signal. Il est donc nécessaire d'attendre pour un temps fini. La détermination de ce temps fini est de nature empirique et n'est pas robuste. `NOProbe` intercepte donc les appels à l'appel système `sigprocmask(2)` [41] pour empêcher le blocage de son signal de synchronisation.

Modification concurrente

Une des critiques précédentes est la modification des instructions à l'origine d'une sonde durant l'exécution du programme. Une telle modification est très spécifique à l'architecture matérielle où s'exécute le programme. Cependant une technique répandue est d'interrompre l'exécution de tous les fils d'exécution du programme et d'effectuer les modifications à l'aide de l'appel système `ptrace(2)`. Il est évident que cette technique dite de l'arrêt du monde – quoique portable – impose un arrêt complet de tous les calculs dans le programme pour une période assez grande, ce qui est inacceptable dans plusieurs cas. Une technique proposée [38] est d'effectuer la modification en plusieurs étapes séparées par des périodes de grâce de temps. Cela permet de garder une cohérence entre les accès en mémoire des instructions et leur exécution en laissant le temps aux modifications de se propager partout dans le système matériel. Cette période de grâce est déterminée de façon empirique pour différents modèles de processeurs, ce qui n'est pas très robuste. Il est cependant possible sur Linux de remplacer ces périodes de grâce par une synchronisation de la mémoire et des processeurs par un appel système `membarrier(2)`. Une seconde technique proposée [40] est d'effectuer une synchronisation sur chaque fil d'exécution en forçant l'exécution d'un gestionnaire de signal qui fait la synchronisation puis d'avertir l'instrumenteur. Or, comme dans le variant signal de URCU [31], il est nécessaire que cela soit coopératif, sans quoi il est possible qu'un fil d'exécution ne réponde jamais. C'est le cas, par exemple, si le signal utilisé est masqué par un fil d'exécution pour une durée indéterminée.

Branchement indirect

Les techniques de calembourage des instructions sont pratiques pour augmenter le degré de liberté. Cependant, les techniques proposées [27, 36] ont toutes des approches conservatrices, supposant que toutes les instructions sont des entrées de bloc de base ou qu'un fil d'exécution est possiblement sur le point d'exécuter une des instructions secondaires. Ainsi, toutes les têtes des instructions secondaires sont remplacées par des instructions d'un octet qui génèrent une interruption synchrone, ce qui peut induire un surcoût important en temps. Or, il est

possible de déterminer une partie des blocs de base en désassemblant la fonction où se trouve l'origine de la sonde, c.-à-d. de déterminer les cibles des branchements directs. Il est aussi possible de déplacer les fils d'exécution sur le point d'exécuter une instruction secondaire vers l'emplacement correspondant dans le tampon d'exécution relocalisée [5]. Néanmoins, il est nécessaire d'avoir une approche conservatrice lorsque la fonction à instrumenter contient un branchement indirect. En effet, dans ce cas la cible du branchement est déterminée au moment de l'exécution de l'instruction. Faute d'information sur le CFG du programme – par exemple le registre `basic_block` du programme dans la section `.debug_line` [42] – il faut considérer toutes les instructions de la fonction comme étant une potentielle entrée d'un bloc de base.

Filtrage par motif Il est possible de réduire le nombre d'instructions d'une fonction qui sont possiblement la cible d'un saut indirect dans celle-ci. Pour ce faire, [43] propose d'utiliser un filtrage par motif sur les branchements indirects. En désassemblant les instructions dans le sens inverse à partir d'un branchement indirect, il est possible de construire un motif. Si celui-ci correspond à un motif connu, il est possible de déterminer les possibles instructions cibles du branchement indirect. Cette technique a l'avantage d'augmenter le degré de liberté, mais est en contrepartie très dépendante de la chaîne de compilation utilisée pour compiler le programme. En effet, les motifs d'un compilateur ne seront pas les mêmes d'un autre compilateur et même, de version en version, les motifs d'un même compilateur peuvent changer. Il faut donc construire une base de données de motifs et l'entretenir. Une alternative qui est proposée est d'utiliser la mesure de sécurité d'Intel CET comme avantage. En effet, lorsque la mesure est activée et qu'une instruction de branchement indirect n'a pas le préfixe `notrack`, alors la cible du branchement doit être une instruction `endbr64`, les autres instructions ne sont donc pas de potentielles cibles du branchement indirect.

Autres techniques Il existe bien d'autres techniques [44] pour résoudre les branchements indirects, mais celles-ci demandent pour la plupart une analyse approfondie du programme, ce qui demande un coût en temps et mémoire. Cela les rend pratique lors d'une instrumentation dynamique hors ligne, mais peu utilisables pour de l'instrumentation dynamique en temps réel. De plus, certaines techniques sont fortement couplées à la chaîne de compilation utilisée pour compiler le programme tandis que d'autres sont applicables uniquement pour certains environnements d'exécution. Il existe cependant une technique notable et applicable à tous les programmes [45]. L'idée est d'usurper les branchements indirects pour un branchement direct vers un gestionnaire. Ce dernier vérifie la cible du saut indirect qui est connue durant l'exécution et détermine si le branchement peut être fait ou bien s'il faut effectuer le

branchement ailleurs.

Mesures de sécurité

Concernant les systèmes ayant un modèle de sécurité WXE, il semble qu'aucune recherche n'ait été effectuée en ce sens pour proposer des solutions à ce problème. Il est cependant possible d'avoir deux pages virtuelles vers la même page physique. Une de ces pages serait en mode exécution et l'autre en mode écriture. Cela est hautement dépendant du système d'exploitation, mais sur Linux cela est possible avec l'appel système `memfd_create(2)` [46].

Pour les protections du type Intel CET, la solution serait de ne simplement pas instrumenter les instructions `endbr64`, mais aussi d'utiliser des appels systèmes pour contrôler les piles fantômes du processus ou simplement désactiver la mesure de sécurité durant l'instrumentation [47].

CHAPITRE 3 DOMAINE DE LA PROBLÉMATIQUE

À la suite de la présentation de l'état de l'art en termes d'instrumentation dynamique, il est possible de caractériser le domaine de la problématique que tentent de résoudre les précédentes recherches.

Le nerf de la guerre en ce qui concerne l'instrumentation dynamique semble être – du moins sur `x86-64` – le degré de liberté associé à l'origine d'une sonde. En effet, plusieurs algorithmes [27,36,39,40] ont été proposés pour augmenter le nombre de branchements possibles à partir d'un emplacement dans un programme. Cela est dû à la taille variable des instructions sur une architecture telle que `x86-64`. Cela n'est pas un problème sur des architectures qui ont des tailles d'instructions uniformes. Par exemple, sur SPARC 32 bits, chaque instruction fait 32 bits et est alignée de la sorte. Dans son jeu d'instructions, un branchement utilise toujours un décalage relatif au compteur de programme. Celui-ci est multiplié par quatre, car toutes les instructions sont alignées sur 32 bits, et est additionné au compteur de programme pour obtenir l'adresse cible. Il est donc possible sur cette architecture de faire un branchement n'importe où dans le programme, car l'espace d'adressage virtuel est de 4 gigaoctets, ce qui est encodable sur 32 bits. En somme, chaque instruction dans un programme a un degré de liberté maximal. Cette caractéristique de l'architecture SPARC 32 ne s'applique pas forcément à d'autres architectures de type RISC comme ARM. Il n'est donc pas suffisant pour caractériser le problème de considérer uniquement le degré de liberté. Il s'agit en fait d'une seule portion du branchement. La seconde portion – où se trouve le trampoline – n'est presque jamais abordée et il semble être pris pour acquis qu'une simple recherche par la force brute avec l'appel système `mmap(2)` fonctionne. Or, cette seconde partie est critique pour toutes instrumentations dynamiques et ce pour toutes les architectures, même SPARC. En considérant les deux parties du branchement, on peut nommer et quantifier la première problématique comme étant **la couverture** de l'instrumentation dynamique qui est mesurée comme le rapport des instructions dans un binaire [1] où il est possible de faire une instrumentation sur le nombre total d'instructions dans ce binaire. On y intègre la notion de branchement entre l'origine de la sonde et son trampoline ainsi que l'allocation en mémoire du trampoline. Elle comprend donc deux sous-problèmes :

1. Les degrés de liberté des branchements associés aux origines des sondes.
2. L'emplacement des trampolines et leur allocation dans la mémoire du processus.

1. Programme ou bibliothèque se trouvant en mémoire. Sur les sysèmes UNIX, on parle de fichier ELF.

Un autre problème souvent abordé est la modification de l'exécutable pendant son exécution concurrente sur d'autres processeurs. Différentes approches ont été proposées [27, 36, 38, 40] mais soit elles ne sont pas valides dans tous les cas, ou alors elles sont fortement couplées à des modèles précis de processeurs. De plus, toutes ces méthodes n'adressent pas d'éventuels problèmes sur des systèmes ayant des mesures de sécurité telles que WXE. On peut donc nommer la seconde problématique comme étant **l'exactitude et la portabilité** de l'instrumentation dynamique. On entend par exactitude la qualité de conserver l'intégrité du programme pendant et après son instrumentation, et par portabilité sa capacité à pouvoir être utilisée sur plusieurs systèmes.

Une des conséquences de la préservation de l'intégrité du programme est l'introduction de primitives de synchronisation qui ont un impact non négligeable sur l'exécution du programme. De plus, l'exécution d'une sonde demande un temps de calcul. Ces deux surcoûts en temps forment une autre problématique qu'on nomme **le surcôut temporel** de l'instrumentation dynamique. De façon orthogonale, il y a aussi un surcoût associé à l'utilisation de la mémoire du processus, par exemple les allocations des trampolines. C'est une autre problématique qu'on nomme **le surcoût spatial** de l'instrumentation dynamique. Ces deux surcoûts sont étroitement liés puisqu'il arrive souvent que l'un soit privilégié au détriment de l'autre. C'est le compromis temps-mémoire ; on nomme la problématique qui englobe les deux dimensions **le surcoût spatio-temporel**.

En résumé, les recherches précédentes se sont surtout concentrées sur les problématiques de **la couverture** et de **l'exactitude**. Les problématiques de **la portabilité** et **du surcoût spatio-temporel** quant à elles ne sont que partiellement abordées ou carrément délaissées dans certains cas.

Le prochain chapitre présente l'unification des techniques présentes dans l'état de l'art sous la forme d'une bibliothèque en espace utilisateur. Cette bibliothèque incorpore aussi de nouveaux algorithmes et alternatives pour pallier les lacunes de ces techniques. Il s'agit du texte verbatim d'un article scientifique soumis à `Software: Practice and Experience`.

CHAPITRE 4 ARTICLE 1 : DYNAMIC PATCHING OF BINARIES IN USERSPACE

Écrit par

Olivier Dion
Mathieu Desnoyers
Mohammad Nassiri
Michel Dagenais

Soumis à `Software: Practice and Experience` le 2023-02-27.

Abstract The insertion and removal of new or existing tracepoints at runtime in a program is an increasingly popular technique called dynamic instrumentation. However, mutating the natural control flow of a program brings several challenges, *e .g.* system security measures, runtime overhead and memory usage. Our proposed solution is in the form of a userspace library called `Libpatch` which unifies and improves the latest algorithms found in the literature, and proposes new ones. Our experiments on a wide range of executable programs show that `Libpatch` can achieve an instrumentation success ratio of around 99.1% in runtime at the cost of a slight overhead of about 32 ns for the execution of an empty tracepoint.

4.1 Introduction

Modern software packages are ever increasing in their computation complexity. With the decline of the Moore's law, developers can not anymore only rely on the next hardware generation for gains on execution speed. Furthermore, the added complexity introduces new types of bugs that are hard to diagnose, such as race conditions. Instrumenters are tools that are specialized in software tracing. Usage examples for an instrumenter include profiling the execution of a program, in order to find execution bottlenecks, or tracing the events that lead to a very hard to reproduce bug. Thus, the key characteristics looked after for instrumenting a program is to be able to gather information about its state at certain points of its execution, while minimizing the impact on the runtime. We can represent the instrumentation of a program, from an instrumenter, with a specification, which is a set of tracepoints. Tracepoints can be seen as procedures at certain locations in a program that, when executed, compute some information derived from the state of the program at these

points. That information will later be consumed and reported to the end users.

An instrumentation is static when its tracepoints are either manually placed by the developers in the source code, when tracepoints are automatically generated by the compiler via some option or when a transformation is applied on a binary file to insert new tracepoints. The main characteristic of static instrumentation is its embedded tracepoints in the binary file. It is fast and persistent but not flexible in that no new tracepoints can be created the moment the program has been loaded in memory.

Dynamic instrumentation, on the other hand, is when the specification of the instrumentation is changed at runtime. This can be done at load time, *i.e.* before the program main routine. This can also be done concurrently to when the program is running. We will refer to them respectively as pre-loaded dynamic instrumentation and concurrent dynamic instrumentation. The main characteristic of dynamic instrumentation is its ability to generate new tracepoints the moment the program has been loaded in memory. It is slower than static instrumentation and is volatile.

We are interested here in the concurrent form of dynamic instrumentation, and its techniques. The main contributions of this paper are new algorithms, as well as a combination of the best aspects of other algorithms, found in the literature, to achieve efficient and flexible concurrent dynamic instrumentation. The result is presented as a shared library in userspace, that can be loaded or injected at any point during the execution of a program, in order to perform concurrent dynamic instrumentation.

This paper is divided as follows. A summary of the state of the art is presented in Section 4.2. We then formalize the problem domain in Section 4.3. Our proposed solution is thereafter presented in Section 4.4. The results are shown and discussed in Section 4.6, followed by the conclusion in Section 4.7.

4.2 Related Work

Tools like `LTTng` and `ftrace` are able to consume dynamic instrumentation points such as the ones provided by `Kprobe` and `Uprobe`. However, the former only applies for instrumenting the Linux kernel executable while the latter imposes a great overhead on the program runtime. Tracers like `lttng-ust` and `uftrace` would benefit from having new ways of consuming instrumentation points in userspace. Thus, this section presents the state of the art when it comes to dynamic instrumentation of binaries in userspace.

4.2.1 Dyninst

`Dyninst` [5] is a tool capable of static binary rewritting as well as concurrent dynamic instrumentation. It works by reconstructing the full Control Flow Graph (CFG) of the program and its dependencies. Typically, instrumentation in a function is done by putting a branch at its entry to a relocated variant. This variant of the function is an augmented CFG with the requested instrumentation. The modification of the program during its execution is achieved using the `ptrace(2)` system call on Linux, where all threads are stopped, *i.e.* it follows a *stop of the world* approach. Reclamation of function variants is done by unwinding the stack of each thread in the program to determine if it is currently executing such regions. The major drawbacks of this approach are the memory usage and latency imposed on the program while Dyninst extracts the CFG and patches the program.

4.2.2 Kprobe

Although `Kprobe` [15] was developed for instrumenting the Linux kernel, some of its techniques can be applied in userspace. By default, Kprobe will place a trap at the probe origin. Using a trap in kernel mode is less problematic, since the context switching overhead is smaller in kernel space. However, when available for the target architecture, and applicable for the target instruction location, the trap is replaced by a jump to a detour buffer that emulates a trap and calls the instrumentation. Thereafter, it executes out of line the replaced instructions – with their displacements possibly changed – and returns to the program. This optimization is only used if some safety checks are met. That includes checking that no relative branches can jump into the region, at the exception of the first byte, that there is no indirect jump or instruction that can trigger an exception in the instrumented function, that the instructions in the replaced region lie in the same function and that they can all be executed out of line. Thus, `Kprobe` follows a conservative approach to ensure system integrity at the expense of possible runtime overhead.

4.2.3 Dyntrace

The first problem for jump based instrumentation on `x86-64` is that instructions are of variable size. If the size of an instruction is greater or equal to a five bytes relative branch on that architecture, that single instruction can be simply replaced. However, if that is not the case, it is then necessary to replace multiple instructions to form a region that is big enough. Furthermore, some of these instructions could be a basic block entry, and thus jumps in the middle of the replaced region would be possible. `Dyntrace` [27] proposes to use a one byte trap

instruction for each head of the instructions that composed the region, with the exception of the first, which will be the opcode for a jump. By doing so, an instrumenter can be less conservative about where an optimization can be done, while ensuring program integrity. Indeed, any jump to an instruction in the middle of the region will trigger a trap that will be handled appropriately. On the other hand, the usage of these traps greatly limits the possible offsets that can be encoded, thus limiting the possible branch destinations. Consider for example a five-byte instruction versus two instructions of three bytes. The former has 256^4 possible destinations, while the latter only has 256^3, a significant difference.

The second problem is that instructions in the replaced region must be executed someplace else. This is typically done in an Out of Line eXecution (Out of Line eXecution) buffer. The problem arises when instructions that are dependent on the program counter have to be relocated. Despite the fact that it is possible to recompute the displacement of such instructions, this greatly limits where the OLX buffers can be placed in memory. One way could be to merge the OLX buffer with the trampoline at the jump destination, since the latter is already constrained in its location, but `Dyntrace` proposes instead to emulate such instructions. For example, it may use another register as the base register, with the value of the program counter stored in it.

The last problem is the replacement of the region with a jump to the trampoline. `Dyntrace` does this by executing a `cmpxchg8b` instruction. At first glance, this technique seems valid but, unfortunately, it does not respect the cross-modifying protocol stipulated by the vendors [8, 9]. In that protocol, a serializing instruction, such as `cpuid`, must be executed before the execution of the modified instructions. This implies the modification of other instructions, for inserting such serialization, leading to a paradox. Althought [9] stipulates that it could be valid if the store is performed atomically on a naturally aligned quadword, this would fail if the region overlaps two lines of cache.

4.2.4 Liteinst

To circumvent the limitations of `Dyntrace`, with its embedded traps in the instrumented region, `Liteinst` [36] proposes to also use one-byte illegal instructions [34, 35], which they call instruction punning. On `POSIX` systems, one simply has to handle `SIGILL` along with `SIGTRAP`. This greatly augments the search space of possible jump destinations for trampolines, while keeping the integrity of the program.

It proposes a cross-modifying protocol [38], that uses a time grace period, to wait for the propagation of the information through the system. Although this method imposes less synchronization overhead, the grace period is determined empirically and is highly microar-

chitecture dependent. It also proposes to use a light synchronization primitive offered by the operating system – like `membarrier(2)` on Linux – if available.

4.2.5 E9Patch

`E9Patch` [39] is a tool that does static instrumentation in the form of binary transformation. However, some of its techniques, for extending the `Liteinst` instruction punning, are applicable to concurrent dynamic instrumentation. Its first technique, called pad jump, uses redundant prefixes for the branch instruction. By offsetting the jump instruction in that way, it is possible to reach more bytes at the tail of the region. In fact, it may be possible to reach the full search space, as if the first instruction was five bytes. Its second technique, called successor eviction, replaces the instruction that follows the first instruction with a branching instruction to a secondary trampoline. Then, the first instruction can use these new bytes as new byte patterns by applying the first technique for branching to a primary trampoline. In that case, there is a strong dependency between the two trampolines. Its last technique, called neighbour eviction, replaces an instruction further ahead of the first instruction, and uses it as a jump to a trampoline. Then, the original instructions can be replaced with a two-byte relative branching to the evicted neighbour.

4.2.6 NOProbe

The neighbour eviction technique from `E9Patch` incurs potential false instrumentation. By that we mean that a tracepoint could be executed even when not hit. In fact, it is not possible to determine if the tracepoint was called due to a branch at the intended location, or due to another control flow path. This has the potential of generating incorrect instrumentation information. To overcome this issue, it is possible to hijack unused bytes in the text segment that are close enough to the patch origin. That is what `NOProbe` [40] proposes to do, taking advantage of a type of compiler optimization. Indeed, compilers often align the start of functions, loops and labels on cache boundaries, for performance considerations. On `x86-64` these alignments are done with NOP instructions. The assumption is that these alignments are not used by the program and contain no embedded data or obfuscated instructions. If this holds true, then these regions can be used as candidates for the neighbour eviction technique, with the advantage that they are only executed when the tracepoint is actually hit.

`NOProbe` also proposes a protocol for cross-modification of the program. By using `POSIX` realtime signal it is possible to force threads in the program to execute a serializing instruction such as `cpuid` in a signal handler, followed by an acknowledgment. This solves the chicken-and-egg paradox for the cross-modifying protocol. There is however a strong assumption in

that no thread will mask the signal. Otherwise, the instrumenter could wait indefinitely for the acknowledgment. This could be fixed on Linux with the `ptrace(2)` system call to change the signal mask of threads.

4.3 Problem Domain

Taking into account the related work, we can identify the remaining gaps and formalize the problem domain for concurrent dynamic instrumentation.

Coverage We define the instrumentation coverage as the ratio of instructions that can be instrumented, over the total number of instructions in a program. We aim at maximizing that rate.

Runtime overhead We aim at minimizing the runtime overhead, imposed on the program by the dynamic instrumentation, in terms of space and time, at instrumentation time and at execution time.

Software integrity Dynamic instrumentation mutates the natural flow of the program. If done incorrectly, it can lead to catastrophic results. We aim at guaranteeing the integrity of the software, while leaving the liberty for instrumenters to modify the state of the program, *e.g.* for security hot fixes.

Portability We aim at supporting as many Linux distributions as possible. This includes conforming to the System V ABI (Application Binary Interface) [11] and system security features such as WXE (Write Xor Execute) and Intel CET (Control-flow Enforcement Technology) [13]. We also aim at supporting most popular hardware architectures.

4.4 Design and Implementation

Our implementation called `Libpatch` [48] aims at solving the problem domain. It only supports Linux and the `x86-64` architecture for now. However, it is for the most part architecture independent. We define a patch to be a derivation of an instrumentation specification which contains all the necessary information – possibly architecture dependent – to implement the instrumentation. Its application to the program, or patching, is a mutation of the process to reflect such specification. `Libpatch` is not an instrumenter, in that it does not do any useful

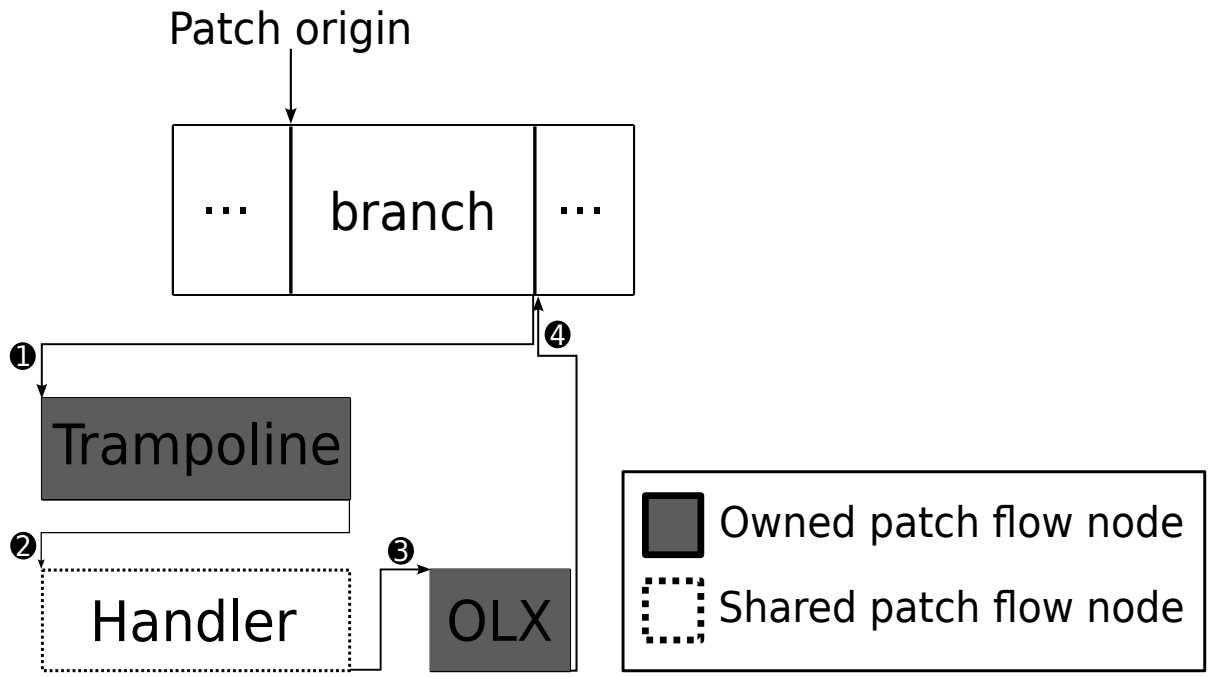
computation derived from the state of the program, but it aims at providing the necessary abstraction so that various dynamic instrumenters can use it at their core.

4.4.1 Patch flow

A patch flow is the control flow introduced in the program by a patch. It is composed of *owned* and *shared* nodes, as shown in Figure 4.1. We say that a node is owned because it is specific to a patch, *i.e.* it contains information that is directly derived from the patch. A shared node, on the other hand, is generic enough that it can be used by many patches.

The flow starts at the patch origin, where the original instructions were replaced by a branching instruction to the trampoline. From there, the information required for the instrumentation is loaded and a call is issued to the patch handler. The handler job is to save the computation context of the program, execute the user-defined probe – the instrumentation – and restore the possibly modified context. Before jumping further to the computation, it returns to the OLX buffer, which emulates the replaced instructions at the patch origin. The main difference between shared and owned nodes is that the former are statically allocated in `Libpatch` as opposed to the latter which are dynamically allocated. Memory reclamation of a dynamically allocated executable region is a hard problem, since any number of threads in the program may be executing it at any point in time. Thus, the owned nodes are marked for reclamation when a patch is reverted. The actual reclamation is done by the `Libpatch` garbage collector, when convenient for the instrumenter.

Figure 4.1 Patch flow.



4.4.2 Jump pattern algorithms

`Libpatch` uses a branch at the patch origin to reach the trampoline. On `x86-64`, this branch is encoded on a five-byte relative `jmp`. Thus, the branch can use four bytes to encode 256^4 offsets. However, this can only be done when the instruction at the patch origin is at least five bytes. We call this the **fit** algorithm. For cases where that algorithm cannot be applied, `Libpatch` falls back to 3 other algorithms. These are presented in the order in which they are attempted.

The primitive used by these algorithms is what we call a jump pattern, which is a vector of byte patterns. A byte pattern is either any value between 0 and 255 inclusively, or a set of fixed values in the same range.

NOP jump

The algorithm presented in `NOProbe` is used, inserting a two-byte relative jump to a padding hole. A hole is searched in between the procedure of the patch origin, and the previous or next procedures in memory. That hole is acquired by the patch and a five-byte jump to the trampoline is put in it, adding an owned node to the patch flow. However, if the patch origin is the last instruction of a function, then the hole that directly follows – if any and large enough – is used to encode part of the offset of a five-byte jump, thus bypassing the two-byte

jump. We call that special case **nop-fit**. If successful, this algorithm can generate 256^4 jump patterns, but possibly add an extra branch. Holes are found by scanning the `DWARF` debug symbols table of the program.

Alias

This algorithm is a special case of the punning algorithm presented by `Liteinst`. Instructions next to the patch origin are used as part of the offset of a five-byte jump, without any modification. Thus, they serve a dual purpose of being still valid instructions for the program, while being part of the jump offset, depending on the flow of execution. We also leverage the idea of redundant prefixes from `E9Patch`, to shift the patch origin to the right. The best and worst cases of this algorithm are respectively 256^3 and 1 jump patterns.

Punnings

The last resort is the punning algorithm, as presented by `Liteinst`. We however augment the set of one byte illegal instructions [34,35] that they used from 14 to 22, thus generating 23 possible combinations when taking the `int3` trap into account. Furthermore, we also exploit the `E9Patch` idea of redundant prefixes to shift the patch origin. This algorithm can then generate between 23^4 and $256^3 \times 23$ jump patterns. However, with the redundant prefixes, it is possible in some cases to generate all the possible patterns.

4.4.3 Patching protocol

Modifying program instructions during their execution by concurrent threads is prone to undefined behavior. Vendors are clear on that [8–10]. To circumvent this, we propose two solutions.

The first solution can be applied to the `x86-64` architecture and probably others. The protocol is a modified version of the one proposed by `NOProbe`. The main difference is the usage of the `ptrace(2)` and the `membarrier(2)` system calls to ensure threads synchronization, instead of using `POSIX` realtime signals. The protocol is split into five steps and is the same for patching and unpatching, at the exception of the threads' eviction step, which is only required during patching. A step is done in batch, *i.e.* it is applied to all the regions, before going to the next step. All threads in the program are not interrupted by `Libpatch` during the patching. At the exception of the threads' eviction step, there are calls to the `membarrier(2)` system call with the `MEMBARRIER_CMD_PRIVATE_EXPEDITED_SYNC_CORE` command, for core synchronization between steps. The steps are detailed below in the order they are applied.

At every step, we show the progression – and its reverse – of patching a region, represented in Figure 4.2. Thus the initial values of the regions for when patching and after patching are respectively `55 48 89 E5 48 31 C0` and `E9 -- -- -- -- XX XX` where the `-` represent the branch offset digits, which could be anything. Note that we have set the unused bytes in the patching region to `XX`, for clarity, but it could also be anything.

Figure 4.2 Patching of the prologue of an unoptimized procedure.

```
push   %rbp       ;; 55
mov    %rsp, %rbp ;; 48 89 E5
xor    %rax, %rax ;; 48 31 C0
;; becomes
jmp --------      ;; E9 -- -- -- --
XX                ;; XX
XX                ;; XX
```

1. Lock patches

   This step locks the regions in place. For this, we put `int3` at the head – the first byte – of each instruction that compose a region, when patching. In that case, the order does not matter and there is no consequence if the region overlaps multiple cache lines. It is however different for unpatching since there is a dependency between the first instruction and the rest. In that case, the order does matter, and we do the same as with patching, but in two sub-steps, with a core synchronization in between. First, a trap is placed at the first byte of the region. This removes the jump. Then, traps are placed for the head of the other instructions, locking the jump offset. Consider what would happen if it was the other way around. A thread could execute the jump while its offset is being modified, resulting in undefined behavior.

   The idea here is that trap insertion is a special case, typically used by debuggers to insert breakpoints. It is directly supported by most architectures, without requiring a cross-modifying protocol [9]. If a thread jumps to an instruction for which the first byte was replaced by a trap, the trap handler will take control and be in a position to properly handle that case.

   Once the full region is locked, any thread trying to execute any of the instructions in that region will trigger a `SIGTRAP`, which is handled in `Libpatch` by moving the thread to the corresponding offset in an OLX buffer.

   Patching

   55 48 89 E5 **48** 31 C0 → **CC CC** 89 E5 **CC** 31 C0

   Unpatching 1

   E9 – – – – XX XX → **CC** – – – – XX XX

Unpatching 2

CC – – – – XX XX → CC **CC** – – **CC** XX XX

2. Threads eviction

   This step is only required while patching, and does not modify any of the regions. Instead, the set of threads in the process is iterated over. For each thread, two conditions are checked: first, if the thread program counter is within a patch region, and second if the thread is in a signal handler that will return to a patch region. If any of these conditions are met, then the thread is evicted from its location and is moved to the corresponding offset in a OLX buffer. This is done by interrupting the thread with `ptrace(2)` and unwinding its stack frames, which can reliably be done on optimized binaries if either the `.eh_frame` (if compiled with asynchronous unwind tables) or `.sframe` sections are in the ELF file. If both sections were omitted during compilation, it would be possible to fallback to the `.debug_frame` section from `DWARF`. Some ABIs, like for ARM, have mandatory descriptions in the binary for unwinding the stack [49], making it possible to unwind the stack with frame pointers. Thus, we assume that programs instrumented by `Libpatch` are not obfuscated in some way to prevent stack unwinding.

   The reason why this step is necessary is because a thread could be preempted in the middle of a patch region. If it is rescheduled after the patching is done, then the thread will execute whatever composes the branch offset as an instruction. This is not a problem if the original instruction is at a location of a basic block entry, since `Libpatch` will force the usage of a trap or an illegal instruction at that location, triggering the same signal handler as in the first step. We also do not need to check for nested calls, that is if the thread will eventually return to the patch region, because we consider the instruction following a call to be a basic block entry anyway.

3. Bodies patching/unpatching

   In this step, the body of every instruction is set to its final value. By body we mean everything except the head.

   Patching

   CC CC **89 E5** CC **31 C0** → CC CC – – CC **XX XX**

   Unpatching

   CC CC – – CC **XX XX** → CC CC **89 E5** CC **31 C0**

4. Heads patching/unpatching

   In this step, we set the final value of every instruction head, except for the first instruction. We indeed need to do this before we change the first instruction head

to the jump opcode, since the other instructions are merged, becoming part of the branch offset. This is not required for unpatching since we are not merging several instructions into one.

Patching

CC **CC** – – **CC** XX XX → CC – – – – – XX XX

Unpatching

CC **CC** 89 E5 **CC** 31 C0 → CC **48** 89 E5 **48** 31 C0

5. Unlock patches

   The last step unlocks the patches by setting the final value of the head of the first instruction.

   Patching

   CC – – – – – XX XX → **E9** – – – – – XX XX

   Unpatching

   CC 48 89 E5 48 31 C0 → **55** 48 89 E5 48 31 C0

The second solution can be applied to any architecture where the first solution can not be used. For example, this is the case for the ARM architecture, where only a certain class of instructions can be safely modified atomically by a thread, while being executed concurrently by other threads [10]. It is, however, not currently implemented by `Libpatch`. The idea is to stop the world, interrupting the computation of all threads in the program. Then the patching is applied, followed by any synchronization mechanism, *e.g.* instruction cache flush, required by the architecture. Since stopping of the world is a very disruptive operation on the process, we propose to minimize its impacts by focusing on reducing its time frame. For this, a copy of the bounded regions to patch (all the pages from the lowest to the highest addresses of the regions) is made. This copy is stored in an anonymous file, such as with the `memfd_create(2)` system call on Linux. Then, the patching is applied entirely on that copy. Thereafter, all the threads in the program are stopped and the copy contained in the file is mapped at the original location using the `MAP_FIXED` option of the `mmap(2)` system call. This will effectively change the physical pages of the mapping, replacing the original pages with the patched copy. Finally, the threads can be resumed, after proper architecture dependent synchronizations are executed. This solution represents a tradeoff between a reduction of the time frame during which all the computations are stopped, and a peak of memory allocation before the new physical pages are mapped and replace the older ones. It is also difficult to implement, since a thread must be able to do a mapping in the program space, but all threads must be stopped. We thus propose to use a special thread, created using the `CLONE_VM` but

not the `CLONE_THREAD` flags of the `clone(2)` system call. This will allow the special thread to interrupt the threads of the program using the `ptrace(2)` system call, while being able to modify its virtual mapping.

4.4.4 Trampoline allocator

Finding a location for a trampoline is a hard problem, since they have to be close enough to the patch origin, such that the relative jump offset can be encoded. Moreover, the search space can be enormous and sparse. Furthermore, space is limited and there are overlapping solutions between the different patches. For these reasons, we have implemented the allocator as a set of memory pools. We also represent the allocation request as a specification composed of a maximal alignment, base address, and a jump pattern. The key idea is to try to do a pattern matching of the jump pattern onto the offsets, derived from the base address and the chunks that compose a pool.

Pools layout

Pools are pre-allocated in a safe manner, with a `mmap(2)` system call for each pool, using the `MAP_FIXED_NOREPLACE` flag, to avoid any possible collision with existing mappings, and reduce memory fragmentation. Pools are placed at strategic places so that instrumentation can be done, whether the program or any of its libraries was compiled with PIE (Position Independent Executable). A pool placement example is shown in Figure 4.3. Pools are allocated in anonymous files using the `memfd_create(2)` system call, allowing to set the pathname of the mapping, which has a `(deleted)` suffix after closing the underlying file descriptor. The `null` pool is typically used for instrumenting the program. The `lib` pool is used for instrumenting leaf dependencies. The `stack` pool is used for instrumenting top dependencies, *i.e.* the ones that are loaded first by `ld-linux(8)`.

A single pool is around 1000 pages. The last page is reserved for keeping the addresses of the handlers, which must be close enough to the trampolines so that they can be accessed with relative addressing. The rest of the pages are divided in chunks, each being the size of a level one instruction cache line. Each chunk is represented by a bit within a bitmap. If set to one, then the chunk is free, otherwise it is used. The division in chunks of that size simplifies the allocator by using a bitmap and avoids any possible false sharing between trampolines.

Figure 4.3 Pools placement example on a program compiled with -pie.

```
address                   pathname
555555164000-555555554000 /memfd:libpatch:null-trampoline (deleted)
555555554000-555555555000 a.out
...
555555559000-5555555e3000 [heap]
...
7ffff6df2000-7ffff71e2000 /memfd:libpatch:lib-trampoline (deleted)
...
7ffff7fc8000-7ffff7fcc000 [vvar]
7ffff7fcc000-7ffff7fce000 [vdso]
...
7ffff7fff000-7ffff83ef000 /memfd:libpatch:stack-trampoline (deleted)
7fffffde000-7ffffffff000 [stack]
```

Trampoline layout

A trampoline can be represented by the pseudo structure shown in Figure 4.4. The structure contains `instructions` to reach the handler. The padding fields are used to correctly align the other members of the structure. Their size is variable, but the total size of the pseudo structure is always less than or equal to the size of a trampoline pool chunk. Thus, a trampoline always fits within a level one instruction cache line. Furthermore, the padding in front serves to allow more flexibility in the jump pattern by increasing the maximal alignment. Then there is the `trampoline_descriptor` shown in Figure 4.5 which contains all the read-only data necessary to execute the instrumentation. The packed attribute and the order of the members are necessary to ensure natural alignment of all members. This is because the trampoline descriptor is aligned as such in the trampoline.

Figure 4.4 Pseudo C structure of a trampoline.

```
struct trampoline {
  uint8_t padding[];
  uint8_t instructions[];
  uint8_t padding[];
  struct trampoline_descriptor desc;
};
```

Figure 4.5 C structure of a trampoline descriptor.

```
struct trampoline_descriptor {
  uint8_t      has_post_probe:1
  uint8_t      continuation_offset;
  patch_probe  probe_procedure;
  void        *user_data;
  uintptr_t    olx_address;
  uintptr_t    real_pc;
} __attribute__((packed));
```

4.4.5 Trampoline instructions

On `x86-64` the generic trampoline instructions are shown in Figure 4.6. The first instruction is used to preserve the red zone, thus conforming to the System V ABI [11]. The next two instructions load the address of the trampoline descriptor into the `rdi` register. Finally, the indirect call is used to branch to the handler. The two displacements are variable. `DISP` depends on the padding size between the instructions and the trampoline descriptor, while `DISP2` depends on the positions of the call instruction and the handler address within the trampoline pool.

Figure 4.6 Generic trampoline instructions on x86-64.

```
lea -128(%rsp), %rsp
push %rdi
lea DISP(%rip), %rdi
call *DISP2(%rip)
```

Trampoline searching

Since it is possible to have 256^4 valid candidates for a patch, but only one could be free or even none, the search is optimized to prune quickly as many candidates as possible. The first step is to compute the page mask by using the least significant bits of the jump pattern that encodes an offset within a page. Assuming a page size of 4096 bytes, and that all chunks reserved for trampolines are aligned on a level one 64 bytes instruction cache boundary, then there are 64 possibilities per page, and the 12 least significant bits encode the page offset. Thus, the page mask can be encoded on a 64 bits value. If a bit is 1 in the mask, then it may be possible to reach the trampoline at the location represented by it within the page. Otherwise, it is not possible. To compute the page mask, the subpage mask encoded on the least significant byte of the jump pattern is calculated first. It is then propagate onto the

page mask, using the rest of the unused bits of the jump pattern, and possibly the carry of the submask calculation.

The second step is to iterate over the set of pools. For each pool, a free chunk is found by applying the following algorithm. First, a check is made to ensure that the pool is not too far away from the base address. Then, the most significant byte of the jump pattern is pruned. Indeed, considering that the pool chunks are contiguous, then the offset from the base address of the first chunk to the last chunk can only increase. Knowing that, it is possible to prune earlier many solutions that are not in between these two extreme offsets for the most significant byte. From there, the set of pages that can be reached using the 20 most significant bits is made, assuming pages of 4096 bytes. In other words, a pattern matching is applied on the five most significant nibbles of the jump pattern. Finally, the set of pages that can be reached is iterated over, their corresponding bitmap are logically `AND` onto the page mask. Finally, the first least significant bit set to 1 from the set of bitmaps is found, to get a free chunk.

Consider for example a patch origin at address `0xABCEF` where the instruction is 4 bytes. To form a complete 5 bytes region, the next instruction is used. Consider also that the next instruction is the entry of a basic block, thus a trap is used to lock that instruction head. Other one byte illegal instructions could be used for more possibilities, but this will not be considered in this example, for the sake of simplicity. Then, the patch region can be representend as `E9 -- -- -- CC` where `--` is any byte and the base address is `0xABCF4`, since the branch instruction takes into account the size of the instruction. Because it is a little-endian architecture, the `CC` byte forms the most significant bits, and the bits that encode a page offset can be anything. Thus, the page mask is full. Consider now that a single trampoline pool exists, and its chunks span from `0xCC000000` to `0xCC3EEFC0`. The set of pages to iterate over is determined by matching the most significant bits up to a page onto the jump offset. The two differences from the pool boundaries and the base address are `0xCBF5430C` and `0xCC3432CC`. We see that the beginning of the pool does not match the most significant bits of the jump pattern, but the end does match. Thus, the page set will be reduced from 1007 pages (`0xCC000000 .. 0xCC3EEFC0`) to 835 pages (`0xCC0ABCF4 .. 0xCC3EEFC0`). With 64 chunks per page, this gives a total of 53440 possible solutions.

Now consider the same scenario but with a patch region with 3 instructions, where the first two instructions are 2o bytes each, and the last 2 instructions are basic block entries. Thus, the patch region can be represented as `E9 -- CC -- CC`. The page mask is now derived from the address `0xABCF4` and the bits `b--------1100`. To compute the page mask, the subpage mask is first computed by using the least significant bits. Since they can be anything, the sub-

page mask is `0xf` and the carry is set to on. To propagate the submask, the most significant bits are added together to determine the position where the submask has to be propagated. In this case, `((0xABCF4 + (0xCC << 8)) >> 8) & 0xf = 0x8`. This value must be multiplied by 4, since the subpage mask and its position are encoded on 4 bits, and that the page mask is 64 bits (`64 / (4 * 4) = 4`). The value can then be used to shift in place the submask and the result is logically `OR` onto the pagemask. Thus, `pagemask |= (0xf << (4 * 0x8))`. Since the carry is on, the operation is repeated but with the most significant bits plus one. Thus, `pagemask |= (0xf << (4 * (((0xABCF4 + (0xCD << 8)) >> 8) & 0xf)))`. The final page mask is `0xff00000000` which means that it is perhaps possible to reach the 33rd to the 40th chunks in any page, but it is not possible to reach any other chunk. The set of pages is now calculated in the same manner as before. With the new constraint imposed, the set of pages is now discontinuous and is reduced from 835 pages to 208 pages. With 8 (page mask weight) possible chunks per page, this gives a total of 1664 possible solutions.

4.4.6 Handlers

Handlers act as the glue between the program and the instrumentation. They are statically allocated in `Libpatch` and their primary job is to execute the instrumentation while preserving the context of computation of the program. There are currently two handlers available, but more could be created dynamically, depending on the requirements of the instrumentation. For example, in some cases it would be possible to reduce the number of registers saved/restored in some steps, saving time.

Figure 4.8 shows the generic handler on `x86-64`, which is the most flexible, but also the slower, saving/restoring every general-purpose register. Each of its steps are detailed below. Note that CFI directives are used for generating critical information for the garbage collector about the CFA (Canonical Frame Address) in the `.eh_frame` section of the ELF file of `Libpatch`. Similarly to the thread eviction step in the patching protocol, we assume that it is always possible to reliably unwind the stack frames.

Furthermore, this handler supports the optional execution of a second probe after the execution of the OLX buffer. Since the pointer to the trampoline descriptor is lost in between the execution of the probe and the post-probe, the necessary information for the execution of the latter is stored in a thread local storage.

A use-case for a post-probe is when an instrumenter would do a bound checking of a memory reference through a poisoned pointer. In that case, the instrumenter would remove the poison on the pointer after doing the bound checking, allowing the execution of the OLX buffer, doing the memory reference without segmentation fault. The post-probe is then used to

poison the pointer again, so that the next memory reference in the program will trigger a segmentation fault. The fault signal handler would check the bounds and possibly patch the location where the memory reference occurs, so that the next time it is executed, the bounds check is done through instrumentation instead of a fault.

1. Allocating the probe context

   Execution of instrumentation in `Libpatch` is done through the execution of a callback – the probe – which receives a pointer to a context structure shown in Figure 4.7. This context is the state of the thread at the moment it entered the patch flow.

Figure 4.7 C structure of a probe context.

```
struct patch_probe_context {
  patch_reg program_counter;
  patch_reg stack_pointer;
  patch_reg general_registers[PATCH_ARCH_GREGS_COUNT];
  void     *user_data;
  patch_reg status;
};
```

2. Saving the computation context

   Every general-purpose register is saved in the context passed to the probe. This step is currently not saving the extended states, such as XMM on `x86-64`.

3. Setting up stack for C

   Before calling the architecture independent procedure of the handler, the calling convention has to be satisfied.

4. Calling the C handler

   This part of the handler is architecture independent and sets thing up correctly in order to call the instrumentation and getting the continuation address.

5. Storing the continuation

   The continuation address is stored in the stack, in order to change the call chain. If coming from the generic handler, then the continuation will be the address of the OLX buffer. However, if it is coming from the post generic handler, then the continuation is determined by the addition of the continuation offset with the program counter in the trampoline descriptor.

   This step is critical for the garbage collector to determine the transition from the trampoline to the OLX buffer.

6. Restoring the computation context

   The state of execution is restored at that point and could have been modified by the instrumentation.

7. Returning to the continuation

   The handler returns to the continuation, while unwinding the information pushed by the trampoline on the stack and restoring the red zone.

Figure 4.8 Generic handler for x86-64.

```
generic_post_handler:
      endbr64
      /* Emulate the trampoline behavior since coming from the OLX buffer. */
      lea -128(%rsp), %rsp
      push %rdi
      push %rax
      /* No trampoline descriptor. */
      xor %rdi, %rdi
generic_handler:
      .cfi_startproc
      endbr64
      /* 1. Allocate probe context. */
      pushf
      .cfi_adjust_cfa_offset 0x8
      sub $0x90, %rsp
      .cfi_adjust_cfa_offset 0x90

      /* 2. Save computation context. */
      movq %rax, 16(%rsp)
      movq %rbx, 24(%rsp)
      movq %rcx, 32(%rsp)
      movq %rdx, 40(%rsp)
      movq %rsi, 48(%rsp)

      /* Real %rdi was pushed onto the stack by the trampoline. */
      movq 0xa0(%rsp), %rsi
      movq %rsi, 56(%rsp)

      movq %rbp,  64(%rsp)
      movq %r8,   72(%rsp)
      movq %r9,   80(%rsp)
      movq %r10,  88(%rsp)
      movq %r11,  96(%rsp)
      movq %r12,  104(%rsp)
      movq %r13,  112(%rsp)
      movq %r14,  120(%rsp)
      movq %r15,  128(%rsp)

      /* Original stack pointer. */
      lea 0x128(%rsp), %rsi
      movq %rsi, 8(%rsp)

      /* 3. Setup C call. */
      movq %rsp, %rsi
      movq %rsp, %rbp
      .cfi_def_cfa_register %rbp

      /* Respect System V ABI 16-byte stack alignment. */
      andq $0xfffffffffffffff0, %rsp
      push %rbp
      push %rbp

      /* 4. Enter C handler. */
      call dynamic_tp_entry

      pop %rsp
      .cfi_def_cfa_register %rsp

      /* 5. Store continuation. */
      movq %rax, 0x98(%rsp)

      /* 6. Restore computation context. */
      movq 16(%rsp),  %rax
      movq 24(%rsp),  %rbx
      movq 32(%rsp),  %rcx
      movq 40(%rsp),  %rdx
      movq 48(%rsp),  %rsi
      movq 56(%rsp),  %rdi
      movq 64(%rsp),  %rbp
      movq 72(%rsp),  %r8
      movq 80(%rsp),  %r9
      movq 88(%rsp),  %r10
      movq 96(%rsp),  %r11
      movq 104(%rsp), %r12
      movq 112(%rsp), %r13
      movq 120(%rsp), %r14
      movq 128(%rsp), %r15

      add $0x90, %rsp
      .cfi_adjust_cfa_offset -0x90
      popf
      .cfi_adjust_cfa_offset -0x8

      /* 7. Return to OLX or computation. */
      retq $0x88
      .cfi_endproc
```

4.4.7 OLX buffers

The OLX buffers are used for executing relocated instructions. They are allocated using a pool allocator, similar to the trampoline pools, so that they are contiguous in memory, but their placement in memory is unrestricted, unlike for the trampolines. The pool allocator dynamically allocates pages as needed. Each page is divided in chunks of the size of a level 1 instruction cache line, and a single OLX buffer can span multiple chunks, preventing any false sharing between buffers. Furthermore, this allocation scheme greatly helps the memory reclamation by the garbage collector, since the OLX buffer is the last node part of a patch flow that could be removed by the instrumenter during its execution.

The anatomy of an OLX buffer consists of an instruction region, followed by a data region. When an instruction needs to be relocated, `Libpatch` validates whether that instruction is dependent on the program counter. If not, then the instruction is simply copied as is in the OLX buffer. Otherwise, an emulation of it – possibly with multiple instructions – must be constructed. In that case, the data region can be referenced to help the emulation. To do so, we have modified some of the algorithms proposed by `Dyntrace` such that they satisfy the System V ABI, including preserving the red zone.

For example, Figure 4.9 shows the emulation of an indirect jump that uses the program counter as its base. While this emulation conforms to the ABI by preserving the red zone, and the program integrity by preserving the `rax` register, it would be possible with register analysis to avoid such restoration, and thus improve its performance. This is, however, not currently implemented in `Libpatch`. Another example is shown in Figure 4.10 where `SRC` is not read nor written by the instruction `OP`. This generic case applies to almost all instructions that require emulation. Again, with register analysis, it would be possible to avoid the save/restore pairs.

Figure 4.9 Emulation of an indirect jump.

```
jmp *DISP(%rip)
;; becomes
lea -0x80(%rsp), %rsp   ;; save red zone
push %rax               ;; save context
movabs RIP, %rax        ;; get address
mov DISP(%rax), %rax    ;; read memory
xchg %rax, (%rsp)       ;; restore context and store result
ret $0x80               ;; branch to target and restore red zone
```

Figure 4.10 Generic emulation of a relative instruction.

```
OP DISP(%rip), DST
;; becomes
lea -0x80(%rsp), %rsp  ;; save red zone
push SRC               ;; save context
movabs RIP, SRC        ;; get address
OP DISP(SRC), DST      ;; do instruction
pop SRC                ;; restore context
lea 0x80(%rsp), %rsp   ;; restore red zone
```

4.4.8 Memory reclamation

When a patch is removed, the original instructions are restored at the patch origin using the proposed protocol. However, even if this protocol is secure in that it prevents any inconsistent view of the program, it is still possible for any thread to have entered the patch flow and not yet exit. Thus, it is not possible to reclaim any of the owned nodes of the patch flow of a patch until we are certain that no thread remains in it. We instead mark the patch flow for reclamation by the GC (Garbage Collector). The GC assumes that the patch flow entry has been closed, and thus no new thread can enter a patch flow that has been marked for reclamation. The key idea is, like `Dyninst` does, to unwind the stack of each thread in the program to determine if it is in a marked patch flow. However, unlike `Dyninst`, `Libpatch` does not follow a stop of the world approach. Instead, threads are interrupted individually, using the `ptrace(2)` system call, minimizing the impact on the program runtime. If a thread is in a marked patch flow, then it cannot be in any of the other ones. The GC uses that fact to take a decision. Depending on the policy chosen, it either loops until the thread quits the patch flow, or it simply does not reclaim the memory associated with that patch flow, leaving it for a next pass. After iterating over all threads, the marked patch flows free of threads can be reclaimed.

To further minimize the impact on the program of a thread interrupted by the GC, owned nodes of the same type are allocated in contiguous regions, for quick bounds checking and binary search. This motivates the allocation schemes for trampolines and the OLX buffers. Furthermore, stack unwinding requires information about the stack frames. This explains the usage of the CFI directives in the handler, the call from the trampoline to the handler, instead of a jump, and storing the continuation on the stack. By doing so, the GC can determine correctly whether a thread is in any owned node of a patch flow, at any time during the program execution, including during the handling of asynchronous system signals.

4.4.9 Runtime cooperation

When instrumenting a function entry/exit, `Libpatch` changes the return value of the current stack frame of the program, so that the function in the program returns to a `Libpatch` handler instead. The original value is then used to return to the correct location in the program. Since there is an unknown number of calls to other functions, in between the entry and the exit of a function, `Libpatch` pushes the original return value onto an internal stack that is allocated inside a TLS (Thread Local Storage). That value is then popped later and used as before. This technique has the advantage of working as long as the calling convention is followed by the runtime. It also natively supports tail call optimization. However, cooperation from the instrumenter with `Libpatch` is required for supporting special runtime functionalities. For example, an API is provided for communicating with `Libpatch` if the thread stack is being unwinded dues to an exception or a longjump. Another API is provided for coroutines. For example, the GO language implements coroutines in userspace, which requires stack switching. With the cooperation of the instrumenter, `Libpatch` can safely synchronize itself with the runtime by unwinding its internal stack or switch to another one.

4.5 Interaction with security measures

Systems can come with various security measures that aim at preventing certain types of attacks. Although aimed at protecting the system, these measures also affect the work of concurrent dynamic instrumentation. Indeed, dynamic instrumentation involves mutations of the natural control flow of the program and its instructions.

WXE

WXE is a security measure that prevent pages – in userspace and/or the kernel – to be writable and executable at the same time. To overcome this security measure, we propose two solutions. Both of them are Linux specific. Althought it is entirely possible to change the memory protection of pages using the `mprotect(2)` system call, this would involve segmentation faults, of threads executing such pages, between the protection changes. To circumvent this, the main idea is based on the assumption that memory protection is applied on virtual pages and not physical pages. Thus, it is possible to have two virtual mappings, one writable and the other one executable, to the same physical pages.

The first solution is when `Libpatch` needs to allocate executable regions that are private to the library, for examples trampolines pools and OLX buffers. To do this, a file descriptor is first created using the `memfd_create(2)` system call, and then truncated to the desired

size. Then, two mapping are created using the `mmap(2)` system call. One writable and one executable. The file descriptor can then be closed.

The second solution is when `Libpatch` needs to write to an already allocated executable region. For example, the main program. To do so, an anonymous file is created as in the first solution. Then, the executable memory is copied to the file. Finally, the underlying file descriptor is mapped twice. Once anywhere with write protection and once at the original location using `MAP_FIXED` with execution protection.

Intel CET

Intel CET [13] comes with two security measures. The first measure is a new state in the CPU. When taking an indirect branch, the CPU is expecting the next executed instruction to be a `endbr64` instruction, except if the `notrack` prefix is used on the branch. The second measure is a shadow stack that tracks `call` and `ret` instructions. The shadow stack is used against return-oriented attacks, by ensuring that the call chain in the real stack is coherent with the shadow stack. This is a major problem when doing function entry/exit instrumentation by changing the return value on the stack. To circumvent the first measure, it is possible to simply not create a patch that overlaps a `endbr64` instruction, reducing the overall instrumentation coverage by a small margin. The second measure, on the other hand, only impacts function entries/exits instrumentation and is more difficult to thwart. One option would be to communicate some intent – *e.g.* disabling the security measure for the instrumentation duration – to the operating system [47].

4.6 Experiments/Results

4.6.1 Reproductibility

Scientific reproducibility, over a long period, is difficult to achieve in the software world with all the security updates and the version dependencies. This is why we choose to use the purely functional package manager GNU Guix [50]. Indeed, Guix has a time machine functionality that allows going back in time to reproduce the software environment used in this paper. To do so, the script in Appendix 4.9.1 can be executed to reproduce all the artifacts of this paper. The execution with superuser permissions is required for the experiments containing micro-benchmarks with Uprobe. Indeed, they need to configure the tracing filesystem and, for memory-usage measurements, the cgroups configuration.

This only solves the software version side of the problem and does not compensate for the

increasing performance over time of computer systems, for metrics that involve timing. This is why we aim at generating the most deterministic environment, by applying some of the recommendations in this survey on tracers [4]. The specifications of the system environment, in which the experiments were executed, are detailed below.

Processor

- AMD Ryzen 7 5800X 8-Core Processor
- SMT disabled
- Frequency boosting disabled

Kernel

- Linux 6.1.10
- processor.max_cstate=0
- idle=poll

4.6.2 Coverage

In this experiment, we try to install an instrumentation point on each instruction of every function in a corpus of 23 binaries from 23 different packages. The set of functions of a binary is determined by its debug symbol table. Symbols emitted due to optimization, *e.g.* `.cold`, `.isra`, are filtered out, since they often alias other functions. The instrumentation is attempted one instruction at a time, for each binary, to avoid trampoline and patch regions collisions. The corpus and its results are shown in Table 4.1, where the instruction rate is the number of successfully instrumented instructions over the total number of instructions that were targeted. The function entry success rate represents the ratio of function entries successfully instrumented over the total number of functions that were tried. Table 4.2 shows the different averages and standard deviations of the coverage rates of the corpus. We can see that for each measure, the instruction success rate is in the 99% and the function entry success rate is in the 98%. Table 4.3 shows the proportions of each algorithm chosen for every instruction instrumented in the corpus, sorted in the order they were tried. We see that at least 41% of the instructions in a `x86-64` are at least 5 bytes long. The **nop** algorithm works at best in 14% of the cases. As for the **alias** and **nop-fit** algorithms, their sum is less than one percent, but they do generate valid solutions. Finally, even though the punning algorithm is the last algorithm tried, it counts for 43% of the successful instrumentations. In the end, less than 1% of the instructions were not successfully instrumented. Out of these, Table 4.4 shows the proportion of the top 20 instructions arrangements, where the numbers

in the regions represent the instructions size in bytes.

For example, we can see that around 28% and 11% of the regions are composed of respectively a single one byte instruction and two one byte instructions.

Table 4.1 Coverage results.

Binary	Instructions	Instruction success	Functions	Function entry success
bpftrace	365315	0.998	2247	0.997
libcurl.so.4.7.0	87506	0.992	938	0.985
libdbus-1.so.3.19.13	61465	0.953	1197	0.982
libavcodec.so.59.37.100	1943143	0.995	10443	0.999
git	544867	0.995	5220	0.997
libgstreamer-1.0.so.0.2003.0	175143	0.988	2656	0.986
httpd	75615	0.990	1168	0.996
libMagickCore-6.Q16.so.7.0.0	612181	0.997	2587	0.994
libpng16.so.16.37.0	33007	0.991	437	0.991
make	31532	0.994	336	0.973
js102	3510126	0.996	36571	0.964
mysql	117913	0.989	1648	0.900
nginx	185104	0.993	1708	0.987
libopencv_core.so.4.5.4	402771	0.997	4092	0.999
libperl.so	342194	0.995	2032	0.995
libPocoNet.so.81	104707	0.996	2491	0.994
libpulse.so.0.24.0	43106	0.993	495	0.998
libr_core.so.5.1.1	431062	0.980	3295	0.958
sqlite3	264427	0.994	2025	0.988
squid	890047	0.996	10403	0.980
tcpdump	144232	0.990	668	0.981
wireshark	1012742	0.996	10673	0.989
xz	7640	0.993	106	0.991

Table 4.2 Coverage averages and standard deviations.

Method	Instruction success	Function entry success
Arithmetic	0.991 ± 0.009	0.984 ± 0.021
Weighted	0.995 ± 0.005	0.979 ± 0.018
Geometric	$0.991 \stackrel{\times}{\div} 1.009$	$0.983 \stackrel{\times}{\div} 1.022$

Table 4.3 Algorithm chosen (sorted by attempt).

Algorithm	Amount	Proportion
fit	4677141	0.411
nop	1620649	0.142
nop-fit	19985	0.002
alias	25184	0.002
punning	4981770	0.438
none	61116	0.005
total	11385845	1.000

Table 4.4 Top 20 unpatchable regions.

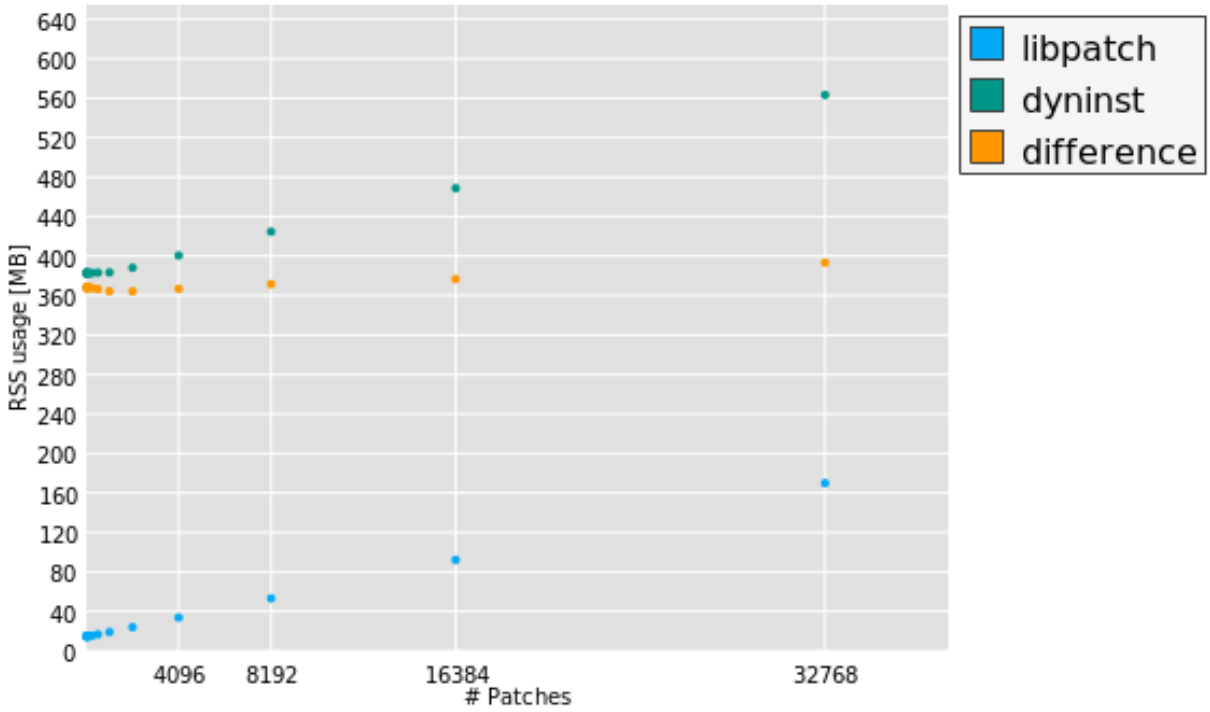
Region	Proportion
(1,)	0.282
(1, 1)	0.112
(2,)	0.098
(2, 2, 1)	0.072
(2, 1)	0.057
(1, 1, 2, 2)	0.042
(1, 3, 4)	0.036
(1, 3, 2)	0.034
(1, 3, 3)	0.033
(1, 1, 1)	0.033
(1, 3, 5)	0.029
(1, 3, 1)	0.020
(2, 1, 1)	0.017
(1, 2, 1)	0.016
(1, 3, 7)	0.015
(3, 1)	0.012
(1, 2, 1, 4)	0.009
(2, 2)	0.008
(1, 2, 1, 2)	0.007
(1, 3, 6)	0.007

4.6.3 Memory usage

In this experiment, we are interested in the memory usage of `Libpatch`. For this, we run two variants of a test program that does nothing. The program is full of dummy empty functions that can be instrumented. The first variant is linked against `Libpatch` which installs the patches. The second variant is not linked against `Libpatch`. Instead, another program is

linked against `Dyninst` and is used to install the patches in the test program. Since one of the variant uses more than one process, the memory usage is calculated by using the cgroup functionality of Linux. Figure 4.11 shows the RSS (Resident Set Size) for the `Dyninst` and `Libpatch` variants. It also shows the difference between the former and the latter. We can see that, for both variants, the memory usage is proportional to the number of patches, as expected. However, we see that the difference between the two is constant at the beginning and then increasing. The constant portion could be explained by the relative weight of the patches against the binary size and shows that `Libpatch` and its dependencies take less space in memory than `Dyninst` and its dependencies. On the other hand, the increasing portion of the difference shows that `Libpatch` patches take less memory space than `Dyninst` patches.

Figure 4.11 RSS usage against the number of patches.



4.6.4 Throughput

In this experiment, we are comparing the base case of a program against the `Libpatch` and `Dyninst` variants. The program runs on T threads over the total of online cores. Each thread increments its counter located in a shared memory. Every counter is aligned to avoid any false sharing. Meanwhile, an external program sums the counters at a sampling frequency

and computes the number of increments per nanosecond. For the variants of the program, P functions – that are never executed by the program – are patched and unpatched by an instrumenter at a frequency of F Hz. For the `Libpatch` variant, the instrumenter is inside the program address space while with `Dyninst` it is an external process that is doing the patching. Furthermore, the variable D denotes the minimal depth of the stack for the threads of the program.

Figure 4.12 shows the throughput – sampled at 10 Hz – of the program and its variants with $T = \frac{4}{8}$, $P = 4096$, $F = 1$ and $D = 128$. We can see that for the base case, the distribution of the throughput is 15.20 ± 0.00. For the `Libpatch` variant, the average is a little lower and the standard deviation is more spread. We also remark a thin line between 14 and 15 $increments/ns$ which probably corresponds to the impact of the cross-modifying protocol and the GC on the execution of the program. For the `Dyninst` variant, we see an average that has dropped by more than half. The standard deviation is also much wider; thus, the execution of the program is less deterministic. Finally, we can see the impact of the stop of the world approach with the drop to 0 $increment/ns$.

In Figure 4.13, the number of threads T used by the program is set to the number of online cores. We can see that the average throughput of the base case is twice as before, which makes sense since the number of threads has doubled. The impact of the variants on the throughput is now more pronounced, since the instrumenters are potentially taking a core that could be used by the program.

In Figure 4.14, the number of patches P is quadrupled. We can see that this does not affect the base case at all, as expected, and the `Libpatch` variant is not affected. However, for the `Dyninst` variant, while being more deterministic now, we can see a longer drop to 0 $increment/ns$.

In Figure 4.15, the stack depth D of each thread is multiplied by 8. We can see a little drop in the average and a more pronounced deviation of the `Libpatch` variant. This can be explained by the cross-modifying protocol and the garbage collector that require unwinding the stack of the threads in the program.

In Figure 4.16, the patching frequency F is unbounded. We can see that the `Libpatch` variant is unaffected by this, while the `Dyninst` variant is greatly affected.

Figure 4.12 Throughput sampling base scenario.

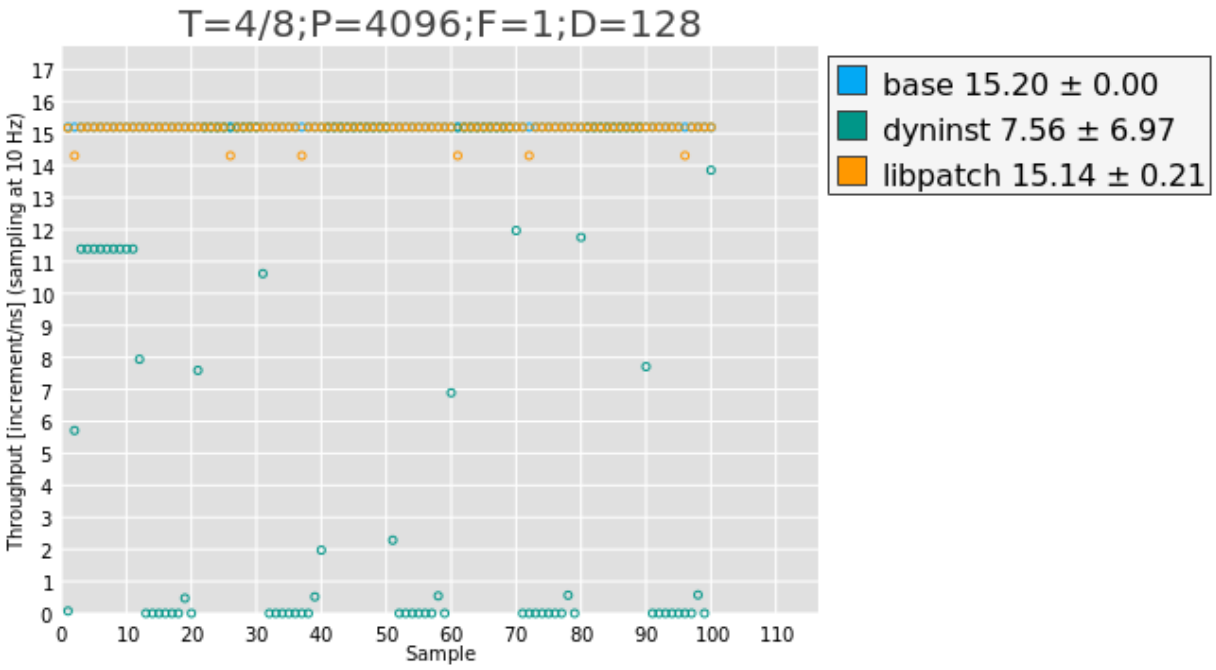


Figure 4.13 Throughput sampling with more threads.

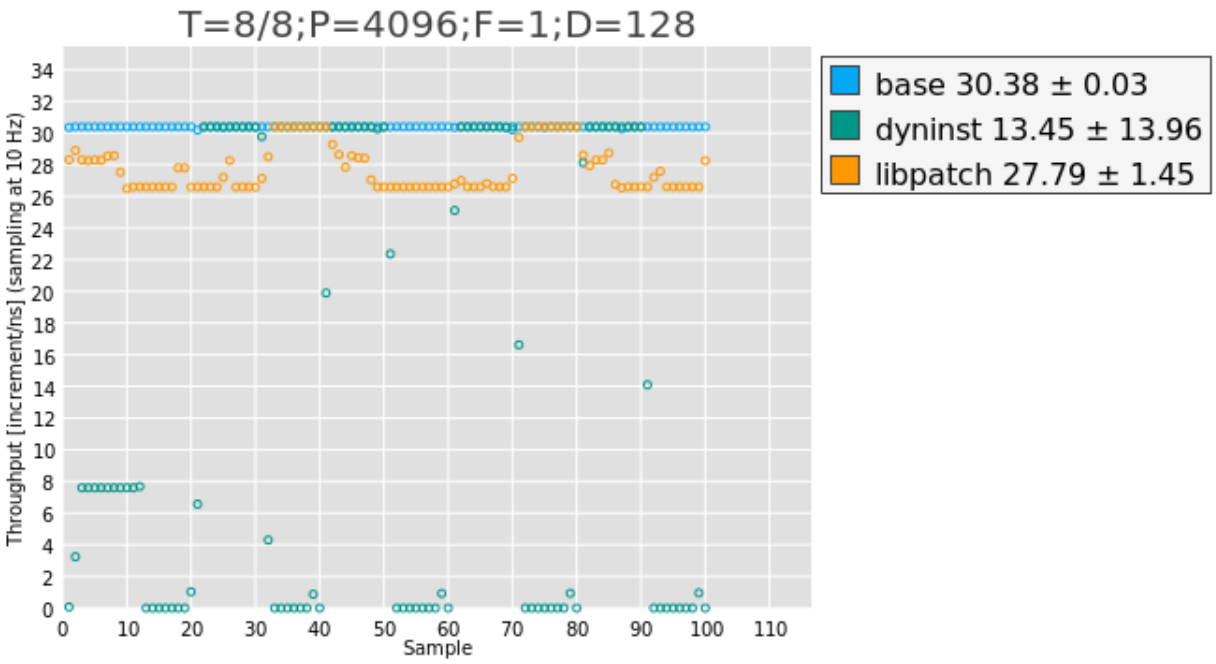


Figure 4.14 Throughput sampling with more patches.

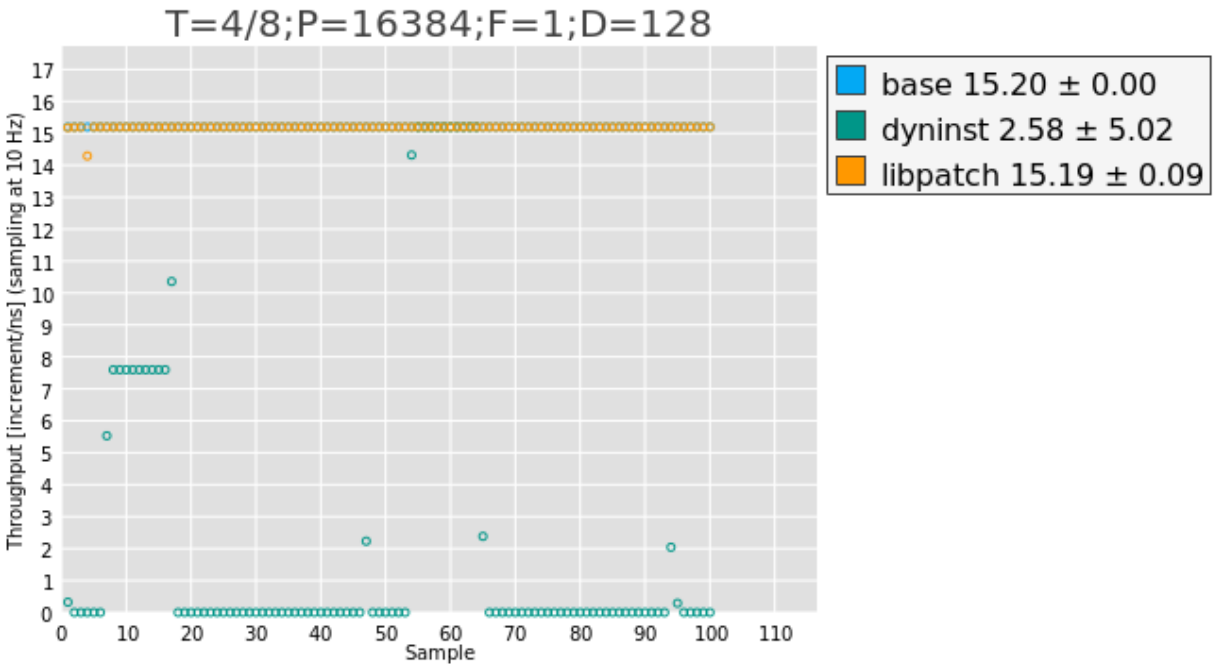


Figure 4.15 Throughput sampling with deeper stack.

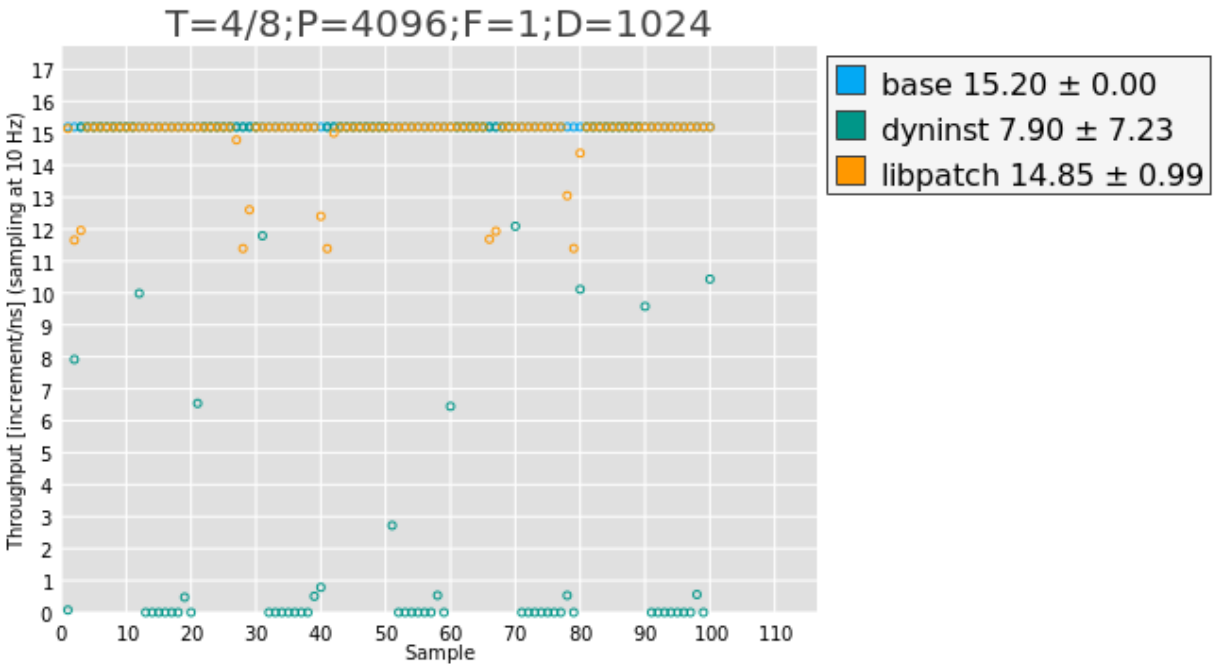
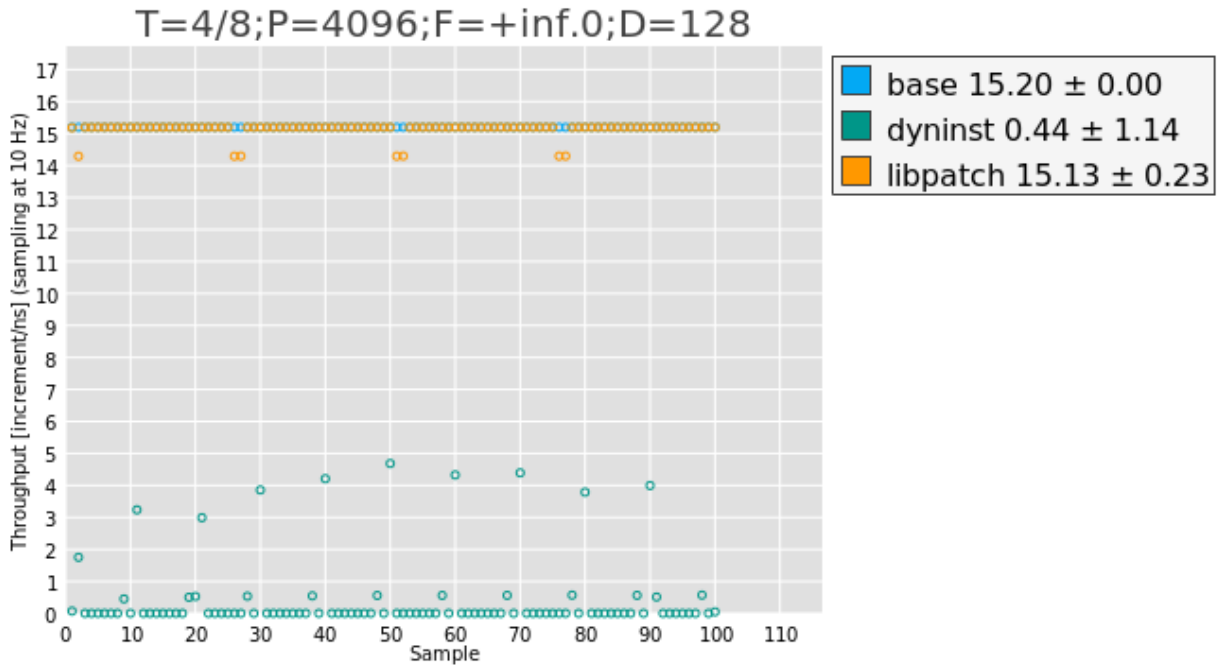


Figure 4.16 Throughput sampling with unbounded frequency.



4.6.5 Micro-benchmarks

In this experiment, a test program executes the function shown in Figure 4.17 in a tight loop 10^8 times with different arguments and in different threads in parallel. The disassembled function is also shown in Figure 4.18. There are seven variants that we compare against the base case. Note that for the GDB variant, we have used the fast tracepoints functionality and patch the head of the `gdb_collect` procedure with a `ret` to prevent any write to a trace buffer.

Four of these variants install a probe at the entry of the `powmod` function. These are shown in Figure 4.19 and Figure 4.20. We can see that the Uprobe and GDB variants scale very poorly. For the former, we assume that this is due to the software interrupt and context switching from userspace to kernel space. For the latter, we have determined that a spinlock is hold in the critical path of the JITed code emitted by GDB [51]. Although it seems that this scales well for less than six threads, the overhead grows linearly for each new thread added afterwards. This can be best explained by the ratio of time during the lock is held by a thread and the time in between the locks as demonstrated by [31]. Therefore, the scaling up to five threads is simply a coincidence that depends on the micro-architecture and the toolchain used for compiling. On the other hand, the `Libpatch` and `Dyninst` variants scale

perfectly by respectively adding a constant overhead of around 32 ns and 31 ns.

The other three variants install a function entry/exit probe on the `powmod` function. These are shown in Figure 4.21 and Figure 4.22. We can see that the Uprobe variant, again, does not scale well. Furthermore, the `Libpatch` and `Dyninst` variants scale again perfectly by respectively adding a constant overhead of around 58 ns and 50 ns.

Figure 4.17 C powmod function.

```c
static uint64_t powmod(uint64_t b, uint64_t e, uint64_t m)
{
        uint64_t c;

        c = 1;
        for(size_t i=0; i<e; ++i) {
                c = (c * b) % m;
        }

        return c;
}
```

Figure 4.18 Object dump of C powmod function.

```
0000000000401540 <powmod>:
  401540:       55                      push   %rbp
  401541:       48 89 e5                mov    %rsp,%rbp
  401544:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  401548:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
  40154c:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
  401550:       48 c7 45 f8 01 00 00    movq   $0x1,-0x8(%rbp)
  401557:       00
  401558:       48 c7 45 f0 00 00 00    movq   $0x0,-0x10(%rbp)
  40155f:       00
  401560:       eb 1b                   jmp    40157d <powmod+0x3d>
  401562:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  401566:       48 0f af 45 e8          imul   -0x18(%rbp),%rax
  40156b:       ba 00 00 00 00          mov    $0x0,%edx
  401570:       48 f7 75 d8             divq   -0x28(%rbp)
  401574:       48 89 55 f8             mov    %rdx,-0x8(%rbp)
  401578:       48 83 45 f0 01          addq   $0x1,-0x10(%rbp)
  40157d:       48 8b 45 f0             mov    -0x10(%rbp),%rax
  401581:       48 3b 45 e0             cmp    -0x20(%rbp),%rax
  401585:       72 db                   jb     401562 <powmod+0x22>
  401587:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  40158b:       5d                      pop    %rbp
  40158c:       c3                      ret
  40158d:       0f 1f 00                nopl   (%rax)
```

Figure 4.19 Average execution time of powmod (function entry).

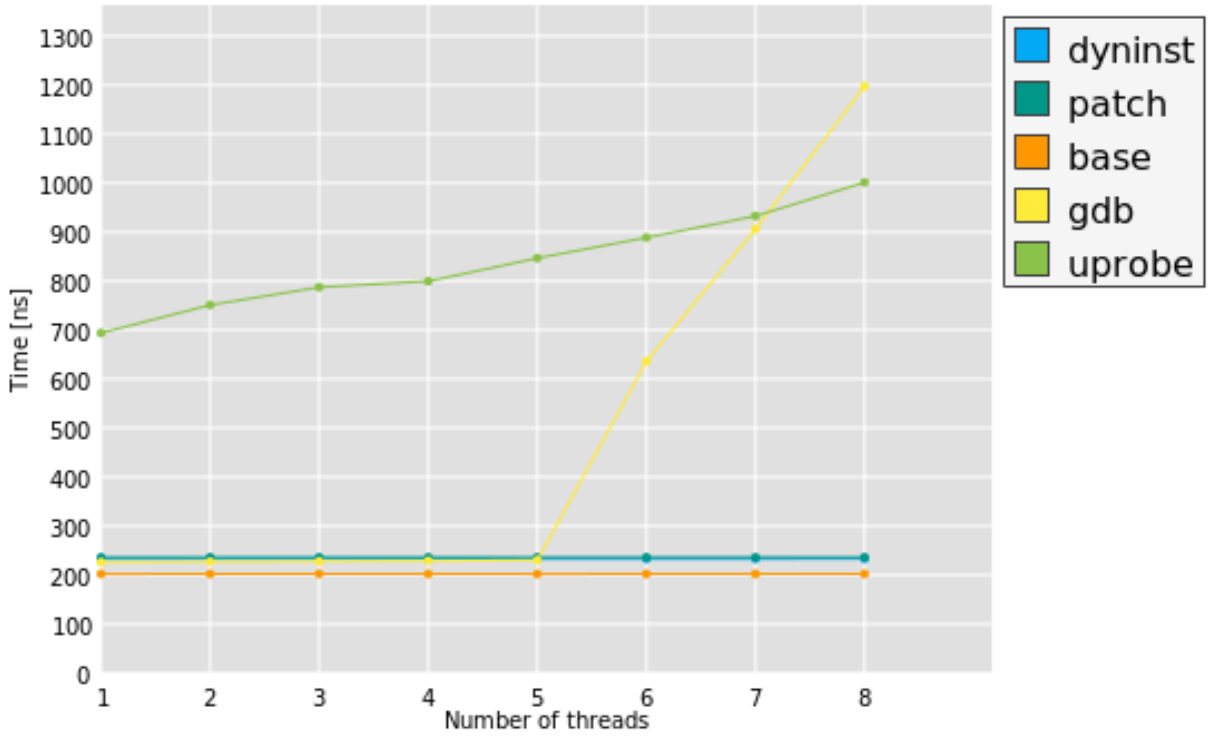


Figure 4.20 Average execution time of powmod (function entry without Uprobe and GDB).

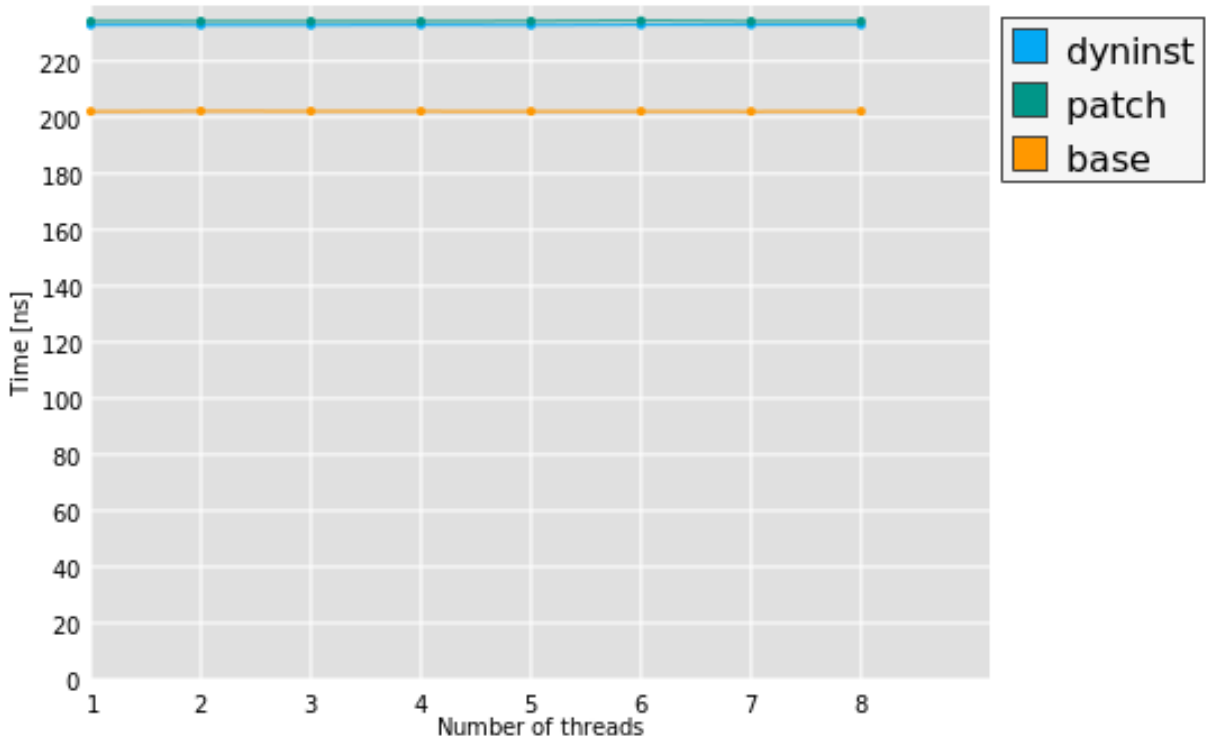


Figure 4.21 Average execution time of powmod (function entry/exit).

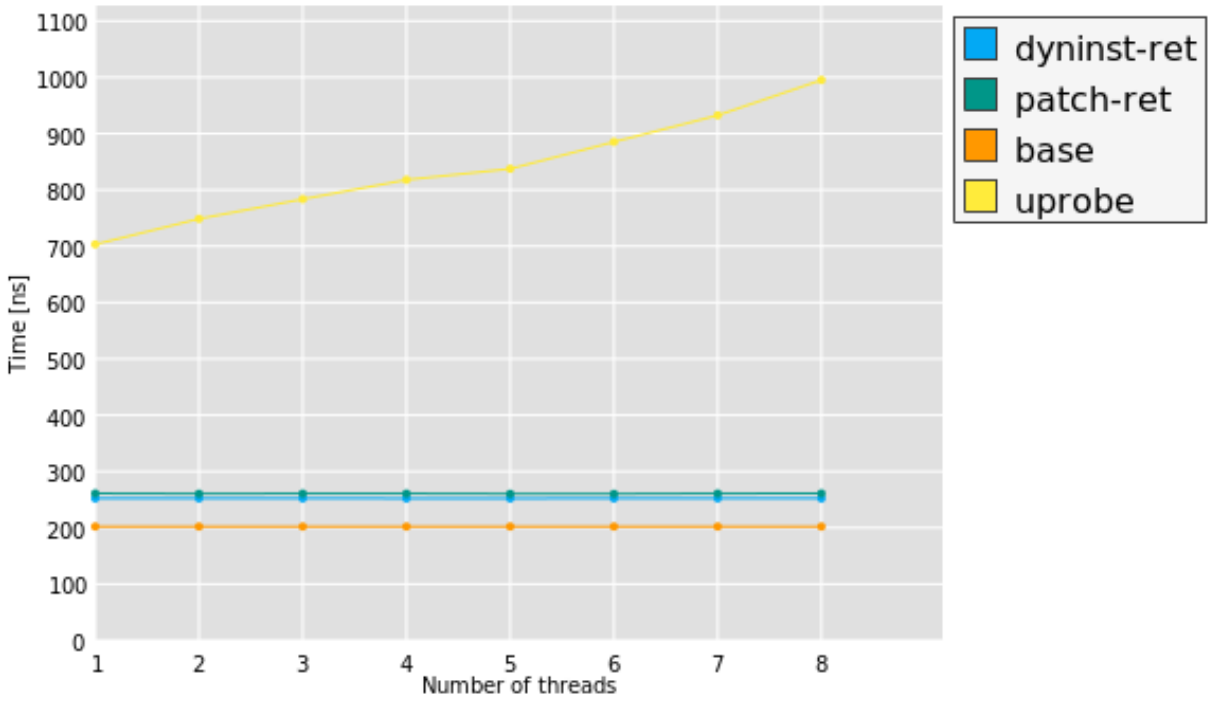
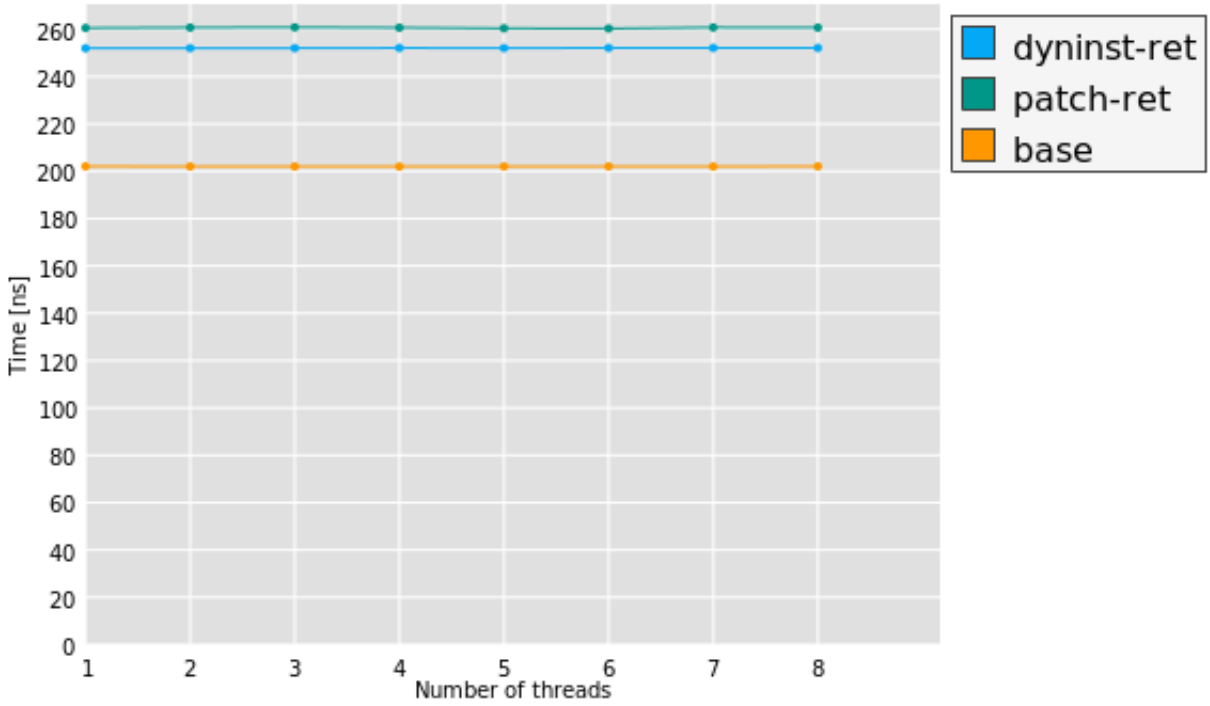


Figure 4.22 Average execution time of powmod (function entry/exit without Uprobe).



4.7 Conclusion

We have presented in this paper a new procedure for concurrent dynamic instrumentation. It improves upon and unifies the latest algorithms presented in recent years. Our implementation is called `Libpatch`. Experiments have shown that `Libpatch` can instrument on average 99.1% and 98.4% of the instructions and functions entries respectively. They also showed that the phyisical memory usage of `Libpatch` is linearly proportional to the number of patches, and that the act of patching a program at runtime does not impact in any meaningful way its computation. Finally, micro-benchmarks showed that the `Libpatch` diversion of the natural control of the program, introduces an overhead ranging from 32 to 58 nanoseconds, and scales perfectly with the number of threads in the program. Many tools like `LTTng`, `Uftrace`, `HPCToolkit` or `GDB` fast tracepoints are either severely limited in their coverage or use `Uprobe` or `Dyninst`. The former inserts probes quickly but their execution carry a large overhead, while the latter is very invasive for inserting probes, but their execution brings a small overhead. The new proposed methods implemented in `Libpatch` offer high coverage, quick insertion, and low execution overhead.

4.8 Acknowledgements

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for funding this project.

4.9 Appendix

4.9.1 Artefact

Figure 4.23 Shell script that reproduce the experiments.

```
#!/bin/sh
git clone https://git.sr.ht/~old/libpatch
cd libpatch || exit 1
git checkout b931ffb0452abdd907fd1c1a47459a340c60357d
cat -> channels.scm <<EOF
(list
 (channel
  (name 'guix)
  (url "https://git.savannah.gnu.org/git/guix.git")
  (branch "master")
  (commit
   "07f19ef04b5a8f4d7a12a8940333e67db8da81c0")
  (introduction
   (make-channel-introduction
    "9edb3f66fd807b096b48283debdcddccfea34bad"
    (openpgp-fingerprint
     "BBB0 2DDF 2CEA F6A8 0D1D  E643 A2A0 6DF2 A33A 54FA"))))
 (channel
  (name 'old)
  (url "https://git.sr.ht/~old/guix-channel")
  (branch "master")
  (commit
   "b6eb0ab773aac41b04819db840577cafc8107d09")
  (introduction
   (make-channel-introduction
    "fba5d96ea99ac4a7b3ab868eab0d68b3cc7285ae"
    (openpgp-fingerprint
     "295C 0246 4AC1 92F1 FFDD  7550 FCC0 88CE 07A0 4DAE")))))
EOF
guix time-machine --channels=channels.scm  \
     --                                    \
     shell --pure bash-minimal guix        \
     --                                    \
     ./dev-env bash - <<'EOF'
./configure
make -j "$(nproc)"
mkdir artifacts
echo "WARNING - Generation of some artifacts require root access."
echo "Generating memory usage benchmark (require root)"
./pre-inst-env run-memory-usage-benchmark
echo "Generating throughput benchmarks"
./pre-inst-env run-throughput-benchmark --threads="$(($(nproc)/2))" --patches=4096  --spamming=1      --depth=128
./pre-inst-env run-throughput-benchmark --threads="$(nproc)"        --patches=4096  --spamming=1      --depth=128
./pre-inst-env run-throughput-benchmark --threads="$(($(nproc)/2))" --patches=16384 --spamming=1      --depth=128
./pre-inst-env run-throughput-benchmark --threads="$(($(nproc)/2))" --patches=4096  --spamming=+inf.0 --depth=128
./pre-inst-env run-throughput-benchmark --threads="$(($(nproc)/2))" --patches=4096  --spamming=1      --depth=1024
echo "Don't mind the warning about ldconfig!"
echo "Dyninst uses it but it's not on Guix."
echo "Generating simple micro-benchmark with Uprobe & GDB (require root) - $(date)"
./pre-inst-env run-micro-benchmark --output=micro-benchmark-simple.png uprobe gdb base patch dyninst
echo "Generating simple micro-benchmark without Uprobe & GDB - $(date)"
./pre-inst-env run-micro-benchmark --output=micro-benchmark-simple-without-uprobe-gdb.png base patch dyninst
echo "Generating RET micro-benchmark with Uprobe (require root) - $(date)"
./pre-inst-env run-micro-benchmark --output=micro-benchmark-ret.png uprobe-ret base patch-ret dyninst-ret
echo "Generating RET micro-benchmark without Uprobe - $(date)"
./pre-inst-env run-micro-benchmark --output=micro-benchmark-ret-without-uprobe.png base patch-ret dyninst-ret
mv ./*.png ./artifacts
pushd workflow || exit 1
echo "Generating coverage (very long) - $(date)"
echo "Be sure to have a lttng-sessiond running as root"
WORKFLOW_CHANNELS=../channels.scm ./run-workflow coverage
mv coverage/*.txt ../artifacts
popd || exit 1
mv artifacts ../libpatch-artifacts
EOF
```

CHAPITRE 5 DISCUSSION GÉNÉRALE

Dans l'article présenté, il a été question de l'état de l'art et des algorithmes qui ont été unifiés, améliorés et complétés au sein de Libpatch. Il a été démontré que certains de ces algorithmes, quoique théoriquement intéressants, peuvent en pratique être confrontés à des difficultés lorsque implémentés sur de vrais systèmes. Par exemple, les mesures de sécurité du système, telles que WXE, et le respect de son ABI. Il a aussi été démontré que le couteau suisse en matière d'instrumentation dynamique, Dyninst, est très gourmand en termes de mémoire physique et que la modification du programme pour augmenter l'instrumentation engendre des pics de surcoûts temporels extrêmes, pouvant affecter le bon fonctionnement du programme. En contrepartie, les données présentées montrent que Libpatch est en mesure d'implémenter les algorithmes de la littérature – et en apporte de nouveaux – tout en sachant s'adapter aux paramètres de différents systèmes. De plus, des comparaisons ont été faites avec Dyninst pour démontrer que Libpatch est comparable en termes de surcoût temporel durant l'exécution de l'instrumentation, mais sans les surcoûts associés à Dyninst. Finalement, il a été montré que Libpatch est en mesure d'instrumenter un peu partout dans un binaire à quelques exceptions près. En somme, il est juste de conclure que Libpatch répond aux objectifs de la recherche. Il est toutefois intéressant de discuter de certains aspects présentés, mais peu expliqués en profondeur dans l'article.

Premièrement, il est intéressant de considérer pour quelles raisons Libpatch performe mieux que Dyninst, sur le plan de la modification du programme pendant son exécution, mais aussi dans quels cas l'écart entre les deux pourrait diminuer. Il faut comprendre que Dyninst privilégie une approche dite de l'arrêt du monde. Donc, tous les fils d'exécution sont arrêtés. Cela est plus complexe que de faire un simple `SIGSTOP` et attendre après le processus, avec l'appele système `waitpid(2)`. En effet, bien que le signal `SIGSTOP` s'applique sur l'ensemble des fils d'exécution du programme, leur arrêt respectif n'est pas synchronisé. Il faut donc attendre sur chaque fil de façon individuelle. Cela a déjà comme première conséquence d'allonger le temps avant que Dyninst ne puisse commencer à modifier le programme. La deuxième conséquence est que le travail effectué par le programme est nul durant la période de modification. En contrepartie, Libpatch a un protocole – sur `x86-64` – qui arrête les fils d'exécution de façon incrémentale. Il n'y a donc aucune latence initiale et, tout au plus, un fil d'exécution ne travaille pas. Cependant, la période durant laquelle un fil d'exécution ne travaille pas est linéairement proportionnelle à la profondeur de sa pile. Cela s'explique par le fait que Libpatch doit dérouler la pile du fil pour savoir s'il se trouve dans un signal système asynchrone. C'est pour cette raison que la variable D a été testée dans le banc d'essai de débit.

D'ailleurs, aucune métrique sur la profondeur moyenne de la pile d'exécution n'a été trouvée dans la littérature. Il a donc été choisi d'utiliser une valeur de 128, qui est vraisemblablement beaucoup plus élevée que la normale, afin de soumettre l'algorithme de Libpatch à un cas particulièrement difficile, et ainsi ne pas l'avantager de manière artificielle. En somme, de par son approche, Dyninst condense l'interruption totale du travail fait par le logiciel sur une période. En contrepartie, Libpatch favorise une approche où la période est plus longue, mais où le travail est marginalement diminué. Cela fonctionne bien dans le cas où il n'y a pas de synchronisation entre les fils d'exécution, comme dans le banc d'essai. Or, on peut imaginer un scénario où l'arrêt d'un fil entraîne l'arrêt d'autres fils – par exemple à cause d'une condition d'exclusion mutuelle – dans le programme. Ainsi, il est possible que, dans certains cas, l'approche de Libpatch tende à offrir des performances semblables à celles de Dyninst, voire pires.

Deuxièmement, il a été question de relocalisation des instructions dans un tampon OLX. Il existe cependant une autre technique – peu explorée par manque de temps – qui a le mérite d'être détaillée. Au lieu de relocaliser uniquement les instructions, on pourrait aussi relocaliser les données. L'idée est de copier les instructions originales dans un tampon et d'utiliser les techniques présentées pour les systèmes WXE dans le but d'avoir des pages virtuelles proches du tampon qui pointe vers les pages physiques des données. Il faut donc uniquement recalculer le déplacement des instructions par rapport aux nouvelles adresses virtuelles des données. Cela présente l'avantage d'être assez générique pour toutes les instructions qui n'effectuent pas un branchement. Cette technique a aussi l'avantage d'être indépendante de l'ABI du système – par exemple on élimine le besoin de préserver la zone rouge – ce qui la rend plus portable.

Finalement, une approche conservatrice est utilisée lorsqu'un branchement indirect est présent dans une fonction. Pour rappel, dans ce cas Libpatch considère que toutes les instructions dans la fonction sont potentiellement des entrées de bloc de base. Or, il serait possible d'être moins conservateur si plus d'information sur le flot de contrôle du programme était disponible. Par exemple, ce serait le cas avec l'aide de l'attribut `basic_bloc` du programme, `.debug_line` dans le standard `DWARF`. Cependant, est-ce que cela est vraiment utile ? Du point de vue de la couverture de l'instrumentation, il faut se demander si le maximum 0.9% à gagner en vaut la peine, sachant qu'aucun compilateur ne supporte l'attribut `basic_bloc` et que l'analyse approfondie du CFG – comme le fait Dyninst – implique des coûts non négligeables en mémoire et temps. Du point de vue de la performance de l'instrumentation, cela ne change strictement rien. Par exemple, si l'instruction n'est pas un bloc de base, mais que de façon conservatrice une instruction illégale est utilisée pour l'algorithme de calembourage. Alors, par définition, on sait qu'aucun saut ne sera fait sur cette instruction. Il n'y aura donc jamais d'impact sur la performance.

CHAPITRE 6 CONCLUSION

En conclusion de ce mémoire, une synthèse des travaux est présentée ainsi que les limitations actuelles de Libpatch et les extensions possibles pour le futur.

6.1 Synthèse des travaux

La contribution principale de cette recherche a été d'unifier, étendre et compléter les différents algorithmes présents dans la littérature, et de valider le tout sous la forme d'une bibliothèque en espace utilisateur. Or, plusieurs algorithmes ont dû être révisés et d'autres ont été inventés par l'auteur, pour pallier les faiblesses de ce qu'on retrouve dans l'état de l'art.

Instruction calembourée L'algorithme original utilise uniquement 14 codes d'instructions d'un octet. Or, il en existe 22 documentés dans les manuels des vendeurs.

Tampon OLX Les algorithmes originaux d'émulation d'instructions relocalisées ne préservent pas tous la zone rouge telle que décrite dans l'ABI System V. Les nouveaux algorithmes proposés le font.

Recherche de trampoline L'ensemble des algorithmes liés à la recherche d'un trampoline sont nouveaux et ont été inventés pour cette recherche.

Réclamation de la mémoire Le récupérateur de mémoire est une invention hautement optimisée pour cette recherche. Bien que les systèmes existants comme Dyninst proposent des méthodes pour réclamer la mémoire, ceux-ci ne sont pas aussi performants.

Flot de contrôle du correctif Quoique similaire à plusieurs recherches, de par le branchement vers un trampoline, le flot de contrôle d'un correctif se distingue par l'abscence d'un compteur de référence et de mécanismes de synchronisation. Il offre ainsi les meilleures performances possibles. En somme, cette contribution s'incrit bien dans la démarche proposée ici qui vise à s'assurer d'une méthode d'instrumentation efficace de bout en bout.

6.2 Limitations des solutions proposées et améliorations futures

Malgré les résultats impressionnants, Libpatch n'est pas parfait et beaucoup reste à faire pour le futur. Premièrement, il serait important de généraliser l'algorithme de recherche des trampolines. En effet, en ce moment, l'algorithme suppose que les pages en mémoire font 4096 octets et que les lignes de la cache de premier niveau des instructions font 64 octets. Or, cela n'est pas du tout portable. Deuxièmement, les gestionnaires de correctifs ne sont pas complets, puisqu'ils ne préservent pas les registres des extensions SIMD (Single Instruction Multiple Data) dits XMM. De plus, il y aurait un gain en performance en générant les portions de sauvegarde et restauration des registres de façon dynamique. En effet, il est possible d'inclure dans la spécification d'un point d'instrumentation les accès en lecture et écriture des différents registres du programme. Ainsi, le correctif dynamiquement généré pourrait faire le minimum d'accès en mémoire dans ces portions. Troisièmement, il est possible d'effectuer une analyse des registres dans les fonctions instrumentées, pour optimiser l'émulation des instructions relocalisées dans le tampon OLX. De plus, certains tampons OLX sont identiques et il serait avantageux de les voir partager la même localisation en mémoire, pour favoriser la cache des instructions. Cela demanderait l'utilisation d'un compteur de références sur les tampons, et de les marquer prêt à être réclamé uniquement lorsque le compteur tombe à zéro. Quatrièmement, porter Lipatch à une architecture telle que `ARM` permettrait de vérifier si les résultats obtenus sont similaires et si les algorithmes indépendants de l'architecture s'appliquent aussi bien que sur `x86-64`. Finalement, l'intégration de Libpatch au sein d'un outil d'instrumentation populaire, tel que LTTng-ust, GDB ou uftrace, serait une preuve de concept forte, et pourrait faire ressortir le cas échéant les limitations de son API.

RÉFÉRENCES

[1] *Intel ® Architecture Instruction Set Extensions and Future Features*, Intel Corporation, 2022, disponible : https://cdrdv2-public.intel.com/671368/architecture-instruction-set-extensions-programming-reference.pdf [En ligne : 12 Février 2023].

[2] S. Dronamraju, *Uprobe-tracer : Uprobe-based Event Tracing*, 2022, disponible : https://www.kernel.org/doc/html/v6.0/trace/uprobetracer.html [En ligne : 24 Novembre 2022].

[3] J. Corbet, *DebugFS*, 2022, disponible : https://www.kernel.org/doc/html/v6.0/filesystems/debugfs.html [En ligne : 24 Novembre 2022].

[4] 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, mar 2018. [En ligne]. Disponible : https://doi.org/10.1145/3158644

[5] 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

[6] B. Buck et J. K. Hollingsworth, "An api for runtime code patching," *The International Journal of High Performance Computing Applications*, vol. 14, nº. 4, p. 317–329, 2000. [En ligne]. Disponible : https://doi.org/10.1177/109434200001400404

[7] M. Kerrisk, *ptrace(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/ptrace.2.html [En ligne : 25 Novembre 2022].

[8] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A : System Programming Guide, Part 1*, Intel Corporation, 2022, disponible : https://cdrdv2-public.intel.com/671190/253668-sdm-vol-3a.pdf [En ligne : 7 Novembre 2022].

[9] *AMD64 Architecture Programmer's Manual Volume 2 : System Programming*, Advanced Micro Devices Inc., 2022, disponible : https://www.amd.com/system/files/TechDocs/24593.pdf [En ligne : 8 Novembre 2022].

[10] *Architecture Reference Manual for A-profile architecture*, Arm Limited, 2022, disponible : https://documentation-service.arm.com/static/62ff43b0e95b0a633aff8a64 [En ligne : 8 Novembre 2022].

[11] M. Michael, J. Hubicka, A. Jaeger et M. Mitchell, *System V Application Binary Interface AMD64 Architecture Processor Supplement*, 2012, disponible : https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf [En ligne : 5 Décembre 2022].

[12] M. Kerrisk, *mprotect(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/mprotect.2.html [En ligne : 25 Novembre 2022].

[13] V. Shanbhogue, D. Gupta et R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," dans *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA : Association for Computing Machinery, 2019. [En ligne]. Disponible : https://doi.org/10.1145/3337167.3337175

[14] J. Edge, *Control-flow integrity for the kernel*, 2020, disponible : https://lwn.net/Articles/810077/ [En ligne : 11 Février 2023].

[15] M. Hiramatsu, P. S Panchamukhi et J. Keniston, *Kernel Probes (Kprobesaadas)*, 2022, disponible : https://www.kernel.org/doc/html/v6.0/trace/kprobes.html [En ligne : 8 Novembre 2022].

[16] S. Goswami, *An introduction to KProbes*, 2005, disponible : https://lwn.net/Articles/132196/ [En ligne : 11 Février 2023].

[17] M. Hiramatsu, *The Enhancement of Kernel Probing - Kprobes Jump Optimization*, 2010, disponible : https://pre.tracingsummit.org/ts/2010/files/HiramatsuLinuxCon2010.pdf [En ligne : 11 Février 2023].

[18] N. Kim, "Uftrace," 2022, disponible : https://github.com/namhyung/uftrace [En ligne : 25 Novembre 2022].

[19] ——, "Uftrace updates," 2029, disponible : https://tracingsummit.org/ts/2019/files/Tracingsummit2019-uftrace-kim.pdf [En ligne : 11 Février 2023].

[20] S. Rostedt, *ftrace - Function Tracer*, 2022, disponible : https://www.kernel.org/doc/html/v6.0/trace/ftrace.html [En ligne : 28 Novembre 2022].

[21] ——, "Debugging the kernel using ftrace - part 1," 2007, disponible : https://lwn.net/Articles/365835/ [En ligne : 11 Février 2023].

[22] M. Kerrisk, *gcc(1) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man1/gcc.1.html [En ligne : 28 Novembre 2022].

[23] I. Free Software Foundation, *Using the GNU Compiler Collection (GCC)*, 2022, disponible : https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/index.html [En ligne : 28 Novembre 2022].

[24] T. L. C. I. Project, *XRay Instrumentation*, 2016, disponible : https://www.llvm.org/docs/XRay.html [En ligne : 28 Novembre 2022].

[25] 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.

[26] T. Commitee, *Executable And Linking Format (ELF) Specification*, 1995, disponible : https://refspecs.linuxbase.org/elf/elf.pdf [En ligne : 28 Novembre 2022].

[27] 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

[28] M. Kerrisk, *The Linux Programming Interface : A Linux and UNIX System Programming Handbook*, 1er éd. USA : No Starch Press, 2010.

[29] ——, *mmap(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/mmap.2.html [En ligne : 25 Novembre 2022].

[30] ——, *sigaction(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/sigaction.2.html [En ligne : 25 Novembre 2022].

[31] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais et J. Walpole, "User-level implementations of read-copy update," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, n°. 2, p. 375–382, 2012.

[32] P. McKenney, "What is rcu, fundamentally ?" 2007, disponible : https://lwn.net/Articles/262464/ [En ligne : 11 Février 2023].

[33] P. Mckenney, "Is parallel programming hard, and, if so, what can you do about it ?" 01 2012.

[34] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D : Instruction Set Reference, W-Z*, Intel Corporation, 2022, disponible : https://cdrdv2-public.intel.com/671143/334569-sdm-vol-2d.pdf [En ligne : 9 Novembre 2022].

[35] *AMD64 Architecture Programmer's Manual Volume 3 : General-Purpose and System Instructions*, Advanced Micro Devices Inc., 2022, disponible : https://www.amd.com/system/files/TechDocs/24594.pdf [En ligne : 9 Novembre 2022].

[36] 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, jun 2017. [En ligne]. Disponible : https://doi.org/10.1145/3140587.3062344

[37] iu parfunc, "Liteinst," 2018, disponible : https://github.com/iu-parfunc/liteinst [En ligne : 5 Décembre 2022].

[38] 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

[39] G. J. Duck, X. Gao et A. Roychoudhury, “Binary rewriting without control flow recovery,” dans *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA : Association for Computing Machinery, 2020, p. 151–163. [En ligne]. Disponible : https://doi.org/10.1145/3385412.3385972

[40] A. Balboul, “Points de trace rapides et efficaces par injection adaptative de sauts en x86,” Mémoire de maîtrise, Polytechnique Montréal, mai 2020. [En ligne]. Disponible : https://publications.polymtl.ca/5271/

[41] M. Kerrisk, *sigprocmask(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/sigprocmask.2.html [En ligne : 25 Novembre 2022].

[42] D. D. I. F. Committee, *DWARF Debugging Information Format Version 5*, 2017, disponible : https://dwarfstd.org/doc/DWARF5.pdf [En ligne : 26 Novembre 2022].

[43] G.-A. Pollo-Guilbert, “Déduction des cibles de sauts indirects pour les applications de traçage x86,” Mémoire de maîtrise, Polytechnique Montréal, décembre 2021. [En ligne]. Disponible : https://publications.polymtl.ca/9990/

[44] X. Meng et W. Liu, “Incremental cfg patching for binary rewriting,” dans *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. New York, NY, USA : Association for Computing Machinery, 2021, p. 1020–1033. [En ligne]. Disponible : https://doi.org/10.1145/3445814.3446765

[45] E. Bauman, Z. Lin et K. W. Hamlen, “Superset disassembly : Statically rewriting x86 binaries without heuristics,” dans *Network and Distributed System Security Symposium*, 2018.

[46] M. Kerrisk, *memfd_create(2) — Linux manual page*, 2021, disponible : https://man7.org/linux/man-pages/man2/memfd_create.2.html [En ligne : 28 Novembre 2022].

[47] J. Corber, “Shadow stacks for user space,” 2022, disponible : https://lwn.net/Articles/885220/ [En ligne : 28 Novembre 2022].

[48] O. Dion, “Libpatch,” disponible : https://archive.softwareheritage.org/browse/origin/https://git.sr.ht/~old/libpatch.

[49] *Differences between v1 and v2 of the ABI for the ARM® Architecture*, Arm Limited, 2005, disponible : https://documentation-service.arm.com/static/5eff24a0cafe527e86f5b243 [En ligne : 24 Janvier 2023].

[50] "Gnu guix," 2023, disponible : https://guix.gnu.org/ [En ligne : 12 Janvier 2023].

[51] "Gdb," disponible : https://archive.softwareheritage.org/browse/content/sha1_git:eeaced8c8d521c5e9c2880a9fcdecac4ef73fec7/?origin_url=git://sourceware.org/git/binutils-gdb.git&path=gdbserver/linux-x86-low.cc#L1214.