Titre: Title:	Vérification dynamique ciblée et interactive de programmes grâce à une architecture modulaire
Auteur: Author:	Paul Naert
Date:	2020
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Naert, P. (2020). Vérification dynamique ciblée et interactive de programmes grâce à une architecture modulaire [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/5280/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/5280/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Vérification dynamique ciblée et interactive de programmes grâce à une architecture modulaire

PAUL NAERT
Département de génie informatique et génie logiciel

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

Mai 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Vérification dynamique ciblée et interactive de programmes grâce à une architecture modulaire

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

François GUIBAULT, président
Michel DAGENAIS, membre et directeur de recherche
Benjamin DE LEENER, membre

REMERCIEMENTS

Je souhaiterais tout d'abord remercier mon directeur de recherche Michel Dagenais pour m'avoir encadré tout au long de ce projet. Son recul sur les différents sujets ainsi que sa disponibilité m'ont permis de toujours obtenir des réponses rapides et éclairantes à toutes mes questions. Merci de m'avoir accueilli dans le laboratoire.

Je tiens également à remercier Geneviève Bastien et tous mes collègues du DORSAL pour leur bonne humeur et leur aide spontanée dès que j'ai rencontré un problème, même dans la période compliquée de confinement qui a marqué la fin de mon projet.

Enfin, je souhaite remercier pour leur soutien financier le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNGC) ainsi que les partenaires industriel du DORSAL : Ciena, Ericsson, EfficiOS, Google et Prompt.

RÉSUMÉ

Le cycle de développement d'une application contient plusieurs phases, de l'écriture au soutien technique suivant la publication. Une phase particulièrement importante est la vérification du programme. Il s'agit de vérifier que le programme produit répond à la spécification au sens large, c'est à dire qu'il présente le comportement prévu sans bogue, quel que soit le scénario et les entrées présentées.

De nombreux outils sont disponibles pour assister l'utilisateur dans cette tâche. Parmi ceux-ci, on trouve les outils de vérification formelle qui permettent de modéliser le déroulement d'un programme et d'en prouver mathématiquement la validité. Les analyses statiques peinent cependant à vérifier des programmes complexes, et une autre famille d'outils est souvent nécessaire. Ce sont les outils dynamiques, qui vérifient l'intégrité du programme pendant son exécution. Dans ce domaine, on trouve surtout des outils spécialisés et efficaces, mais peu flexibles. En effet, en l'absence d'une structure commune, beaucoup d'outils réécrivent intégralement toutes les fonctionnalités de base, et ce coût de développement fait qu'ils se limitent souvent aux fonctionnalités strictement nécessaires. Peu d'outils proposent ainsi une instrumentation dynamique ou une interface graphique.

Dans le cadre de ce projet de recherche, nous proposons une solution à ce problème en prenant exemple sur la philosophie des éditeurs de code modulaires. Il s'agit de partager au maximum les fonctionnalités simples pour éviter de multiplier les réécritures inutiles et ainsi permettre un développement plus avancé des outils de vérification dynamique. Notre objectif est donc de créer une plateforme permettant la mise en commun simple des fonctionnalités de base de la vérification, afin de permettre la création de nouveaux outils et l'amélioration des outils existants.

Nous avons pour cela créé une nouvelle architecture couvrant toute l'infrastructure de vérification, de l'utilisateur au programme débogué. Cette architecture est à deux niveaux : le niveau source et le niveau binaire. Au cœur du niveau source se trouve un éditeur de texte de nouvelle génération pour lequel nous avons conçu un module spécialisé. Ce module permet de charger les outils basés sur notre infrastructure, et adapte son interface en conséquence. En intégrant directement le code source à son interface, il permet un ciblage avancé que ne proposent pas les outils existants.

Le domaine binaire est contrôlé par un débogueur, un outil versatile et interactif permettant la manipulation des fichiers binaires. Ce logiciel permet d'offrir des fonctionnalités partagées aux outils utilisant notre architecture, notamment la possibilité de s'attacher à un programme

en cours d'exécution et d'y charger des librairies. Afin de compléter cette offre, nous avons conçu et intégré au débogueur GNU un mécanisme d'instrumentation dynamique de haute performance utilisant un système de trampolines pour injecter de nouvelles instructions directement dans le programme.

A partir de cette plateforme interactive de vérification dynamique, nous avons pu améliorer plusieurs outils de vérification existants en offrant des versions plus flexibles, ciblées et performantes.

ABSTRACT

The development cycle of an application covers multiple different stages, from code writing to technical support. One crucial phase is program verification and debugging. During this stage, the developers need to make sure that the program they deliver corresponds to both its explicit and implicit specification, meaning that it has to behave correctly and without any bug whatever input is given to it.

Multiple tools exist to assist the developers. Among them, formal verification is a method which proves mathematically the validity of a program by modeling its behavior. However, this type of static analysis struggle to analyse properly complex programs, and developers often also rely on dynamic tools, which check the integrity of a running program. A large number of specialized tools exist in that domain, but they often go for a lean approach, with little flexibility and adaptability. This is partly due to the lack of a common framework for high performance runtime verification tools. Most tools have to reprogram every functionality from the ground up, which means they often limit their scope to what is strictly necessary to reduce development costs. Features such as dynamic instrumentation or even a graphical user interface are seldom available.

As part of this research project, we propose a solution to this problem, taking example on the recent development in integrated development environments. The goal is to provide modularity in order to share underlying features as much as possible. This removes the need for rewriting those basic features and enables developers to focus on more advanced tasks, which in turn produces better verification tools.

For this purpose, we designed a new multi-layered architecture which covers all the debugging infrastructure, from user interaction to program execution. This architecture is divided into two domains: source and binary. The source domain revolves around a modular development environment, for which we developed a special module. This module enables loading and executing different tools by adapting its interface according to the current tool. By integrating directly to the source code, it enables precise targeting of instrumentation, a feature absent from most tools.

At the center of the binary domain we put a debugger, as this versatile and interactive tool already contains the infrastructure for binary manipulation. For instance, it makes it easy for verification tools to attach to running programs and load libraries. One crucial feature of verification tools that the debuggers generally lack is efficient and versatile instrumentation. To solve this issue, we designed a new instrumentation platform within the GNU debugger,

enabling code injection within a program with minimal runtime overhead.

Using this interactive runtime verification framework, we were able to improve several state-of-the-art tools by creating more flexible, targeted and efficient versions.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

VE	Vérification à l'Exécution
GNU	GNU's Not Unix
GDB	GNU Debugger
VSCode	Visual Studio Code
LTTng	Linux Tracing Toolkit Next Generation
ASan	Address Sanitizer

CHAPITRE 1 INTRODUCTION

La vérification de programmes est une phase de plus en plus importante du développement de logiciel. En effet, les applications sont de plus en plus complexes, reposant sur une variété d'outils et de librairies qui communiquent sur des interfaces parfois non fiables. Le débogage d'un programme avant sa publication, mais également son soutien technique nécessitent de pouvoir observer et diagnostiquer le comportement de programmes pendant leur exécution. Parmi les problèmes les plus difficiles à détecter, on retrouve les bogues de mémoire comme la fuite ou la corruption, mais également des problèmes liés à la concurrence de fils d'exécutions.

Pour détecter et résoudre ces problèmes, il existe une grande variété de programmes spécialisés. Parmi ceux-ci, on retrouve des programmes flexibles et exhaustifs comme les outils de la suite Valgrind. Cependant, ces outils sont lourds et entraînent un coût en performance très élevé. A l'inverse, on retrouve des outils performants mais peu flexibles et parfois difficiles à prendre en main. Il n'existe pas d'outil de vérification dynamique offrant à la fois performance et flexibilité.

1.1 Définitions et concepts de base

Vérification de programme : La vérification d'un programme consiste à s'assurer qu'un programme suit la spécification, explicite ou implicite, qu'on souhaite lui imposer. Il s'agit donc de vérifier qu'un programme répond correctement aux entrées qui peuvent lui être proposées, mais également qu'il ne produit pas d'effet de bord indésirables comme des fuites de mémoire. La vérification d'un programme peut avoir lieu avant la compilation, on parle alors de vérification statique, ou sur le fichier binaire, auquel cas elle est qualifiée de dynamique ou vérification à l'exécution. Pour la vérification statique, l'utilisation de modélisation mathématique est souvent utilisée pour la vérification de logiciel critique, on parle alors de vérification formelle. La vérification à l'exécution se décompose en trois parties : l'instrumentation du programme, la collecte de données et l'analyse de ces données. Cette analyse peut se faire pendant l'exécution ou a posteriori, selon les ressources nécessaires et disponibles.

Instrumentation : L'instrumentation d'un programme consiste en la modification de ce dernier afin d'en extraire des données ou de modifier son comportement. L'instrumentation est dite statique quand elle est effectuée au niveau du code source ou de la compilation, par exemple par l'insertion de directives permettant d'imprimer à l'écran certaines variables internes. L'instrumentation du binaire compilé est appelée instrumentation dynamique, et

sert notamment aux débogueurs pour injecter des points d'arrêts dans le programme. L'instrumentation dynamique requiert une modification des instructions binaires d'un programme une fois qu'elles sont chargées en mémoire.

Débogage : Au sens large, le débogage est la recherche et la correction de bogues dans un programme, ce domaine intersecte donc celui la vérification. Le sens plus courant de débogage est l'analyse du comportement d'un programme à l'exécution via l'utilisation d'un logiciel de débogage interactif, ou débogueur. Ce type de logiciel permet à l'utilisateur de contrôler le flux d'exécution d'un programme en arrêtant et reprenant son exécution via l'utilisation de points d'arrêt. Ces derniers sont des modifications du code en mémoire qui interrompent l'exécution du programme en prévenant le système d'exploitation. Celui-ci rend le contrôle à l'utilisateur qui peut inspecter l'état du programme, notamment les valeurs des variables et registres.

Traçage Le traçage consiste en la collecte d'informations sur l'état interne d'un programme, souvent accompagnées de données temporelles. L'analyse de ces données de trace permet de retracer le flux d'exécution d'un programme, mais également les sections plus ou moins rapides. Le traçage est donc un outil primordial de l'analyse de performance. Il est donc crucial pour un outil de traçage d'avoir un coût d'exécution minimal afin d'observer le comportement naturel du programme. Le traçage peut avoir lieu au niveau des applications de l'espace usager, mais également au niveau du noyau du système d'exploitation, auquel cas il est qualifié de traçage noyau. Un point de trace est l'entité responsable de la collecte de données dans le programme. Il s'agit d'un type d'instrumentation, injecté dans le programme de manière statique ou dynamique selon l'outil.

1.2 Éléments de la problématique

On trouve dans le domaine de la vérification de programme une large variété d'outils, avec une plus ou moins grande versatilité. Pour ce qui est de la vérification statique, les compilateurs offrent une bonne plateforme car ils possèdent déjà de façon interne une abstraction de la structure du programme compilé. Un outil de vérification statique peut donc se greffer à un compilateur et travailler directement sur cette représentation intermédiaire abstraite. Pour ce qui est de l'analyse dynamique, il n'existe pas de plateforme aussi naturelle. En effet, augmenter le niveau d'abstraction a un coût en performance pour le programme, et de nombreux outils nécessitent un ralentissement minime du programme visé pour être utilisables. Beaucoup d'outils s'affranchissent des infrastructures complexes à cause de leur coût

et réécrivent tous les outils nécessaires à leur fonctionnement. Ce manque de consensus sur une infrastructure commune cause donc des redondances dans l'écriture des outils de vérification, chaque outil devant développer toute son infrastructure de zéro. A son tour, ce coût de développement entraîne la plupart des développeurs à restreindre au strict minimum les fonctionnalités de leur outil, et donc leur flexibilité d'utilisation. Peu d'outils proposent ainsi une instrumentation dynamique des programmes ou même une interface graphique.

Ce problème est similaire à celui rencontré par les environnements de développement intégrés (EDI) monolithiques. Chaque éditeur de texte devait programmer un outil d'analyse pour chaque langage supporté. Ceci crée un surcoût de développement considérable, et complique la propagation de nouveaux langages et l'adoption de nouvelles fonctionnalités pour les langages existants. Une solution a été apportée par les éditeurs modulaires et le Language Server Protocol, qui instaure la notion de serveur de langage, partagé entre les éditeurs. En implémentant simplement une interface standardisée, ces derniers peuvent proposer tous les langages pour lesquels un serveur existe. Une approche modulaire permettrait donc de simplifier la tâche des développeurs et des utilisateurs dans le domaine de la vérification à l'exécution de programmes. En mettant en commun des fonctionnalités clés telles que l'instrumentation dynamique, le chargement de librairies ou encore l'interface graphique, ceci permettrait à une large variété d'outils spécialisés d'offrir ces fonctionnalités sans avoir d'impact sur le coût de développement de l'outil.

Au travers de notre travail de recherche, nous nous sommes efforcés de trouver une solution fonctionnelle offrant à la fois la modularité et les fonctionnalités souhaitées. Tout d'abord un outil central est nécessaire pour offrir les fonctionnalités de bas niveau, comme l'instrumentation. Le débogueur apparaît comme une solution naturelle. En effet, comme le compilateur pour la partie statique, cet outil versatile contient une importante infrastructure pour manipuler les fichiers binaires, notamment au travers des informations de débogage. Le débogueur permet de contrôler l'exécution d'un programme, mais également de s'attacher à un programme en cours d'exécution et d'y charger des librairies, deux fonctionnalités très intéressantes pour un outil de vérification. Un problème majeur des débogueurs pour notre tâche est l'absence d'une structure d'instrumentation efficace et versatile. L'instrumentation des programmes est faite via l'utilisation de signaux noyaux qui ralentissent fortement l'exécution des programmes débogués. Il est donc nécessaire pour nous de développer une fonctionnalité d'instrumentation performante au sein d'un débogueur afin de s'en servir comme d'une plateforme pour les outils de vérification à l'exécution.

Autour du débogueur, il est alors nécessaire de produire une architecture complète permettant de faire le lien entre utilisateur, outil de vérification et programme à déboguer. Cette

architecture doit contenir une interface graphique ainsi que de nombreux niveaux de pilotage afin d'offrir aux outils et à l'utilisateur une flexibilité maximale. L'utilisation d'un environnement de développement modulaire est pertinente, car d'une part ceux-ci offrent une grande liberté dans l'interface via l'utilisation de modules programmables, d'autre part les EDI proposent déjà une liaison vers le débogueur qui peut être étendue pour s'intégrer dans notre nouvelle architecture. Enfin, l'utilisation d'un éditeur modulaire offre également la possibilité aux outils de dialoguer avec des serveurs de langages, et ainsi de bénéficier de l'analyse du code source.

Un facteur majeur dans la création d'une nouvelle architecture est la possibilité d'y adapter simplement les outils existants. En effet, réécrire l'intégralité d'un outil est pénible et coûteux, même dans une structure généraliste. La possibilité de réutiliser l'infrastructure de collecte de données et d'analyse des outils existants est donc primordiale pour faciliter l'adaptation de ces outils. De même, il faut également s'interroger sur la distribution et l'intégration de différents outils sur une même plateforme, et définir un format standard permettant la création et le partage d'outils de façon naturelle.

1.3 Objectifs de recherche

Il n'existe pas de plateforme permettant la production d'outils de vérification dynamique de programme avec un coût en performance satisfaisant. Notre but est donc de créer une telle plateforme et d'en démontrer l'intérêt.

1.3.1 Objectifs spécifiques et complémentaires

A partir de cet objectif général, nous pouvons définir une série d'objectifs spécifiques :

- analyser les outils de vérifications existants, leurs limitations et redondances ;
- créer une plateforme d'instrumentation performante et versatile dans un débogueur en source ouverte :
- produire une architecture complète autour de ce débogueur permettant la création et l'amélioration d'outils de vérification à l'exécution ;
- évaluer cette architecture et la comparer aux systèmes existants ;
- démontrer l'intérêt de notre architecture par l'amélioration d'outils existants.

Nous pouvons également définir un objectif complémentaire, qui est l'incorporation de notre outil d'instrumentation à la source officielle du débogueur. Ceci permettrait d'une part de faciliter grandement l'adoption de notre infrastructure, et d'autre part cette fonctionnalité très versatile pourrait bénéficier d'un usage plus large, puisqu'elle peut notamment être utile

lors du débogage interactif.

1.4 Plan du mémoire

Le second chapitre de ce mémoire présente une revue critique de la littérature, où est présenté l'état de l'art en matière d'environnements de développement, d'instrumentation et de débogage.

Dans le chapitre suivant, nous décrivons l'environnement de développement et d'évaluation de notre recherche afin d'en faciliter la reproduction et vérification. Nous fournissons également un lien vers le code source libre de nos travaux.

Le chapitre 4 est un article scientifique qui présente en détail l'architecture proposée en solution ainsi qu'une évaluation de ses performances comparées à celles des outils disponibles.

Nous revenons dans le chapitre 5 sur des résultats complémentaires qui n'ont pas été présentés dans l'article, et nous fournissons également une analyse critique des résultats.

Enfin, nous concluons ce mémoire dans le chapitre 6 en offrant un sommaire des différentes contributions apportées et de leurs limitations, ainsi que des pistes sur les travaux futurs à entreprendre pour améliorer notre solution.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre vise à présenter les outils et techniques disponibles pour le développement d'un programme, de son écriture à son débogage. Une première partie décrit l'état de l'art des environnements de développement et l'intégration des différents outils de développement à ces derniers. Ensuite sont présentés les techniques d'instrumentation d'un programme, puis les logiciels de traçage disponibles. Enfin, la dernière section présente les techniques disponibles pour le débogage d'un programme.

2.1 Environnements de Développement

L'outil le plus basique que l'on retrouve dans le processus d'écriture d'un programme est l'éditeur de texte. Sa version spécialisée à l'écriture de code est appelée environnement de développement.

2.1.1 Environnements de développement traditionnels

Traditionnellement, les environnements de développements fournissent un certain nombre de fonctionnalités additionnelles absentes d'un simple éditeur de texte.

- Analyse syntaxique du code : coloration syntaxique, auto-complétion du code ...
- Intégration de la compilation et de l'exécution d'un programme
- Intégration du débogage du programme
- Intégration de la gestion de version
- Réusinage de code

Les environnements de développements intégrés (EDI) traditionnels, tels qu'Eclipse [1] ou Visual Studio [2] intègrent ces fonctionnalités avec le logiciel. Certaines fonctionnalités comme la gestion de version sont souvent déléguées à des logiciels tiers comme Git [3], mais beaucoup sont implémentées spécifiquement pour l'environnement. Ainsi Eclipse utilise un compilateur Java propre, et Visual Studio un débogueur propriétaire [4].

Intégrer toutes ces fonctionnalités pour tous les langages supportés entraîne un coût en espace disque et en mémoire conséquent. Ainsi, Visual Studio recommande un système avec 8Go de mémoire vive et requiert typiquement 20 à 50 Go d'espace disque [1]. Une solution adoptée par ces éditeurs est l'utilisation d'extensions pour traiter certains langages, afin d'éviter d'encombrer inutilement le binaire de base. Eclipse sépare ainsi l'environnement Java (JDT)

1. `docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements`

des environnements C/C++ (CDT) et PHP (PDT).

D'autres environnements tels que Vim [5] ou Notepad++ [6] adoptent une autre approche. Le logiciel initialement très léger peut être adapté aux besoins de l'utilisateur grâce à un système d'extensions. Cependant, ces extensions et en particulier les extensions d'analyse syntaxique de code nécessitent un développement spécifique pour chaque plateforme. Le développement d'outils similaires est donc reproduit en parallèle pour tous les éditeurs traditionnels, ce qui crée une charge de travail redondante.

2.1.2 Environnements de développement modulaires

Une alternative introduite par Microsoft en 2015 avec Visual Studio Code [7] est la mise en commun entre les environnements de développement de fonctionnalités, notamment l'analyse syntaxique, via des protocoles standardisés.

Debug Adapter Protocol

La plupart des environnements de développement populaires offrent une interface graphique de débogage utilisant des débogueurs existants, tels que GDB [8] pour les langages C et C++ ou `pdb` [9] pour le langage Python. Cependant, ces outils ont chacun une interface propre et c'est donc l'EDI qui est responsable la mise en place d'une communication spécifique vers chaque débogueur.

Le Debug Adapter Protocol (DAP) [10] offre une alternative en ajoutant un intermédiaire entre le débogueur et l'environnement de développement appelé extension de débogage. Cette extension spécifique à un débogueur a pour but de traduire les messages de l'environnement pour le débogueur et inversement.

Entre l'environnement de développement et l'extension, la communication est standardisée et basée sur un format JSON. Il est indépendant du langage débogué sous-jacent et de l'éditeur. Par conséquent, il n'est pas nécessaire pour chaque éditeur de développer une interface spécifique à chaque débogueur. Il est également envisageable que des débogueurs introduisent une interface directement compatible avec le Debug Adapter Protocol pour s'affranchir de l'extension.

On retrouve pour la plupart des langages un débogueur pour lequel a été implémentée une extension de débogage [2], notamment GDB et LLDB pour le C++ [3].

2. `microsoft.github.io/debug-adapter-protocol/implementors/adapters/`
3. `code.visualstudio.com/docs/cpp/cpp-debug`

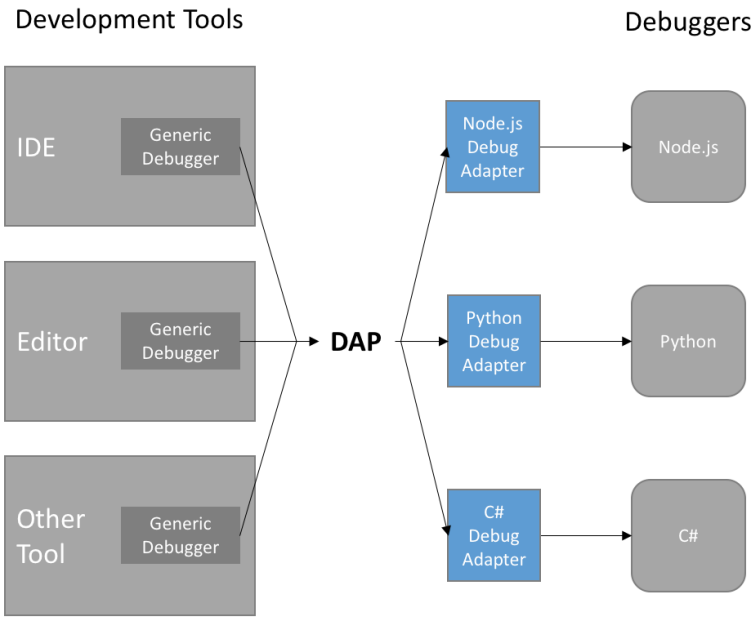

Figure 2.1 le Debug Adapter Protocol : une interface unique pour les différents débogueurs

Source : microsoft.github.io/debug-adapter-protocol/overview

Language Server Protocol

Le Language Server Protocol (LSP) est un protocole de communication introduit par Microsoft en 2015 [11]. Il a pour but de faire interagir un éditeur de code avec un serveur de langage, chargé d'analyser le code et de renvoyer les informations à l'éditeur. Par exemple, le serveur de langage peut fournir à l'éditeur la position de toutes les références à une variable dans un fichier source, en excluant les variables homonymes, ou encore pointer vers la définition d'une fonction. Contrairement au cas du Debug Adapter Protocol où une interface est établie vers des débogueurs existants, les serveurs de langage sont autonomes et doivent offrir directement une interface au format spécifié.

Cette approche permet d'isoler le développement de l'éditeur de texte de celui des analyseurs de code. Ainsi, lorsqu'un nouveau langage est introduit ou qu'un langage existant est mis à jour, seuls les serveurs de langages correspondants doivent être mis à jour, et non les éditeurs.

Les fonctions d'un serveur de langage étant souvent nécessaires au compilateur, un même outil peut être adapté pour faire les deux tâches, réduisant encore ainsi le besoin de développement parallèle d'une même fonctionnalité. C'est le cas pour le compilateur C et C++ `clang` de la suite LLVM qui fournit un serveur de langage nommé `clangd` [12].

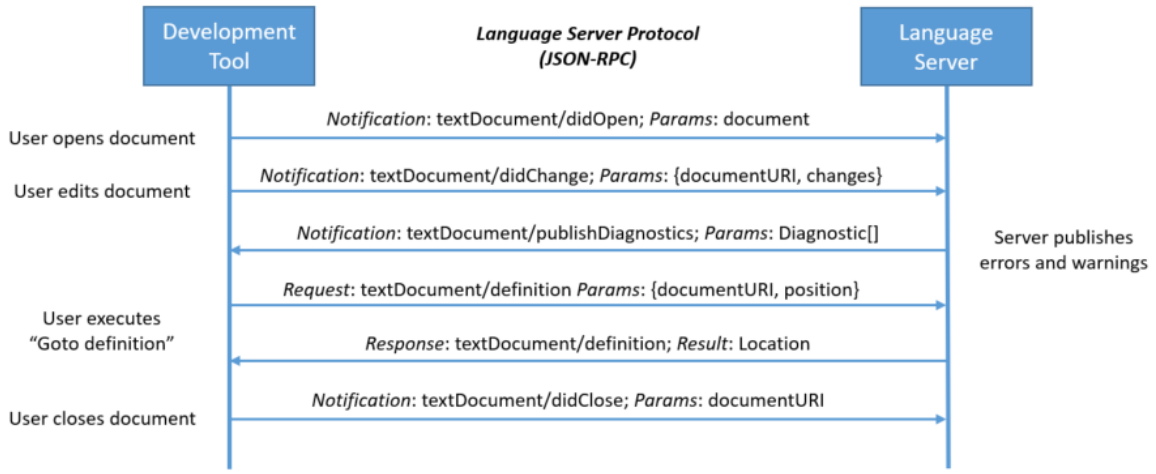

Figure 2.2 Un exemple d'interaction entre éditeur et serveur de langage

Source : microsoft.github.io/language-server-protocol/overviews/lsp/overview/

Des serveurs de langages sont disponibles pour la plupart des langages [4] implémentant la totalité ou une partie des fonctionnalités de la spécification.

Éditeurs disponibles

Visual Studio Code

Visual Studio Code (VSCode) est un environnement de développement modulaire en source ouverte développé par Microsoft et publié en 2015 [7]. Il s'agit du premier éditeur supportant les protocoles LSP et DAP qui ont été créés pour lui. Il s'agit d'un environnement hautement personnalisable, tant au niveau de l'interface utilisateur que des différents modules qui peuvent être chargés. VSCode intègre un module d'analyse de syntaxe pour le Javascript, laissant la prise en charge des autres langages à des modules qui peuvent être téléchargés via un outil de recherche intégré. Les modules d'extensions sont écrits en Javascript ou Typescript, et peuvent être publiés sur le catalogue librement, sous réserve cependant d'un certain enregistrement vis à vis de Microsoft Azure [5].

Visual Studio code utilise le cadriciel Electron [13] et Node.js [14], et est écrit en Typescript et Javascript.

VSCode offre également la possibilité de travailler sur une machine distante ou un conteneur via l'extension de développement à distance [15] publiée par Microsoft. Cependant, cette extension est propriétaire et en source fermée car elle partage du code avec le logiciel commercial

4. `langserver.org/`
5. `code.visualstudio.com/api/working-with-extensions/publishing-extension`

Visual Studio [6].

Ce logiciel publié en 2015 est l'environnement de développement le plus populaire depuis 2018 selon les sondages 2018 et 2019 du site Stack Overflow [7] [8] portant respectivement sur 101,592 et 88 883 développeurs dans le monde entier [9] [10].

Eclipse Theia

Eclipse Theia est un environnement de développement en source ouverte maintenu par la fondation Eclipse et publié en 2017 [16]. Assez similaire à VSCode, il reprend en effet l'éditeur de texte Monaco ainsi que l'apparence générale du logiciel. Theia est cependant plus modifiable, permettant à la fois l'addition de modules comme VSCode et l'addition d'extensions qui modifient plus profondément la structure de l'environnement. Il est également conçu pour opérer à distance de façon native, avec par exemple le système de fichier dans un conteneur et l'interface dans un fureteur.

L'interface des modules VSCode est également compatible avec Theia, rendant la migration de modules d'un service vers l'autre possible.

Intégrations à d'autres environnements

L'essor de VSCode et la publication de nombreux serveurs de langages ont rendu intéressant pour les environnements de développement traditionnels le support du LSP et du DAP. Parmi les environnements supportant le Language Server Protocol et le Debug Adapter Protocol, on compte Vim, Emacs et Eclipse (sous le nom LSP4E) [11] [12]. D'autres outils ne supportent que le LSP, comme IntelliJ.

2.2 Instrumentation

Lors de l'écriture ou l'évaluation d'un programme, il est souvent utile d'inclure des éléments non nécessaires à l'exécution. Typiquement, on retrouve dans cette catégorie des instructions facilitant le débogage ou l'analyse de performance du programme. Ces instructions non essentielles sont ce que l'on nomme ici l'instrumentation du programme.

L'instrumentation d'un programme peut se faire principalement à quatre niveaux : au niveau du code source, au niveau du compilateur, au lancement du programme et pendant

6. `code.visualstudio.com/docs/remote/faq`
7. `insights.stackoverflow.com/survey/2018`
8. `insights.stackoverflow.com/survey/2019`
9. `insights.stackoverflow.com/survey/2018#methodology`
10. `insights.stackoverflow.com/survey/2019#methodology`
11. `https://microsoft.github.io/debug-adapter-protocol/implementors/tools/`
12. `langserv.org`

l'exécution. On qualifiera les deux premières approches de statiques, et les deux dernières de dynamiques.

2.2.1 Instrumentation statique

L'instrumentation statique a lieu avant l'exécution du fichier binaire. La forme la plus simple est l'insertion de commande d'impression pour suivre le déroulement d'un programme. Ceci dit, une telle méthode a un impact important sur la performance et la lisibilité du programme. On peut donc implémenter des versions plus efficaces où ce suivi est uniquement activé par l'utilisation d'un certain argument à l'exécution du programme. De très nombreux programmes proposent ainsi une option `verbose` qui permet d'ajuster la quantité de données affichée.

Une autre forme d'instrumentation qui évite d'encombrer le code source est l'instrumentation par le compilateur. Le compilateur GCC [17] propose un grand nombre d'instrumentations possibles à la compilation [13], comme l'addition au binaire d'informations de débogage. Certains types d'instrumentations statiques sont destinés à être accompagnés par le chargement dynamique d'une librairie, comme l'option `fsanitize=address` qui instrumente les accès mémoire et joint la librairie de l'outil Address Sanitizer [18].

2.2.2 Instrumentation Dynamique

L'instrumentation dynamique d'un programme est la modification d'un programme pendant son lancement ou son exécution. Plusieurs méthodes existent, chacune utilisée par un ou plusieurs outils.

Utilisation de signaux

Une technique basique est de remplacer les instructions que l'on souhaite instrumenter par une instruction provoquant un signal. Lorsque l'instruction est atteinte, le noyau du système d'exploitation donne le contrôle au programme responsable de l'instrumentation qui peut ensuite exécuter la commande voulue puis rendre le contrôle au programme instrumenté. Parmi les programmes qui utilisent cette approche, on retrouve la majorité des débogueurs, par exemple GDB [8] et LLDB [19], et certains outils de traçage comme Uprobe [20]. Bien que facile à mettre en place et versatile, cette approche a l'inconvénient de devoir donner le contrôle au noyau du système d'exploitation à chaque fois qu'une instruction instrumentée est exécutée, produisant donc deux changements d'anneau de protection. Ceci résulte en un fort impact sur les performances quand beaucoup d'instructions sont instrumentées.

13. `gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`

Utilisation de trampolines

Pour s'affranchir de ces changements d'anneaux de protection, une solution possible est d'exécuter le code d'instrumentation dans le contexte du programme et d'y envoyer le programme directement, sans intervention de l'outil d'instrumentation ni du noyau pendant l'exécution. Il s'agit donc de remplacer l'instruction à instrumenter par une instruction de saut vers le code d'instrumentation. On modifie ainsi le flux d'exécution du programme. Cette approche, bien que nettement plus efficace en termes de rapidité d'exécution présente un certain nombre de difficultés. D'une part, il faut exécuter l'instruction que l'on a remplacée à une position différente, ce qui peut être problématique pour certaines instructions utilisant des références à la position dans le code. D'autre part, il faut pouvoir placer une instruction de saut sur l'instruction souhaitée. Dans les architectures où les longueurs d'instruction sont variables, ceci est un problème majeur. Pour x86 notamment, il existe des instructions de sauts de 2, 3, 5 octets de long [21] avec des portées respectives de 2^7, 2^{15}, et 2^{31} octets. Il faut donc pouvoir trouver une zone de taille suffisante à portée de l'instruction instrumentée pour y placer le code d'instrumentation. Dans la plupart des cas, pour simplifier cette tâche, on utilise un trampoline, c'est à dire que le saut se fait vers une petite portion de code, qui elle-même exécute un saut plus long vers la portion du code correspondant à l'instrumentation.

Parmi les outils utilisant cette approche, on trouve Dyninst [22] [23]. Celui-ci permet à une application d'instrumenter un programme tiers par l'ajout de trampolines. Dyninst utilise une approche à deux trampolines qui permet de pré-instrumenter et post-instrumenter l'instruction choisie. Le pilotage de cet outil se fait via une API en C++ [24] qui permet de définir et de placer des morceaux de code appelés "snippets". Cet outil permet également l'instrumentation statique d'un exécutable sur disque, entre la compilation et l'exécution. Un inconvénient de cette approche est que l'écriture et l'injection de code ne se fait pas de manière naturelle et nécessite l'utilisation d'un langage spécifique.

Une alternative à Dyninst sur l'architecture x86_64 est LiteInst [25], un outil permettant de s'attacher à un programme et d'y ajouter des points d'instrumentation. Il utilise également une approche basée sur des trampolines, mais prend en charge l'instrumentation des instructions courtes différemment. Il utilise une technique nommée "punning" qui se base sur la dualité instruction / données d'une valeur en mémoire. En particulier, l'argument d'une instruction de saut (le décalage) peut également être interprétée comme une autre instruction si l'exécution débute au milieu de celui-ci, par exemple en raison d'un autre saut dans le code. En plaçant le trampoline de façon appropriée, il est donc possible de minimiser la modification du code original, en ne changeant qu'un ou deux octets parmi les cinq concernés par exemple. Si maintenir les instructions d'origine n'est pas possible, LiteInst insère à la place

des instructions générant un signal (SIGINT ou SIGILL). Ces signaux ne sont activés que si le programme saute directement au milieu de l'instruction de saut, ce qui arrive rarement (entre 0% et 14% des points d'instrumentation touchés dans l'article). Cette approche est plus rapide que Dyninst pour l'instrumentation d'une seule instruction car elle s'affranchit du coût très conséquent d'analyse fait par Dyninst lors de son initialisation. Cependant, cet avantage s'annule dès que Dyninst instrumente de nombreuses instructions en une fois, le coût d'instrumentation d'une instruction sur Dyninst après initialisation étant environ deux fois moins important que celui de LiteInst.

Plusieurs applications de traçage utilisent également un système de trampolines pour ajouter dynamiquement des points de trace, notamment le traceur du débogueur GDB [26] et ftrace [27].

Recompilation du code

Le problème majeur de l'utilisation de trampolines est que tout le code ajouté doit s'insérer dans la mémoire du binaire en causant le moins de perturbation possible, et donc s'adapter à une structure inconnue et variable. Une solution pour s'affranchir de ce problème est de réécrire totalement le binaire dynamiquement, en ajoutant les portions de code voulues. Cette technique est nommée recompilation dynamique.

Un outil basé sur ce principe est la plateforme d'instrumentation PIN [28] développée par Intel en source fermée. Elle permet l'insertion de fragments de code dans un binaire sur les architectures Intel 32 et 64 bits, mais également ARM. Cette instrumentation se fait par la recompilation en temps réel du programme. Le code initial est exécuté dans une machine virtuelle et les portions utilisées sont recompilées en temps réel pour être exécutées sur le processeur, après ajout de l'instrumentation requise. Un avantage majeur de cette approche est la liberté totale d'instrumentation du programme. Cependant, la recompilation du programme vient avec un coût en performance même si aucune instrumentation n'est faite. Cette approche n'est donc pas adaptée à une instrumentation localisée du programme, où seules quelques instructions sont instrumentées.

L'outil Valgrind [29] utilise également une forme de recompilation dynamique du code. Le code en entrée est traduit vers une forme interne indépendante de l'architecture, instrumenté, et ensuite recompilé en instructions machine qui sont exécutées sur une machine virtuelle. Un aspect important de Valgrind est la présence de "mémoire fantôme" : chaque registre ou variable possède une version fantôme qui peut enregistrer des métadonnées sur les données enregistrées. Les outils développés sur la plateforme Valgrind sont libres d'utiliser cette espace fantôme pour leurs analyses. Dans le cas de l'analyseur de fautes en mémoire Memcheck [30], cet espace fantôme est notamment utilisé pour enregistrer quels octets sont initialisés. Cette

méthode d'instrumentation est particulièrement lente, mais permet une analyse plus poussée que les autres.

Performance des différentes approches

Pour comparer la performance de ces outils, deux approches sont envisageables. D'une part, on peut mesurer si la performance du programme est affectée par l'utilisation de l'outil sans instrumentation. D'autre part, on peut mesurer à quel point l'exécution du programme instrumenté est ralentie par l'outil dans le cas d'une instrumentation.

Sans instrumentation, les approches utilisant les signaux et les trampolines ne sont pas ralenties puisqu'elles ne modifient pas l'exécution du programme. Pin a un surcoût d'environ 60% pour les opérations entières et 5% sur les opérations flottantes [28]. Valgrind entraîne un ralentissement d'environ 300% sans instrumentation [31].

Pour l'instrumentation, une mesure classique se fait sur le comptage des blocs unitaires. L'outil a pour but de compter combien de fois chaque bloc de code source d'un programme est exécuté. Bernat et Miller [22], Luk et al. [28] et Nethercote et Seward [29] fournissent des données différentes sur plusieurs exécutables tests. Il en ressort que les plateformes PIN et Dyninst ont un surcoût assez similaire, avec un léger avantage à Dyninst. Ce surcoût se situe typiquement entre 10 et 200%. V. Zhao [32] compare les temps totaux d'exécution (instrumentation et exécution) de PIN et LiteInst sur différents bancs d'essais, LiteInst prend typiquement entre 1 et 15 fois le temps nécessaire à PIN. Cependant, LiteInst est plus efficace que PIN sur l'instrumentation d'outils légers avec peu de points d'instrumentation. Valgrind est environ trois fois plus lent que ces outils.

L'instrumentation par signaux est encore nettement plus lente pour ce genre de tâche. Une comparaison de Dyninst avec GDB par Buck et Hollingsworth [33] décrit un ralentissement allant jusqu'à 900 fois pour l'instrumentation d'un exécutable comprenant de nombreuses opérations par seconde.

2.3 Traçage

Une forme importante d'instrumentation est le traçage, c'est à dire l'enregistrement d'informations sur l'exécution d'un programme. Le traçage permet d'une part de suivre le comportement d'un programme et de vérifier qu'il est conforme aux attentes du programmeur, mais il permet surtout d'analyser la performance d'un programme, ce que l'intrusivité d'un débogueur ne permet souvent pas de bien diagnostiquer. Un des critères principaux dans la sélection d'un traceur est donc sa rapidité, et plus généralement son faible impact sur l'exécution du programme tracé.

2.3.1 Traçage statique

La notion de traçage statique se réfère ici au type d'instrumentation utilisé pour placer les points de trace. Un traceur statique utilisera donc une instrumentation statique de l'exécutable.

Ftrace

Ftrace [27] est le traceur standard du noyau Linux. Il utilise un mécanisme de trampolines pour activer ou non pendant l'exécution du noyau un certain nombre de points de trace définis au préalable. Quand le noyau est configuré pour autoriser le traçage, le compilateur réserve un certain espace dans les fonctions (avec des instructions NOP) que Ftrace va modifier pour positionner des trampolines qui appellent le code de traçage. Les données sont ensuite enregistrées sur des tampons circulaires pour éviter la saturation. Chaque cœur possède son propre tampon pour éviter de ralentir les autres cœurs.

LTTng

LTTng [34] est un traceur statique de haute performance pour Linux en source libre, développé par Efficios. Initialement conçu comme un traceur pour le noyau Linux uniquement, il existe depuis 2009 une version pour les applications en espace usager [35]. L'idée derrière LTTng est de minimiser autant que possible l'utilisation de verrous entre les cœurs d'un processus. Chaque cœur possède donc un tampon circulaire pour stocker les données de trace. Ces tampons peuvent être sauvegardés et vidés pendant l'exécution, notamment pour l'analyse de trace en temps réel [36], ou être sauvegardés à la demande, par exemple lorsqu'un problème de performance est remarqué. Ce dernier mode est appelé mode instantané [37]. LTTng utilise le format de trace Common Trace Format (CTF) [38], un format binaire et compact qui permet de minimiser l'espace mémoire consommé.

Le traceur noyau utilise les mêmes points de trace que Ftrace. Le traceur usager permet de définir des points de trace dans le code source d'un programme, ainsi que les données enregistrées. Par la combinaison des traces noyaux et usagers, LTTng permet l'analyse avancée d'un programme, notamment en mettant en évidence les causes du ralentissement d'un appel système, ce dont est incapable un traceur purement usager [39].

TAU

Tuning and Analysis Utilities (TAU) [40] est un outil de traçage et de profilage développé par l'université d'Oregon. Il permet le traçage d'applications en espace usager. Un intérêt de TAU est que l'instrumentation peut être faite de plusieurs manières. Il est ainsi possible d'instrumenter du code manuellement, automatiquement à la compilation, et dynamique-

ment grâce à l'utilisation de Dyninst [14]. TAU peut également être utilisé pour le profilage d'application, c'est à dire pour agréger différents compteurs afin d'obtenir une vision plus générale sur l'exécution du programme. Une métrique de profilage est par exemple le temps processeur moyen d'exécution d'une fonction.

OpenTracing

Le standard OpenTracing [41] a pour but de fournir un format partagé pour le traçage distribué. En effet, la plupart des traceurs sont prévus pour tracer un processus exécuté sur une seule machine, or le développement de l'informatique distribuée a créé un besoin de traçage réparti qui puisse mettre en relation les évènements provenant de différentes machines.

Le standard OpenTracing définit un traitement standard des évènements, ainsi qu'un mécanisme de communication de ces évènements. Il est implémenté par plusieurs traceurs en source ouverte, notamment Jaeger [42] développé par Uber, et Zipkin [43] développé par Twitter. Dans tous les cas, les points de trace sont implantés dans le code à l'écriture.

2.3.2 Traçage dynamique

Le traçage dynamique peut couvrir différentes techniques, notamment l'activation pendant l'exécution de points de trace définis statiquement. Ici nous nous intéressons uniquement aux outils de traçage capables d'instrumenter dynamiquement le programme.

Un avantage majeur du traçage dynamique est la possibilité d'instrumenter librement un programme déjà compilé, en particulier pendant une exécution repérée comme anormale.

Dtrace

DTrace [44] est un traceur du noyau et des applications usagers initialement développé pour le système d'exploitation Solaris. Il est à la fois statique et dynamique. En effet, il permet d'une part de définir avant la compilation des points de trace statiques où le compilateur installera des instructions NOP, remplacées au besoin pendant l'instrumentation par des points de trace. D'autre part, il permet d'ajouter des points de trace dynamiquement par le biais de sondes. Pour chaque point d'instrumentation, DTrace laisse l'usager définir un script dans un langage nommé *D* proche du C. Dtrace nécessite l'ajout d'un module noyau qui traite chaque point de trace. Pour le traçage dynamique d'applications en espace usager, l'outil force un passage par le noyau pour chaque point de trace, ce qui ralentit nettement l'exécution du programme. Dtrace est ainsi environ 10 à 50 fois plus lent que LTTng par exemple [15].

14. `cs.uoregon.edu/research/tau/docs/usersguide/ch01.html`
15. `dorsal.polymtl.ca/fr/blog/yannick-brosseau/userspace-tracing-comparison-dtrace-vs-lttng-ust`

GDB Tracing

Le débogueur GDB [8] offre une interface de traçage sur des machines distantes via l'outil `gdbserver` [26]. Cet outil permet de s'attacher à un programme en cours d'exécution et de l'exécuter depuis une machine distante. L'interface de traçage de GDB offre deux possibilités : d'une part un traçage basé sur des signaux captés par le noyau, ce qui permet de placer les points de trace n'importe où mais est particulièrement lent, d'autre part des points de trace rapides fonctionnant avec un système de trampolines. Ces points de trace ne pouvant être placés que sur les instructions de taille suffisante, les possibilités de placement sont limitées sur les architectures à longueur d'instruction variable, comme x86.

Le système d'enregistrement des traces est lui-même assez inefficace, puisque GDB écrit dans un tampon partagé entre tous les fils d'exécution du programme. Si deux fils cherchent à écrire en même temps dans ce tampon, l'un restera bloqué en attente du verrou détenu par le premier jusqu'à que ce dernier ait fini.

Extrae

Extrae est un outil d'instrumentation de systèmes parallèles développé par le Centre National de Supercalcul à Barcelone (BSC). Il permet d'analyser les appels aux fonctions des librairies de calcul parallèle, notamment pthread, MPI, OpenMP, Cuda et OpenCL. Typiquement, cet outil s'interpose entre le programme et les librairies par le préchargement de librairies d'instrumentation avec la fonctionnalité "LD_PRELOAD", mais il peut aussi utiliser Dyninst [45] ou utiliser une instrumentation statique. Extrae utilise un format de trace propre. Conçu pour les systèmes à haute performance, Extrae installe des points de trace avec une latence particulièrement faible grâce à l'utilisation d'un tampon par fil d'exécution, selon Gebai et Dagenais [46]. A noter cependant que cet article ne traite que de la version statique d'Extrae ; une instrumentation basée sur Dyninst aura probablement un coût légèrement supérieur.

Extrae offre peu de liberté dans les données enregistrables par un point de trace. Seul un type de point de trace existe, qui ne permet pas à l'usager d'enregistrer les valeurs de variables.

2.3.3 Analyse de trace

Paraver

Paraver [47] est l'outil d'analyse de trace développé par le BSC. Principalement axé sur l'étude en performance de systèmes fortement parallèles, il permet l'analyse de traces récupérées à l'aide de l'outil Extrae, par plusieurs types de représentation : d'une part des visualisations graphiques de l'évolution d'état d'un thread ou l'évolution de la valeur d'une variable ; d'autre part, une représentation textuelle qui permet d'obtenir des informations détaillées sur un

évènement précis. Enfin, une représentation appelée analyse, où des mesures sur une portion de la trace peuvent être compilées pour faire apparaître des tendances.

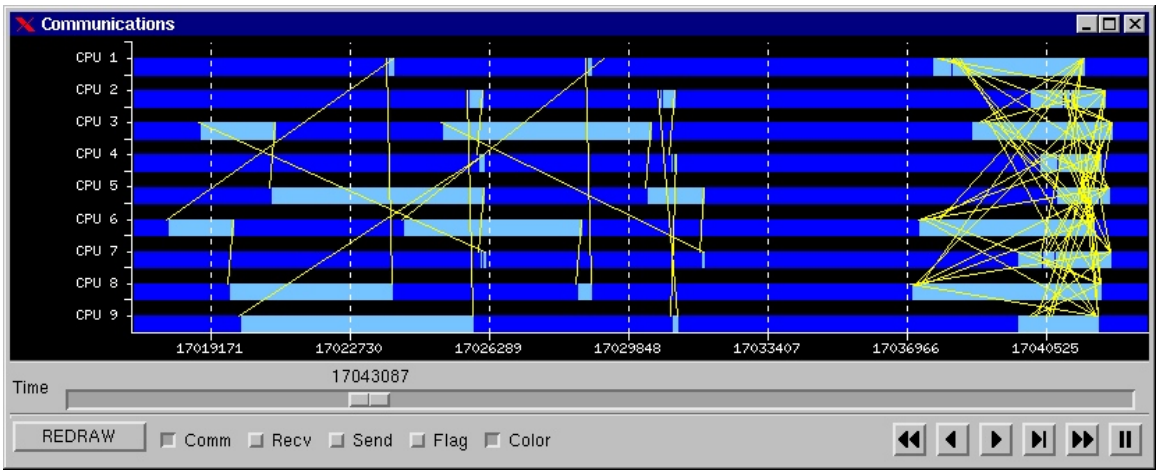

Figure 2.3 Vue des communications entre processeurs dans Paraver

Source : tools.bsc.es/paraver#views

Chrome Viewer

Le fureteur Chromium [48] développé par Google offre un outil de visualisation de traces intégré. Il accepte les traces au format Trace Event, un format textuel utilisant un fichier JSON, mais également d'autres formats comme le format Ftrace [49]. Il offre une interface graphique qui permet de naviguer simplement dans une trace, et notamment de visualiser les piles d'appels correspondant à l'exécution d'un thread. Initialement développé pour déboguer le fureteur, il permet de mettre en relation une trace avec l'affichage d'une page web (Figure 2.4).

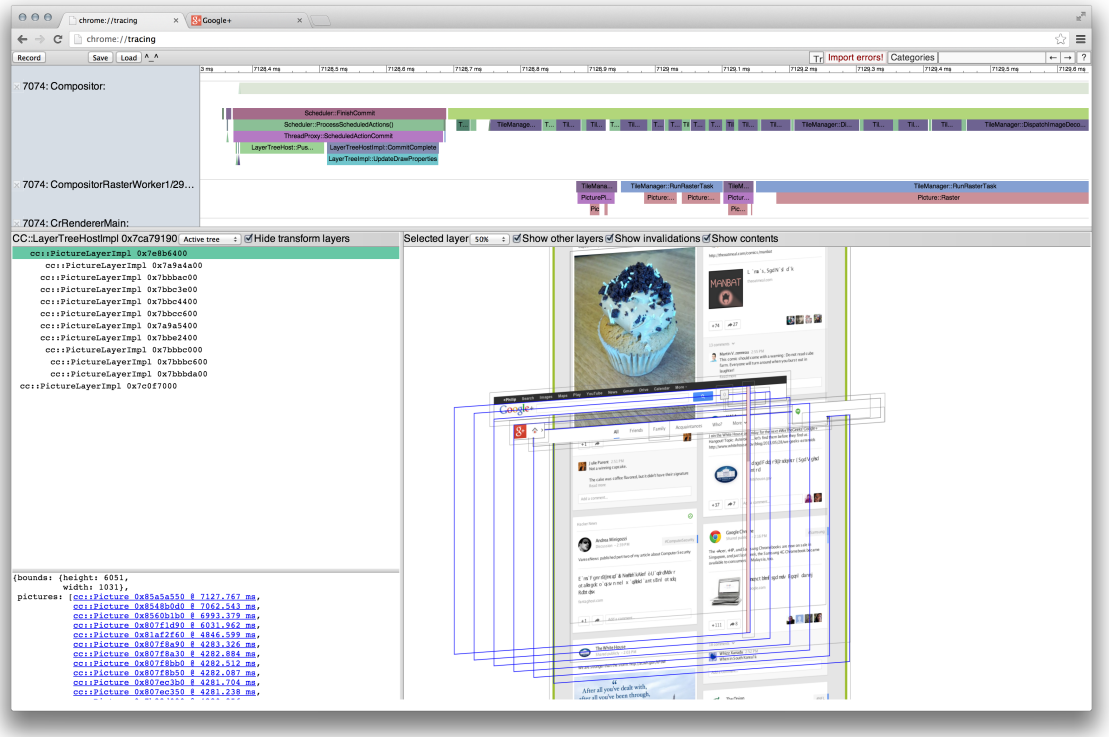

Figure 2.4 Affichage d'une trace dans Chromium

Source : chromium.org/developers/how-tos/trace-event-profiling-tool

Trace Compass

Trace Compass [50] est un logiciel de visualisation de traces en source ouverte basé sur l'environnement de développement Eclipse. Il permet d'ouvrir de nombreux formats de trace, parmi lesquels le format CTF de LTTng, le format Trace Event et le format de traces GDB. Trace Compass interprète les traces en utilisant un automate à états qui permet d'offrir un certain nombre de vues, notamment la vue de flux de contrôle où on peut suivre l'activité de chaque fil d'exécution ou la vue matériel qui permet de suivre l'activité des différents processeurs. Trace Compass offre la possibilité de programmer d'autres vues et analyse en utilisant des fichiers XML.

Il est également possible d'écrire des scripts afin d'analyser la trace et de créer de nouvelles vues [51], en Python ou en Javascript.

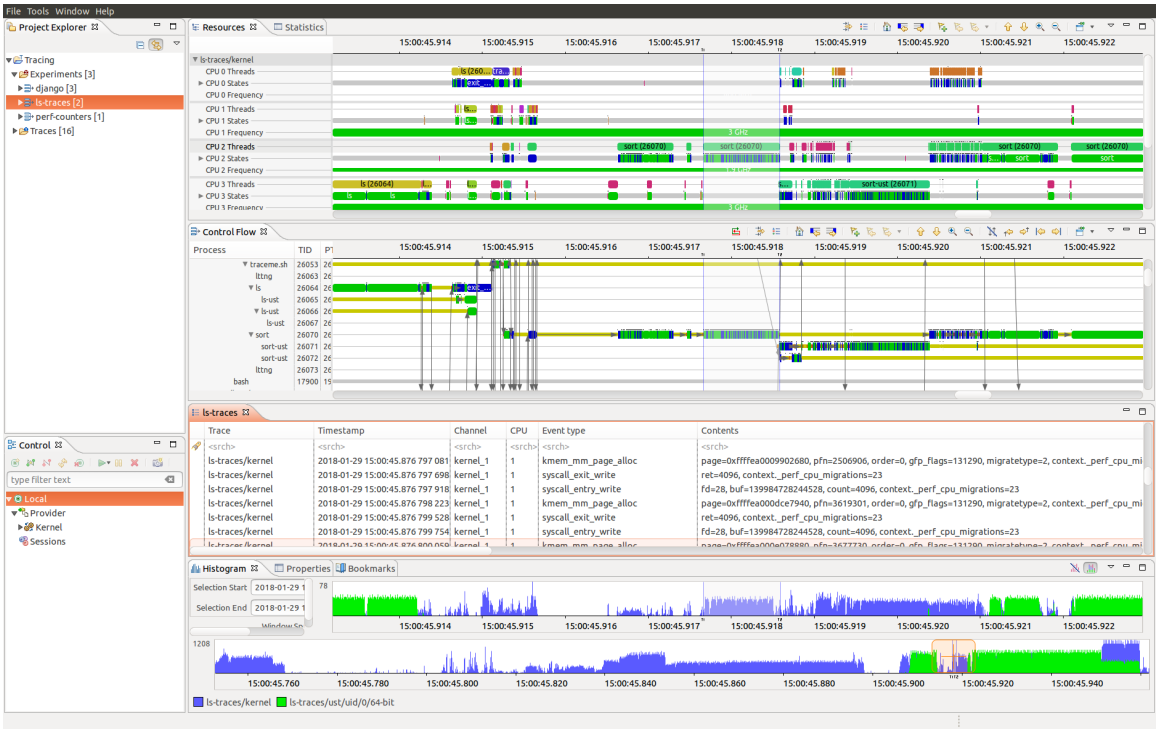

Figure 2.5 Visualisation d'une trace noyau dans Trace Compass

Source : eclipse.org/tracecompass/

2.4 Débogage

Au cours de la vérification ou de l'utilisation d'un programme, un problème peut être identifié ou suspecté. Ce problème, ou bogue, va généralement correspondre à l'une de ces trois catégories :

— Problème d'exécution : le programme échoue avec une erreur et termine prématurément
— Problème de correction : la sortie n'est pas conforme à la spécification
— Problème de performance : le programme nécessite trop de ressources de calcul ou de mémoire

Le débogage consiste en l'élimination de ce problème. Pour ceci, l'agent de débogage (utilisateur ou programme) doit appliquer la méthode scientifique. Il s'agit donc dans un premier temps d'émettre une hypothèse sur le comportement d'une portion du programme, puis de confirmer ou d'infirmer cette hypothèse par l'observation. En itérant, on peut alors cibler précisément l'origine du problème.

Il faut donc se munir d'outils permettant d'observer le comportement des sous-parties d'un programme. L'un des outils les plus simples est d'imprimer à l'écran les variables sur lesquelles

portent les hypothèses, pratique appelée débogage *printf*. Bien que simple, cette approche a des inconvénients majeurs, notamment parce qu'elle est manuellement ciblée *a priori* par l'usager. Pour chaque hypothèse, il faut donc aller modifier le code source pour afficher ce qui nous intéresse. Il y a donc encombrement du code et possiblement de la sortie, ainsi qu'un surcoût conséquent à chaque itération du débogage.

Pour rendre ce processus plus efficace, des programmes de débogage, appelés débogueurs, ont été développés.

2.4.1 Débogueurs interactifs

Un logiciel de débogage, ou débogueur, est un programme permettant d'observer le comportement interne d'un programme cible. Une des fonctionnalités principales est l'utilisation de points d'arrêt qui interrompent l'exécution du programme à un certain niveau, afin de pouvoir accéder à l'état des variables et registres à ce moment précis de l'exécution sans avoir à modifier le code source du programme. Ils permettent également d'exécuter un programme ligne par ligne, ou instruction par instruction.

Information de débogage

Pour s'exécuter, un programme ne nécessite que les adresses en mémoire des différentes instructions, fonctions, variables et constantes qu'il va utiliser. Cependant, pour qu'un utilisateur puisse déboguer efficacement un programme, il a besoin d'un lien entre les adresses mémoire et le code tel qu'il a été donné au compilateur. Ce dernier va donc pouvoir produire, en plus du code binaire exécutable, des fichiers d'information de débogage selon un format standard. Sur les plateformes Windows, cette information est stockée dans le format binaire Portable Execution / Common Object File Format [52]. Sur les autres plateformes, le format principal est DWARF [53]. Le format DWARF consiste en un graphe dont les nœuds sont appelés Debug Information Entry (DIE : nœud d'information de débogage). Chaque DIE a une étiquette et des champs, qui eux-mêmes peuvent référencer un autre DIE. Par exemple pour un DIE représentant une fonction, les champs pourront être son nom, son origine dans le code source (fichier et ligne), son type de sortie (un autre DIE), et une adresse de début ou de fin.

Ces informations de débogage sont lues par le débogueur pour reconstruire les tables de symboles, qui font le lien entre nom du symbole, adresse en mémoire et position dans le code source, permettant ainsi à l'usager de faire référence aux symboles de façon naturelle. Via le débogueur, l'usager peut ainsi accéder aux adresses et valeurs des différentes variables du

Tableau 2.1 Un exemple de DIE du format DWARF

SUBPROGRAM

Name	main
File	main.c <1>
Line	1
Type	int <0x4a>
LowPC	0x114a
High PC	0xb

programme.

Points d'arrêt

La fonctionnalité principale permettant le contrôle du flux d'exécution d'un programme est l'utilisation de points d'arrêt (*breakpoints*). Le principe d'un point d'arrêt est d'interrompre l'exécution du programme en soulevant une exception au niveau du noyau quand le point d'arrêt est touché. Il se situe en général sur une instruction exécutable, mais peut également être placé sur des données, auquel cas il sera nommé point d'observation (*watchpoint*).

Il existe deux types de points d'arrêt : les points d'arrêt logiciels et les points d'arrêt matériels [54]. Un point d'arrêt logiciel consiste à remplacer une instruction par une autre qui va soulever une exception (généralement SIGTRAP). Une fois l'exception levée, le noyau donne le contrôle au débogueur. Le nombre de points d'arrêt logiciel concurrents dans un programme est limité uniquement par la capacité du débogueur. Il existe également des points d'observation logiciels, pour lesquels le débogueur est forcé d'exécuter le programme pas à pas en vérifiant à chaque étape si la valeur a changé [16]. Ceci a pour effet d'énormément ralentir l'exécution du programme. D'autres techniques existent pour limiter l'impact des points d'observation logiciels, notamment la protection en écriture des pages mémoire, qui peut être accompagnée d'une instrumentation dynamique des accès [55]. Ceci peut réduire l'impact sur la performance, ralentissant environ 3 fois le programme, contre plus de 1000 fois dans un débogueur classique.

Dans le cas d'un point d'arrêt matériel, le processeur contient un certain nombre de registres qui, s'ils contiennent une adresse, feront soulever une exception quand cette adresse est lue, écrite ou exécutée par le processeur. Le principal avantage des points d'arrêt matériels est qu'ils permettent de poser des points d'observation sans avoir d'impact sur la performance du programmé débogué. Contrairement aux points d'arrêt logiciels, ils ne modifient pas la mé-

16. sourceware.org/gdb/current/onlinedocs/gdb/Set-Watchpoints.html

moire du programme et peuvent donc être utilisés pour contourner des restrictions imposées par le producteur du programme pour éviter que son programme ne soit modifié [56] [57]. Le problème principal est qu'il n'est possible d'en avoir qu'un nombre très limité de façon simultanée : 4 pour l'architecture x86, 1 pour les architectures PPC64 et S390 [54].

Points d'arrêt conditionnels

Plusieurs débogueurs permettent également la création de points d'arrêt conditionnels, c'est à dire de points d'arrêt qui ne donnent le contrôle à l'utilisateur que si certaines conditions sont remplies dans le programme, par exemple si la valeur d'une variable répond à un certain critère. Une implémentation classique des points d'arrêt conditionnels est de vérifier si la condition est remplie juste avant de rendre le contrôle à l'utilisateur, c'est à dire après avoir reçu le contrôle par le noyau suite à l'exception soulevée par le point d'arrêt. Si la condition est remplie, le contrôle est donné, et sinon l'exécution du programme est reprise. Cependant, cette approche a le désavantage de devoir passer inutilement par le noyau au prix d'un changement d'espace. Elle est donc coûteuse en temps dans le cas où la condition est rarement remplie.

Une seconde approche utilise des points d'arrêt conditionnels rapides [58]. Dans ce cas, le débogueur modifie le programme de façon à ce qu'il ne soulève une exception que dans le cas où la condition est vérifiée. La vérification se fait donc dans le contexte du programme débogué, et il n'y a changement de contexte que quand le point d'arrêt est effectivement valide.

Débogueurs disponibles

- **GDB** [8] : Le débogueur GNU (GDB) est le débogueur écrit pour la suite GNU. Il s'agit d'un logiciel libre maintenu par la "Free Software Foundation". Il supporte principalement les langages C et C++, mais également d'autres langages comme Go et Python. Il fonctionne sur plus d'une quinzaine d'architectures différentes, notamment x86, ARM et PPC64.
- **LLDB** [19] : LLDB est le débogueur de la suite LLVM. Dans l'esprit de la suite, ce logiciel a une architecture modulaire, utilisant des portions communes notamment avec le compilateur Clang de la suite LLVM. Il s'agit également d'un logiciel libre, maintenu par le groupe de développement LLVM. Il supporte les langages C, Objective-C et C++, sur les architectures x86 et ARM.
- **WinDbg** [59] : WinDbg est un débogueur propriétaire développé par Microsoft pour son système d'exploitation. Il possède une interface graphique intégrée.

On ne s'intéressera essentiellement dans la suite qu'aux débogueurs en source ouverte et libre.

Fonctionnalités avancées des débogueurs

Débogage bidirectionnel

L'un des problèmes souvent rencontrés lors du débogage d'un programme est l'incapacité de revenir en arrière dans le programme. En effet, si on observe un problème, retrouver la fonction ou l'instruction responsable implique souvent de devoir redémarrer le programme en positionnant divers points d'arrêt jusqu'à trouver l'origine réelle du programme. Cependant, le problème peut ne pas se manifester à nouveau, si par exemple la mémoire est allouée différemment par le noyau, si les fils d'exécution sont ordonnancés différemment ou encore si la date ou la durée d'exécution influent sur le programme.

Une solution à ce problème est d'enregistrer toutes les entrées du programme, l'ordonnancement des fils d'exécution et les réponses des appels système, qui pourront ensuite être redonnées au programme pour garantir une exécution identique [60]. On appelle ceci le débogage Enregistrement Répétition (Record Replay debugging). Cette approche est assez peu coûteuse en performances et en mémoire lors de l'avancée car on n'enregistre que peu de données. Cependant, chaque pas en arrière implique de recommencer l'intégralité du programme jusqu'à ce point. Elle n'est donc pas très efficace pour du débogage inversé à petite échelle.

Une seconde solution est d'enregistrer pour chaque instruction les différences de valeurs des registres et de la mémoire. Pour revenir en arrière, il suffit donc pour chaque instruction d'inverser cette différence, un peu à la manière d'un logiciel de contrôle de source pour du code. Bien que plus efficace pour revenir en arrière à petite échelle, cette technique a l'inconvénient de nécessiter plus de mémoire et plus de calcul pour enregistrer toutes les données. Elle a été implémentée dans GDB à partir de la version 7.0 en 2009. Le surcoût de GDB est néanmoins énorme car il arrête le programme entre chaque instruction pour enregistrer les différences, produisant un ralentissement d'environ 1000 fois [17].

Il existe également une approche hybride qui consiste à utiliser des points de sauvegarde de l'état du programme. A partir de ces points de sauvegarde, le débogueur enregistre les différents facteurs extérieurs au programme pour pouvoir reproduire son exécution. On évite ainsi de devoir revenir en début de programme à chaque retour en arrière, et on limite également l'impact d'enregistrement de l'état puisqu'on peut ajuster la fréquence de cet enregistrement. Cette approche est nommée débogage bidirectionnel par reconstruction [61]. Cette approche a été implémentée dans plusieurs systèmes, notamment `rr` [62]. Ce logiciel libre maintenu par Mozilla enregistre l'exécution d'un programme et permet son exécution bidirectionnelle a posteriori dans GDB. Cet outil ralentit l'exécution d'environ 20% pour

17. `code.google.com/archive/p/rebranch/`

une application à un fil, mais ce coût peut augmenter sur les programmes à plusieurs fils d'exécution car ceux-ci sont sérialisés.

LLDB n'incorpore pas de débogage bidirectionnel.

Débogage non-stop

Lors du débogage d'une application à plusieurs fils d'exécution, un des désavantages des débogueurs est le fait que le débogueur va généralement interrompre l'exécution du programme complet dès qu'un fil d'exécution rencontre un point d'arrêt. Dans des systèmes en temps réel avec de nombreux fils, cette approche peut être gênante car elle rend le débogage très intrusif. GDB a introduit en 2008 un système appelé débogage *non-stop* [63]. Le principe est de n'arrêter que le fil impliqué dans le point d'arrêt, laissant les autres s'exécuter normalement pendant le traitement du point d'arrêt.

Modification du flux d'exécution d'un programme

Pendant le débogage, l'usager peut vouloir modifier le flux d'exécution d'un programme, c'est à dire modifier la façon dont le programme va se comporter et quelles fonctions vont être appelées. Cela peut être utile pour tester une modification mineure sans avoir à recompiler l'intégralité du programme. Dans le cas le plus simple, modifier la mémoire du programme avant un branchement (clause "if" par exemple) pourra suffire à modifier le comportement du programme. Une autre possibilité est d'écrire dans le registre `pc` l'adresse de la fonction que le programmeur souhaite voir le programme exécuter. En utilisant la plateforme d'instrumentation PIN [28], H. Pan et al. [64] ont également mis en évidence un outil pour modifier l'exécution d'un programme, en répétant ou en sautant certaines portions.

Cependant, ceci ne couvre que les cas où le code que nous souhaitons exécuter se trouve déjà dans le binaire compilé. Avec la version 7.9, GDB a ajouté une fonctionnalité de compilation et d'exécution dynamique [65]. Cette fonction permet d'ajouter dynamiquement du code dans le programme, code qui a accès en lecture et en écriture aux variables du programme débogué. Cette commande fonctionne en interagissant avec le compilateur GCC [17]. Cette commande permet d'ajouter une fonction au programme, mais celle-ci ne sera exécutée qu'une seule fois. En effet, une fois la fonction exécutée, GDB supprime tous les fichiers liés. Ceci n'est donc pas adapté à la modification d'une partie du programme exécutée plusieurs fois, puisque la fonction additionnelle doit être recompilée à chaque exécution.

Une autre possibilité pour modifier du code déjà compilé est de recompiler dynamiquement la ou les fonctions que l'on souhaite modifier. Cette approche est disponible pour le langage Java sous le nom de "Hotswap" [66], qui permet de recharger le code d'une classe qui a été modifiée pendant la session de débogage.

Traçage

Certains débogueurs peuvent également servir d'outils de traçage dynamique. GDB possède un module de traçage utilisant un système de trampolines pour ne pas avoir à effectuer un changement de contexte vers le noyau pour chaque point de trace [26].

Interfaces de débogage

Débogage en ligne de commande

La plupart des débogueurs, a l'instar de GDB et LLDB, ont une interface native en ligne de commande, c'est à dire dans un terminal. L'usager a uniquement accès à la sortie des commandes qu'il entre.

En complément de cette interface simple, GDB propose une interface plus avancée en utilisant la librairie Ncurses [67] qui permet de séparer le terminal en plusieurs parties pouvant afficher diverses informations telles que le fichier source, le code désassemblé ou les valeurs des registres.

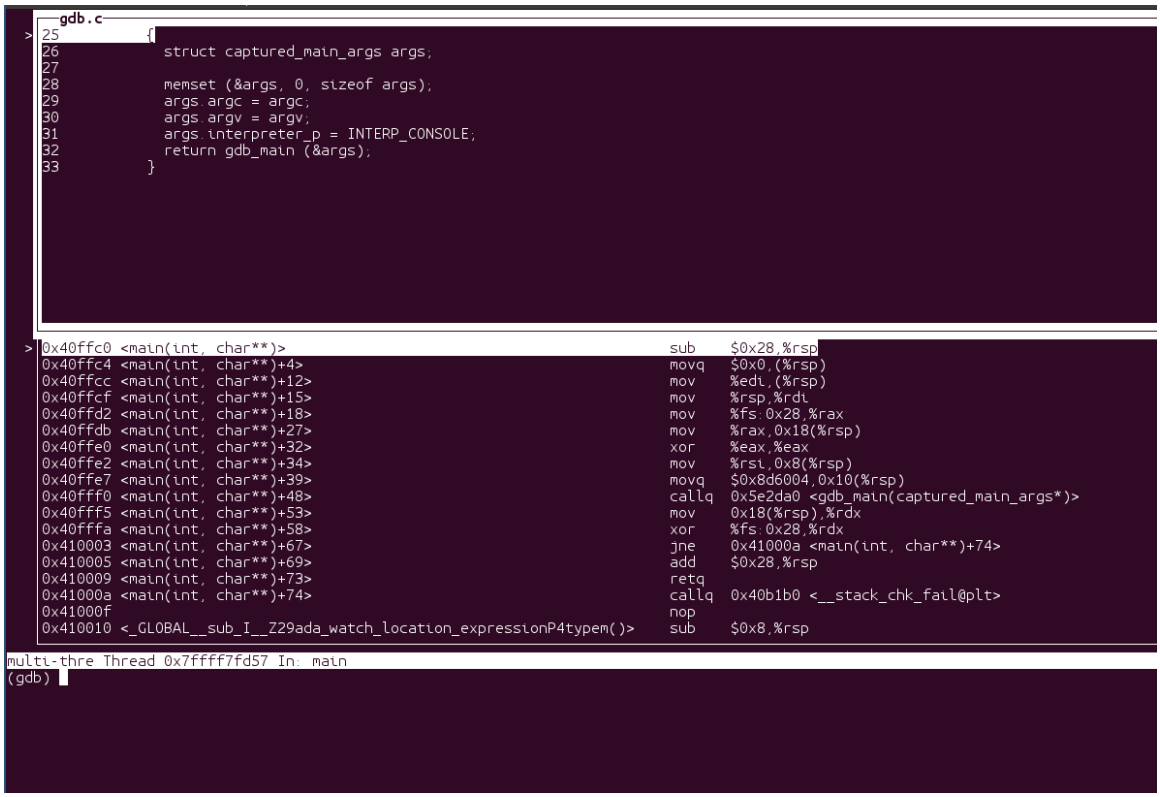

Figure 2.6 Affichage du code source et du code désassemblé en ligne de commande dans GDB

Débogage avec interface graphique

Certains débogueurs proposent une interface graphique de débogage de façon native (WinDbg) mais dans le cas de GDB et LLDB, ces interfaces graphiques sont prises en charge par des logiciels spécialisés comme DDD [68] pour GDB, ou par les environnements de développement (Eclipse CDT, Visual Studio Code ...).

Le dialogue avec ces logiciels frontaux se fait dans le cas de GDB et LLDB par une interface appelée Interface Machine (MI). Cette interface texte permet de standardiser les échanges entre GDB et le logiciel frontal, notamment en fournissant à chaque commande une réponse sous un format standardisé.

Débogage scripté

Pour les tâches de débogage répétitives ou nécessitant une liberté plus grande de l'utilisateur, LLDB et GDB fournissent également deux interfaces de script.

La première utilise les commandes classiques, mais permet de les agencer avec des boucles et des tests pour créer de nouvelles commandes [69].

La seconde interface est un interpréteur Python intégré, ainsi qu'un module python spécifique qui permet d'interagir avec le débogueur. Ce mécanisme permet d'ajouter des commandes de débogage, et est le seul moyen d'ajouter une extension au débogueur sans le recompiler.

Instrumentation par le débogueur

Des outils utilisant le débogueur comme plateforme d'instrumentation existent, notamment Verde, développé par R. Jakse [70]. Cet outil offre une plateforme d'instrumentation basée sur GDB qui permet de définir des scénarios d'exécution. Ces scénarios consistent en une machine à états dont les transitions sont dépendantes de certains évènements définis dans le scénario. L'automate est affiché et son état est mis à jour en temps réel dans une fenêtre graphique séparée du débogueur, permettant à l'usager de suivre le déroulement du programme. Par exemple, un utilisateur peut créer un évènement sur les méthodes *push* et *pop* d'une pile. Quand une des méthodes est appelée, Verde fait évoluer l'automate pour refléter le changement de taille de pile. Si la pile dépasse la longueur maximale, l'automate atteint un état non-acceptant et l'exécution est arrêtée pour présenter le problème à l'utilisateur. Un problème majeur de cet outil est la performance. En effet, l'outil utilise des points d'arrêt pour instrumenter le code du programme. Chaque évènement produit donc une interruption qui doit être gérée par le noyau et le débogueur. Il y a donc un délai d'environ 0,5 à 1 ms pour chaque évènement. Dans le cas où ces derniers sont fréquents, cela peut ralentir énormément le programme.

Un autre outil basé sur GDB est *PIN-augmented GDB* ou PGDB [71]. Chatterjee et al. ont conçu un système où GDB et PIN communiquent via le protocole distant de GDB (utilisé

notamment pour communiquer avec `gdbserver`) pour ajouter à GDB un mécanisme de détection de concurrence et d'interblocage dans les applications à multiples fils d'exécution. L'outil créé ainsi est très spécifique au problème ciblé, mais l'approche utilisée est reproductible dans d'autres scénarios. Cependant, l'utilisation de PIN implique une perte de performance même en l'absence d'instrumentation.

Q.Zhao et al. ont développé un autre outil basé sur GDB pour l'instrumentation nommé EDDI (*Efficient Debugging using Dynamic Instrumentation*) [72]. Cet outil utilise la plateforme d'instrumentation DynamoRIO [18] qui fonctionne en copiant le code vers un tampon avant de l'instrumenter et l'exécuter. Il présente à l'usager une interface communiquant en arrière-plan avec le débogueur et la plateforme d'instrumentation. Une application présentée dans le papier est l'insertion de points d'observation logiciel rapides par l'instrumentation de chaque accès mémoire. Le ralentissement moyen observé sur le banc d'essai SPEC2000 est de 2,5 fois en n'installant pas de point d'observation, et de 3,5 fois en inspectant toutes les adresses mémoire.

2.4.2 Débogage algorithmique

Une autre approche au débogage est d'automatiser la recherche d'hypothèses pour expliquer le comportement fautif d'un programme. C'est le principe du débogage algorithmique [73]. Par une série de questions à un oracle (généralement le programmeur), le débogueur est en mesure de déterminer la partie fautive du programme et de la pointer au programmeur sans que celui-ci n'ait à inspecter le code.

Fonctionnement

Un débogueur algorithmique fonctionne en construisant un arbre d'exécution à partir du code source et des entrées. Chaque nœud de cet arbre contient un prédicat vérifié dans l'exécution du programme. Par exemple, il peut s'agir du résultat d'une fonction à partir d'une certaine entrée. Les nœuds fils correspondent aux prédicats qui ont permis d'établir le résultat obtenu dans le nœud père. Par exemple, il peut s'agir des appels de fonctions faits par la fonction père pour obtenir son résultat. A la racine de l'arbre on retrouve donc le résultat final. Si celui-ci ne correspond pas à ce qui est attendu par l'oracle, le débogueur algorithmique va parcourir l'arbre en présentant pour chaque nœud le prédicat à l'oracle qui doit déterminer s'il est correct ou non. En élaguant les branches correctes, on finit par retrouver la feuille fautive, c'est à dire le nœud de plus bas niveau où un problème apparaît. Il suffit alors de

18. `dynamorio.org`

faire correspondre cette feuille avec une ligne de code, qu'il faut donc corriger.

Limitations et solutions

Bogues multiples Cette méthode ne permet d'identifier qu'un problème à la fois, il faut donc la répéter tant que le résultat n'est pas celui attendu. Ceci dit, la plupart des débogueurs algorithmiques gardent en mémoire les réponses précédentes pour éviter de répéter les questions inutilement.

Parcours de l'arbre Une fois l'arbre construit par le débogueur, ce dernier doit déterminer quelles questions poser et dans quel ordre pour minimiser le temps total de débogage.

La première technique proposée pour parcourir l'arbre [74] était de commencer par les feuilles de l'arbre, et de s'intéresser au nœud père si tous ses fils étaient corrects. Le premier nœud identifié comme problématique est donc directement le nœud responsable de la faute. Cependant, cette approche peut être particulièrement longue puisque l'on élimine les nœuds un par un. De plus, les questions posées à la suite sont potentiellement non corrélées et il peut être difficile pour le programmeur d'y répondre rapidement.

Une seconde approche plus naturelle a été proposée par Av-Ron [75]. En partant de la racine, le débogueur descend dans la branche incorrecte jusqu'à trouver le nœud fautif. Cependant, rien ne garantit ici un ordre dans les questions sur les nœuds fils d'un nœud problématique. L'oracle peut perdre du temps si les questions ne sont pas corrélées correctement, ou si le débogueur pose des questions sur des nœuds ayant peu de chance d'être fautifs (peu de nœuds dans leur sous-arbre induit par exemple).

Plusieurs autres méthodes ont été développées, notamment une méthode nommée "Divide and Query" proposée également par E. Shapiro [74] qui choisit le nœud à proposer à l'oracle en fonction du poids du sous-arbre dont il est la racine, le but étant de diviser par deux ce poids à chaque question pour garantir une complexité logarithmique vis à vis du nombre de nœuds de l'arbre.

En termes de performance, selon l'article de J. Silva [73], l'algorithme le plus performant est une variante de cette dernière méthode "Divide and Query". L'approche de bas en haut basique de Av-ron est environ 1,5 fois plus lente. Le premier algorithme de parcours de l'arbre est quant à lui 3.8 fois plus lent.

Débogueurs algorithmiques disponibles

Bien que prometteuse, cette forme de débogage est encore essentiellement réservée à un usage académique. Il existe ceci dit plusieurs débogueurs algorithmiques pour divers langages :

— Buddha [76] : Ce débogueur utilise une approche paresseuse dans la construction de l'arbre d'exécution afin de ne construire que les nœuds nécessaire. Il fonctionne sur des programmes écrit en Haskell.

— Declarative Debugging for Java [77] : Cet outil combine l'algorithme "Divide and Query" avec d'autres techniques comme la fusion de nœuds pour accélérer la découverte du problème source sur des programmes écrits en Java.

2.4.3 Débogage avancé

L'usage de débogueurs interactifs ou algorithmiques repose sur l'observation d'un problème, et sur la possibilité de sa reproduction dans l'environnement contrôlé du débogage. Cependant, la nature exceptionnelle de certains problèmes et l'intrusivité du débogage rendent parfois cette tâche complexe. Une corruption de données ou un interblocage peut n'avoir lieu que lors d'un ordonnancement particulier des fils d'exécutions, et les débogueurs ne permettent pas de manipuler facilement l'ordonnancement. Additionnellement, il est très intéressant de détecter les problèmes avant de publier le logiciel, et une infrastructure de test ne peut être exhaustive. Des outils permettant l'automatisation de la détection de concurrence entre fils d'exécution et d'analyse de mémoire sont donc nécessaires en complément des débogueurs classiques.

Analyse de fuite de mémoire

Gestion de la mémoire

La plupart des langages autorisent l'allocation dynamique de mémoire, c'est à dire que le programme va pouvoir demander au système un espace mémoire dont il va pouvoir disposer librement. En C par exemple, une fonction responsable de cette allocation est `malloc`. Une fois que le programme n'a plus besoin de cet espace, il doit le libérer afin de ne pas surcharger la mémoire du système inutilement. Dans des langages comme Java ou Python, supprimer les pointeurs vers ces zones mémoire suffit. En effet, ces langages sont munis d'un système de récupération de mémoire [78] [79]. En C++, un système similaire existe : les pointeurs uniques [19], dont la cible est libérée à la suppression. Cependant, dans le cas général en C et en C++, c'est à l'usager de gérer sa mémoire. Pour chaque appel à `malloc`, il doit donc y avoir

19. `cppreference.com/w/cpp/memory/unique_ptr`

un appel à `free`, sans quoi le programme alloue de la mémoire qui est perdue. Cette mémoire est réservée par le système d'exploitation et peut s'accumuler jusqu'à ce que le programme ou le système entier soient affectés.

Pour éviter ces problèmes de fuites de mémoire en C et C++, des systèmes de récupération de mémoire existent, notamment le *ramasse-miette Boehm-Demers-Weiser* [80]. Le principe de cette librairie est d'analyser régulièrement l'espace mémoire du programme pour déterminer quelles zones sont accessibles ou non depuis les variables actives. Les zones non accessibles sont alors libérées. Cependant, l'ajout d'un système de récupération ajoute un coût en complexité et en taille au programme [81]. De plus, ce ramasse miette est conservateur, c'est à dire qu'il considère toute suite de bits en mémoire comme une adresse mémoire potentielle. Certaines zones mémoires non utilisées peuvent donc toujours être considérées comme actives et non libérées. L'utilisation de ce ramasse miettes est par conséquent rare en pratique.

Analyse statique

L'analyse statique d'un programme a pour but de déterminer les fuites de mémoires possibles du programme sans l'exécuter. Les avantages d'une telle approche sont multiples. D'une part, l'analyse statique peut se faire tôt dans l'écriture d'un programme et ne nécessite pas que toutes les parties du programme soient fonctionnelles. Les problèmes peuvent donc être corrigés tôt dans le développement. Elle permet également de tester exhaustivement le code, y compris des portions peu utilisées en pratique ou absentes d'un ensemble de tests.

L'analyse statique du code peut se faire au niveau du compilateur, ainsi Clang [82] intègre des outils d'analyse statique [83]. En reconstruisant les flux d'exécutions possibles du programme, le compilateur peut suivre le nombre de références à une adresse mémoire et analyser ainsi s'il y a possibilité de fuite, d'usage après désallocation ou de double désallocation.

D'autres outils plus avancés existent, qui se basent sur une analyse des flux de valeurs [84] [85]. Ces approches construisent un graphe liant les allocations aux désallocations de mémoire, en ajoutant aux arêtes entre les nœuds des conditions correspondant au flux d'exécution du programme. Par l'analyse du graphe, ils déterminent alors s'il existe un ensemble de conditions faisant qu'une zone mémoire ne soit jamais désallouée, ou bien produisant deux désallocations de la même zone.

Un inconvénient de l'approche statique est le nombre de faux positifs qui peuvent être produits. Un programmeur doit manuellement vérifier si une fuite déclarée est effectivement possible ou non. Pour les trois outils mentionnés ici, les taux de faux positifs sont de respectivement 25% (Clang [82]), 14% (FastCheck [84]) et 19% (Saber [85]) sur le benchmark SPEC2000 [85].

Analyse dynamique

Une autre possibilité souvent complémentaire pour l'analyse d'un programme est l'analyse dynamique. Cette analyse prend place pendant l'exécution d'un programme, soit pendant une série de tests, soit en production. Contrairement à l'approche statique, seuls sont analysés ici les chemins d'exécution effectivement empruntés. Pour ce qui est de la charge sur le programmeur, l'analyse dynamique produit généralement moins de faux positifs mais l'origine réelle d'une fuite peut être difficile à trouver dans le code.

Il existe deux types d'outils dynamiques. D'une part les outils que l'on qualifiera de totalement dynamiques, qui peuvent agir sur n'importe quel binaire compilé, d'autre part les outils hybrides, qui nécessitent d'instrumenter le code à la compilation et donc ne peuvent pas analyser un binaire compilé indépendamment de l'outil. Pour l'analyse de fuite de mémoire, les outils sont en général totalement dynamiques car ils peuvent généralement se contenter d'instrumenter uniquement les fonctions d'allocation de mémoire (`malloc`, `free`...).

Dans la première catégorie on peut trouver des outils comme Leak Sanitizer [86], l'outil d'analyse de fuite mémoire d'Address Sanitizer, un outil en source libre maintenu par Google. Contrairement à Address Sanitizer, Leak Sanitizer ne nécessite pas d'instrumentation du code à la compilation. Il fonctionne via une librairie chargée dynamiquement qui remplace les fonctions d'allocation de mémoire de la librairie C pour enregistrer quelles portions de la mémoire sont allouées et désallouées. En fin d'exécution, le programme peut afficher des informations sur les portions qui n'ont pas été désallouées. L'avantage de cette approche est un coût en performance quasi nul excepté la vérification finale en sortie de programme [87]. Un autre outil plus complet est Memcheck [30], l'outil de vérification de mémoire de la suite Valgrind [29]. En recompilant et instrumentant le fichier binaire, Memcheck analyse entre autres les fuites de mémoire. Contrairement à dmalloc, Memcheck permet également d'analyser les fuites sur des systèmes d'allocation de mémoire autres que malloc [20]. Il permet également d'utiliser les informations de débogage d'un programme pour identifier les fichiers sources et lignes correspondant à une fuite. Cependant, cet outil a un très fort impact sur la performance, avec une exécution typiquement 20 à 30 fois plus lente [21]. Dans la suite Valgrind, on trouve également l'outil Massif [88], qui a pour but d'établir des statistiques sur l'usage de la mémoire par le programme cible. Il comprend également un outil d'analyse de fuite, qui permet de trouver des fuites de mémoire non analysées par Memcheck [88]. Un outil similaire existe, nommé HeapTrack [89] et maintenu par KDE. Il se veut une alternative plus rapide à Massif car il ne recompile pas toutes les instructions du programme. Cependant,

20. `valgrind.org/docs/manual/mc-manual.html#mc-manual.mempools`
21. `valgrind.org/docs/manual/quick-start.html#quick-start.mcrun`

contrairement aux outils de Valgrind il ne peut pas analyser la mémoire allouée hors du système classique de la librairie C.

Pour tous ces outils, l'accès aux informations de débogage est nécessaire pour pouvoir pointer correctement vers la ligne du fichier source correspondant à l'erreur.

Analyse de corruption de mémoire

Une analyse qui va souvent de pair avec l'analyse de fuite de mémoire est l'analyse de corruption de mémoire. On parle de corruption de mémoire quand une lecture ou une écriture en mémoire se fait dans une région ne correspondant pas à celle initialement visée, ou dans une région invalide. Les problèmes les plus fréquents de corruption de mémoire proviennent de dépassement de tampons ou d'usage de mémoire après désallocation. Dans certains cas, une manipulation incorrecte de pointeurs ou la lecture d'une valeur non initialisée peut aussi entraîner des accès invalides. Ces problèmes sont aussi la source principale de vulnérabilités dans un programme. En effet, près de deux tiers des attaques se basaient dessus en 2000-2003 [90].

Les analyses de fuite et de corruption étant proches, on les retrouve souvent dans les mêmes outils. Ainsi, MemCheck détecte également un grand nombre de types de corruption de mémoire. Un autre outil complètement dynamique est Electric Fence [91]. Cet outil fonctionne en ajoutant autour de chaque région allouée par la librairie C deux pages protégées en lecture et en écriture. Si un programme cherche à accéder à de la mémoire hors de la zone allouée, par exemple lors d'un dépassement de tampon, le système soulève une exception qu'Electric Fence analyse pour déterminer d'où provient la faute. Electric Fence protège également les pages une fois que la mémoire a été désallouée, afin de détecter les cas d'usages après désallocation. Les limitations de cet outil sont d'une part sa consommation de mémoire virtuelle (3 pages par allocation) qui peut être problématique sur des architectures 32 bits, d'autre part le fait que la protection ne peut se faire qu'au niveau d'une page, et donc ne détecte pas forcément les dépassements de quelques octets. Une autre approche pour analyser de façon dynamique et plus exhaustive les fuites de mémoire est Data Watch [92], un outil propriétaire développé par Ciena. Là encore, l'outil remplace les fonctions d'allocation de la libraire C. Au lieu de retourner directement une adresse correcte, la fonction renvoie une adresse teintée qui pointe vers de la mémoire protégée. A chaque accès à cette adresse invalide, le noyau va soulever une erreur de segmentation que l'outil va attraper. En analysant l'adresse teintée qui a été déréférencée, l'outil détermine l'adresse initiale et le décalage de l'accès mémoire. Si cet accès est dans les limites allouées initialement, alors la valeur correcte est lue ou écrite. Sinon, une exception est soulevée qui pointe vers l'instruction responsable de l'erreur. Cette

approche permet de déterminer précisément la position d'un dépassement de tampon dès le premier octet, mais elle vient avec un coût en performance. En effet, chaque déréférencement d'un pointeur provoque une erreur de segmentation qui doit être gérée par le noyau. De plus, l'implantation d'un module dans le noyau est une approche intrusive qui limite son utilisation généralisée.

Address Sanitizer [18], en plus de son module d'analyse de fuites de mémoire, permet également l'analyse de plusieurs problèmes de corruption. Cet outil n'est pas totalement dynamique, et requiert une instrumentation du code au moment de la compilation. L'outil est divisé en deux portions : d'une part l'instrumentation, intégrée au compilateurs Clang [82] et GCC [17], d'autre part une librairie chargée dynamiquement à l'exécution du programme. Cette librairie va ajouter pour chaque zone mémoire allouée une zone mémoire "fantôme", où chaque octet fait référence à 8 octets de la mémoire originale. Un octet "fantôme" contient une valeur entre 0 et 8 correspondant au nombre d'octets alloués sur les 8 référencés (les zones allouées étant alignées sur 8 octets, un octet ne peut être accessible si l'octet précédent ne l'est pas). Au moment de la compilation, une vérification est insérée au niveau de chaque accès mémoire, pour vérifier que l'accès ne recouvre pas un octet invalide. Ceci est fait simplement en comparant la taille de l'accès et la valeur de l'octet "fantôme" correspondant. En gardant les zones désallouées en quarantaine, cet outil permet également de repérer les usages après désallocation. Contrairement à Electric Fence et Data Watch, Address Sanitizer peut analyser les accès mémoire à la pile. En termes de performances, Address Sanitizer ajoute environ 100% de surcoût en calcul, et 200% de surcoût en mémoire [18], ce qui peut le rendre difficilement utilisable sur des systèmes embarqués ou en temps réel.

Analyse de concurrence

Une source de problèmes difficilement reproductibles lors de l'analyse d'un programme à plusieurs fils d'exécution est la présence de compétition entre les fils et d'interblocages. Une compétition apparaît quand plusieurs fils accèdent à la même adresse mémoire sans synchronisation et qu'au moins un des accès est en écriture. Un interblocage survient lorsque plusieurs fils sont tous en attente de ressources retenues par les autres. Cela a pour effet de bloquer l'exécution de ces fils et donc en général du programme complet.

Thread Sanitizer [93] est un outil en source libre maintenu par Google qui permet d'analyser la présence de compétition entre les fils d'exécution. Comme Address Sanitizer [18], Thread Sanitizer est un outil dynamique hybride, qui nécessite une instrumentation à la compilation. En effet, cet outil instrumente chaque accès mémoire pour reconstruire un graphe des accès en lecture et écriture et des synchronisations possibles. Si une situation de compétition est dé-

tectée, le programme est interrompu et l'usager notifié de l'adresse mémoire, des fils impliqués et de leur pile d'appel. Thread Sanitizer permet également la détection d'autres problèmes de concurrence notamment l'usage d'une variable potentiellement libérée par un autre fil. L'outil propose également une version expérimentale de détecteur d'interblocages potentiels, qui fonctionne en construisant un graphe des verrous acquis par les fils [94]. Thread Sanitizer a un coût en performance considérable, multipliant de 2 à 30 fois le temps et jusqu'à 10 fois la mémoire utilisée par le programme initial [93] et n'est donc pas utilisable en production.

Helgrind [95] est un module de la suite Valgrind destiné à la détection de problèmes de concurrence. Il permet la détection d'un très large éventail de problèmes, notamment des compétitions entre fils et des interblocages potentiels. Comme Thread Sanitizer, Helgrind est très coûteux en performance et peut augmenter jusqu'à 100 fois le temps d'exécution du programme ciblé [95] [93].

2.5 Conclusion de la revue de littérature

Nous avons vu que certains domaines comme le traçage et le débogage avancé proposent de nombreuses fonctionnalités très utiles au programmeur mais leur cadre d'emploi est souvent complexe. Les outils dynamiques comme Valgrind ou Dtrace ont un fort impact sur la performance, alors que les outils plus performants comme LTTng ou Address Sanitizer nécessitent une instrumentation statique du code et donc une recompilation.

Un moyen d'allier la performance des outils statiques au confort des outils dynamiques est d'utiliser une plateforme d'instrumentation dynamique. Cependant, les plateformes existantes sont assez peu utilisées et ne proposent pas une interface naturelle à l'utilisateur. Une solution à ces deux problèmes est d'inclure un système d'instrumentation dynamique efficace à un débogueur. En effet, ces logiciels sont très répandus et ils possèdent déjà une interface naturelle via les environnements de développement modulaire et le Debug Adapter Protocol.

CHAPITRE 3 MÉTHODOLOGIE

A des fins de reproductibilité et de mise en contexte des résultats, nous présentons dans cette section le matériel ainsi que les différents logiciels utilisés.

3.1 Matériel

Les mesures de performance ont été effectuées sur un ordinateur personnel dont la configuration est présentée au tableau 3.1

Tableau 3.1 Configuration matérielle

Processeur	i7-4790 - 4 coeurs à 3.6-4.0 GHz
Mémoire	32 GB DDR3 1600MHz
Disque	512 GB SSD

3.2 Logiciel

3.2.1 Système d'exploitation

Le système d'exploitation installé sur notre machine pour les mesures était Linux, avec la distribution Ubuntu [1] 18.04LTS. La version du noyau utilisée est 4.15.0-88.

3.2.2 Logiciels utilisés

Un résumé des logiciels utilisés et de leur version est présenté au tableau 3.2

Tableau 3.2 Logiciels utilisés

Fonction	Logiciel	Version
Compilateur	GCC	7.4.0
Débogueur	GDB	9.0.50 [2]
Mesures	GNU time	1.7.0
Analyse de mémoire	Address Sanitizer	4
Analyse de mémoire	Valgrind Memcheck	3.13.0

1. ubuntu.com
2. Nous avons modifié ce logiciel à partir de cette version

3.3 Code source

Le code source des différentes modifications apportées à GDB et des extensions pour Visual Studio Code est stocké sur le site GitHub à l'adresse `github.com/paul-naert`.

Le code source de GDB incluant les modifications se trouve dans le répertoire `binutils-gdb`. La branche *master* contient une version stable de la fonctionnalité d'instrumentation mais seules les instructions de 5 octets ou plus peuvent être instrumentées. La branche *short-insn-patching* contient le code permettant d'instrumenter les instructions courtes. La branche *fast-data-watch* contient une version antérieure du code et notamment une fonction spécialisée pour l'utilisation de Fast Data Watch.

Le code des extensions de Visual Studio code est dans le répertoire VSCodeTrace. L'application de traçage dynamique est sur la branche *master* tandis que l'application généraliste modulaire se trouve sur la branche *modular-app*. Des exemples sont présents dans le dossier `example`.

3.4 Article scientifique

Le chapitre suivant est un article de recherche soumis au Elsevier Journal of System Architecture, présentant notre solution ainsi qu'une évaluation de ses performances.

Il met en valeur les principales contributions de notre recherche :

- **Une nouvelle architecture pour le développement d'outils de vérification dynamique** offrant aux outils un fonctionnement entièrement dynamique et ciblé, ainsi qu'une interface commune au travers d'un EDI modulaire.
- **Une plateforme d'instrumentation dynamique performante** intégrée au débogueur GDB, permettant de modifier librement le comportement d'un programme après sa compilation, notamment pour en vérifier la correction.

CHAPITRE 4 ARTICLE 1 : INTERACTIVE AND TARGETED RUNTIME VERIFICATION USING A DEBUGGER-BASED ARCHITECTURE

Soumis au : *Elsevier Journal of System Architecture*.

Authors

Naert, Paul <paul.naert@polymtl.ca>

Azhari, Seyed Vahid <vazhari@ciena.com>

Dagenais, Michel <michel.dagenais@polymtl.ca>

4.1 Abstract

Runtime verification of software (RV) often relies on two categories of tools: dynamic heavy-weight tools, which significantly impact performance, and lighter and more efficient but static tools, which require recompiling the binary. In this paper we propose a new framework for building efficient and targeted dynamic RV tools, bridging the gap between those two categories. This framework is separated into two domains: source and binary. On the source level, a modular development environment provides a custom user interface which allows for precise targeting of instrumentation, as well as advanced interactivity. The binary level revolves around a debugger, which controls binary manipulation and library loading. In order to create fully dynamic tools, we added new instrumentation capabilities to the GNU debugger, using trampoline-based probes to inject code in the binary efficiently and interactively. Our framework focuses on accessibility for users via the graphical interface, and for developers by making it easy to adapt existing tools and by relying on popular programming languages such as Python and C++.

As a demonstration of our framework capabilities, we provide a significantly faster implementation of conditional breakpoints for GDB, as well as targeted and fully dynamic versions of two state-of-the-art runtime verification tools: Address Sanitizer and Data Watch.

4.2 Introduction

Runtime Verification (RV) consists in monitoring a program behavior at runtime to detect various forms of incorrect behavior. This includes checking for memory corruption and leaks, as well as thread deadlocks and races. RV consists of three phases than can take place sequentially or simultaneously:

— Instrumentation
— Data collection
— Analysis

Instrumentation is the act of modifying a program from its original design, for instance by adding logging or tracing to monitor the behavior and performance of the program. Instrumentation can take place at most steps of a program life cycle: in the source code, at compile time and in the generated binary – either before or during the execution. This last form of instrumentation –Dynamic Binary Instrumentation (DBI)– offers the most flexibility to the user, as there is no need to recompile or even have access to the source code. One of the most common functionalities of DBI is the injection of *probes*. These functions are attached to a specific instruction and are called each time it is executed. For instance, they can be used to trace or modify program variables. These probes can be injected in different ways. Trap-based probes use system interrupts to suspend the execution of a program while the probe function is run, then hand control back once it succeeded. While this approach offers great flexibility, the switching back and forth between kernel and user space incurs a significant slowdown in the execution of the program. An alternative is the use of trampoline-based probes, which load the probe function into the program space and replace the instrumented instruction with a jump to this location. This approach makes instrumentation significantly more efficient, but jump instructions cannot be inserted anywhere arbitrarily, making it less flexible than trap-based instrumentation.

Data collection and analysis are tool-dependent and are usually handled by a specific library. Calls to the functions of those libraries are generally added by instrumentation.

One key attribute of RV tools is their performance. In order to catch even the rarest bugs in a program, RV needs to be enabled in production or at least during extensive testing, which means that the overhead of the RV tool needs to be small enough not to impact the program usability. In some cases, such as embedded real-time systems, free memory and computation power margins are minimal, which means that very few tools can be used.

In this article we introduce a new interactive RV architecture based on the GNU Debugger (GDB). It relies on a new instrumentation framework using special trampoline-based probes, combined with the debugger capabilities which allow it to dynamically load libraries into the inferior program. This framework focuses on interactivity and accessibility: the instrumentation code can be written in C or C++ and the whole process can be managed through Python scripting and an IDE interface.

This paper provides several contributions to the domain of runtime verification:

— A new RV framework focusing on flexibility, precise targeting of instrumentation and

performance.
— New efficient instrumentation capabilities for GDB, relying on signal-free trampoline-based probes.
— Several modules for the Visual Studio Code environment, which provide advanced interfaces for RV tools.
— Multiple improvements on existing RV tools built using our framework: GDB fast conditional breakpoints, Targeted Dynamic Address Sanitizer and Fast Data Watch.

In the next section we provide a brief overview of the literature on instrumentation and runtime verification. This is followed by section 4, which discusses the details of our debugger-based framework. In section 5 we present several tools modified using our framework, which demonstrate the different capabilities offered. Finally, in section 6, we provide a quantitative evaluation of both the instrumentation framework and the applications built using it.

4.3 Related Work

Dynamic Binary Instrumentation Several frameworks are available for Dynamic Binary Instrumentation, relying on different mechanisms to inject or replace code. A very basic approach is using the GDB scripting capabilities combined with breakpoints to instrument some instructions. However, this comes with a high performance impact and is unusable in most cases.

Pin [28] is an instrumentation framework for the x86 and ARM architectures that relies on recompiling the code dynamically, as it is executed, and injecting the different instrumentation probes at that time. It is very effective for thorough instrumentation, but it incurs a runtime overhead, even when no instrumentation is performed, because of the cost of dynamic recompilation.

Dyninst [22] uses trampolines to instrument a binary and can do it both dynamically at runtime and statically on the binary file. Each individual instrumentation requires heavyweight analysis of the binary, which means that it is most effective at bulk insertion of probes rather than on-demand, interactive instrumentation.

LiteInst [25] is a tool that uses instruction punning to insert trampoline-based probes in a program. This allows to avoid running large-scale analysis to insert a single probe. When the insertion of a jump instruction is not possible, it relies instead on trap-based instrumentation.

Runtime verification A large number of specialized RV tools exist to check for specific properties in a program, such as memory leaks and corruption or thread race conditions.

Popular tools for these tasks are Address Sanitizer [18] (memory analysis) and Thread Sanitizer [93] (race detection). These tools instrument programs at compile time and load a custom library at runtime. This library collects the information provided by the added instrumentation, and uses it to build data structures dynamically, which track the state of the program. When a specific property is met in the data structure, the tool generates an error to stop the program and generates a log detailing where it occurred. For instance, if at some point two nodes representing memory writes to the same address on different threads are not linked by any ordering edge, then Thread Sanitizer generates an error for Data Race. Another more generic approach is provided by the tool Valgrind [29] [96]. It is designed as a heavy-duty instrumentation framework, and offers modules such as Memcheck [30] for memory analysis and Helgrind [95] to detect synchronisation errors. Contrary to tools like Address Sanitizer, Valgrind tools are fully dynamic, i.e. they do not require recompiling the binary. Valgrind instruments the code by dynamically recompiling the program, as it executes, and running it in a virtual machine. It also maps shadow memory, to store metadata about the program memory. This recompilation and virtual execution have a significant performance impact, and Valgrind tools often slow the program by a factor of 10 to 100 [1] [2]. As for generic RV, Monitoring-Oriented Programming (MOP) [97] is a framework designed to check specifications against the implementation for a given program. The user provides logical assertions that MOP checks at runtime.

GDB as a framework The GNU Debugger (GDB) is a very popular debugger for C and C++ code. While its main use is interactive debugging through the use of breakpoints, it contains a large number of features that make it suitable as a framework for use with different goals, such as RV. Among those advanced features are reverse debugging, trampoline-based tracepoints, and non-stop debugging for multi-threaded programs. In version 7.9 (2015), GDB also added the *compile* command which allows the user to compile and execute a snippet of C/C++ code in the context of the inferior program, with full access to local and global variables. The debugger offers a scripting interface in both command line syntax and Python, and a machine interface which allows to communicate with other tools, such as Integrated Development Environments (IDE), notably through the Debug Adapter Protocol [3]. Tools such as GDB Extended Features [4] take advantage of this scripting capability to add functionality to GDB such as displaying the virtual memory mappings or compiling assembly code.

1. valgrind.org/docs/manual/quick-start.html#quick-start.mcrun
2. valgrind.org/docs/manual/hg-manual.html#hg-manual.todolist
3. microsoft.github.io/debug-adapter-protocol
4. https://gef.readthedocs.io/

Interactive Runtime Verification IRV was introduced by Jakse et al [98] along with a GDB-based framework called Verde. It allows the user to define scenarios and properties to generate a finite-state machine which monitors the execution of the program. The nodes of that automaton are program states, and the edges between those nodes are events. Whenever such an event is triggered, the automaton is updated accordingly and the graphical interface shows the new state. This type of verification can check for queue accesses validity for instance, updating the length of the queue on *push* and *pop*, and raising an error whenever the user pops an empty queue. However, one major issue of Verde is that it relies on GDB breakpoints to monitor events (i.e. trap-based instrumentation) which makes it perform poorly and slows down the program when an event is triggered frequently. Furthermore, this tool is not well suited to adapt and improve existing RV tools, as they need to be completely re-written to fit the Verde framework.

4.4 Debugger-based runtime verification

4.4.1 A versatile architecture for RV

Most Runtime Verification tools go for a lean approach, focusing on one specific task (memory analysis, thread contention analysis...) and doing that efficiently. However, in the absence of an efficient runtime verification framework, most tools are built independently of one another. All the basic features need to be re-implemented on an individual basis, generating development costs. For non-essential features, this cost may prove too important and these features may get dropped, even though other tools have already implemented them. Given the proper infrastructure, most of the code could be re-used or shared, and features such as attaching to a running program and dynamic instrumentation could see a more widespread adoption.

More versatile tools exist, such as Valgrind, but the machinery needed to provide that versatility comes with a very large performance cost, even when no analysis is performed. This makes these tools unfit for light targeted analysis of a specific problem.

Debuggers come up as a natural middle ground between both approaches. Indeed, they are generalist tools that already implement a large number of binary manipulation features, such as attaching to program and loading libraries. When no instrumentation is performed, they have very little impact on execution performance, and they already contain the infrastructure to process debug information and navigate binaries. Furthermore, most IDEs already implement a graphic interface for debuggers, which can significantly improve the usability of tools compared to the command line interface used by most RV tools. The most recent IDEs such

as Visual Studio Code [7] and Eclipse Theia [16] come with support for user designed modules, which can improve user experience by offering a custom interface for debugger-based tools.

One key missing feature in debuggers is the possibility to freely and efficiently instrument code. In order to demonstrate the versatility of debugger-based RV, we introduced new instrumentation capabilities to GDB. By extending the *compile* command of GDB, we were able to produce an instrumentation mechanism with a very intuitive interface for developers familiar with the debugger. To maximise efficiency, it is implemented using trampoline-based instrumentation and instruction punning. This feature is crucial in our approach, as most advanced RV tools use one form of instrumentation of the binary, and performance is key in the usability of a tool.

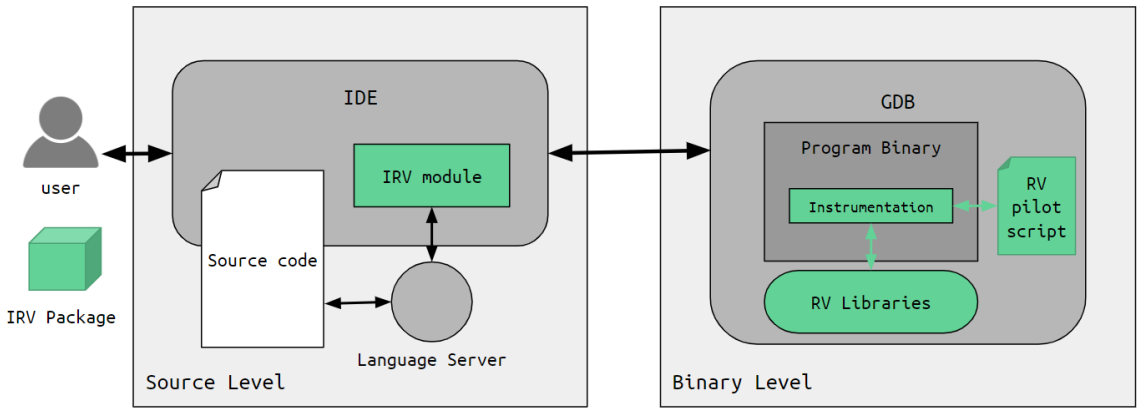

Figure 4.1 The proposed IRV framework architecture. Items in green are generally specific to a RV tool.

The new RV architecture we propose is shown in Figure 4.1. It is separated into two domains: source code and binary. At the source code level we find a modular IDE that can interact with a language server to process the code. The IDE is also responsible for user interaction and thus controls the binary level, which contains GDB and the program being debugged.

This framework makes it possible to add new capabilities to existing tools, without modifying the runtime library which is often the most critical part. It enables efficient dynamic instrumentation of binaries, as well as advanced user interactive features, such as precise targeting of functions and source files and letting the user adapt the instrumentation as the program runs. This opens up possibilities for designing truly interactive, precisely targeted high-performance RV tools, a functionality not offered by other frameworks.

4.4.2 Framework Interfaces

User interface As part of our interactive RV framework, we designed a versatile backbone application for VSCode which provides support for loading user-defined RV tools. We defined a standard format for such tools which uses a JSON file as reference. A RV tool package can include several libraries and Python scripts, which are called by custom commands in the IDE. The user is presented with a choice of commands, some of which use IDE information such as the source file currently open or the cursor position. These commands are forwarded by the IDE to the binary domain, which instruments and runs the program.

Programming interface The proposed instrumentation tool is integrated within the traditional GDB command line interface. It uses the keyword *patch* and a syntax similar to the *compile* command. The patch location can be specified using a function name, a source file line or an explicit memory address. Unlike other frameworks which require using a specific API to pilot the instrumentation, this tool uses a simple interface with which debugger users are already familiar. The instrumentation code to be injected consists in a regular C or C++ code snippet which is compiled with GNU's GCC compiler. If the program is compiled with debug information, the snippet can reference the global and local variables from the instrumented frame, without any declaration. These variables can be read and modified freely.

As part of the GNU debugger, the instrumentation framework introduced here can easily be manipulated using the GDB scripting capabilities. The integrated Python interpreter enables designing runtime verification tools which can instrument code at will. Ease of use is insured because a basic understanding of the Python language is sufficient to design a custom RV tool using GDB. Being programmed in Python, this tool can also use the plethora of libraries and modules available, making it easy to create advanced tools.

When creating a new tool or converting an existing one to our framework, developpers must mainly focus on two aspects: user commands and the GDB pilot script. Defining user commands consists in creating a clear designation for the user, and translating it into a command for the pilot script. It can incorporate IDE data such as a file path and line numbers. As for the pilot script, it is written in Python and is responsible for instrumenting the binary. It has access to the GDB API, which makes available a variety of GDB structures such as symbol tables and breakpoints

In order to provide advanced user interactivity, developpers can also create an IDE module specific to a RV tool, instead of using the versatile module that we designed. It can then take advantage of all the APIs of the modular IDE and provide custom interactions, such as

displaying data in a separate tab or directly annotating the source code. As an example, we designed a prototype dynamic tracing extension which lets the user place custom tracepoints in the code, for instance to monitor a specific variable. Extensions such as this one can also use the advanced source code processing offered by the IDE. Our demo application can ask a language server for all occurrences of a specific variable and then add a tracepoint to each location where it is modified, in order to trace when it is modified.

4.4.3 Efficient instrumentation in GDB

In this section, the implementation of the instrumentation framework that we integrated into GDB is described. Currently, the only functional implementation is for x86_64 Linux, but most of the code is architecture independent. As of March 2020, this work is not yet included into the main branch of GDB but is in the submission process.

Instruction punning We chose to implement the instrumentation framework using trampoline-based probes. They work by replacing instructions in the code by a jump instruction to a trampoline. The trampoline is a small chunk of memory allocated close enough to the instrumented instruction to be reachable by a jump. This trampoline contains all the instructions necessary to perform the various tasks surrounding the actual instrumentation, i.e. saving and restoring registers, executing the instruction that was replaced by the jump and calling the actual probe function with the proper arguments. Figure 4.2 illustrates how trampoline-based instrumentation works.

One significant issue with trampoline-based instrumentation on the x86_64 architecture is the length of the jump instructions. In order to be able to reach an unallocated area in memory large enough to put a trampoline, a 5 byte long jump instruction is often required. This complicates the instrumention of shorter instructions, which generally make up more than half of the instructions of a program [5]. Another possibility is relocate several instructions at once, until there is enough room to put a 5 byte jump. However, if the program contains a jump to the second replaced instruction, the processor will interpret the part of the jump offset as an instruction and the program will likely crash.

In the following paragraphs, we will describe the instruction layout at the address to instrument like this:

ABCDE

5. strchr.com/x86_machine_code_statistics

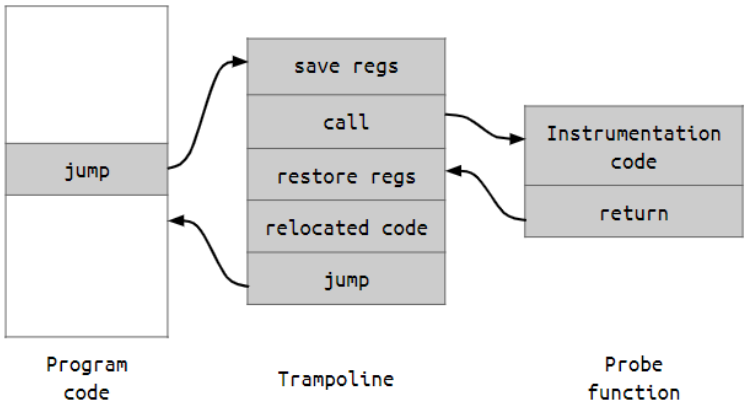

Figure 4.2 Trampoline-based instrumentation probe

where each letter represents a byte. Upper case letters correspond to the start of an instruction and lower case letters to the other bytes. For instance

AbcDe

corresponds to a 3 byte long instruction (Abc which we want to instrument) followed by an instruction at least 2 byte long, De. To illustrate the previous example, if we blindly replace instruction **A** by a jump (opcode **J**) to some free area of memory (offset **wxyz**), we obtain layout

Jwxyz

If at a later time, a thread jumps to the location of former instruction **De**, it will only read bytes **yz** which will be interpreted as an instruction unrelated to the original.

Our solution is an adaptation of the technique introduced by Chamith et al [25], which uses instruction punning to instrument the binary. This instrumentation method has the advantage of requiring few modifications to the memory in order to insert a probe. This has the advantage of providing fast instrumentation times, which is critical in an interactive framework.

Instruction punning is a practice used in code compression or obfuscation that takes advantage of the dual nature of instructions as data. Depending on where someone starts reading a set of bytes, the resulting instructions can be completely different. More specifically, when using instruction punning for instrumentation, we can choose where to put the trampoline according to the instructions following instruction **A**. In the case of layout

AbcDe

we can try and find a trampoline address with offset ****de*** from instruction **A**, where the bytes labeled ***** can be selected to provide more possibilities. If there are bytes **y** and **z** such that

Jyzde

is a jump to an area where a trampoline can be installed, then even if the program were to jump to instruction **D**, it would behave as if there had been no instrumentation.

However, this approach significantly limits the search space for a compatible trampoline address. In the previous example, for instance, the most significant bytes of the offset are set to **e** and **d** because the x86 architecture is little-endian. If **e** corresponds to a large negative value, it can render the offset completely invalid, regardless of the other three bytes.

To circumvent that limitation, it is possible to modify the instruction overwritten, as long as we can insure that the behavior will be the same, if a thread jumps to where instruction **D** is supposed to be. One instruction that can be placed there is a SIGTRAP generating instruction, such as INT3. Using a handler for that signal, we are able to redirect any thread that hits this trap to the end of the corresponding trampoline, so that it executes the instruction that was overwritten and jumps back to the program. Using illegal instructions, instead of only INT3, allows for even more flexibility in the choice of the jump offset, as they can be used more easily in place of the most significant byte. This is because some can be interpreted as positive integers (0x06, 0x07), while INT3 (0xCC) is a negative signed integer.

For instance, using the same example as earlier, we can replace instruction **De** with bytes ***ix*** where ***i*** can be chosen among 16 possibilities (**d**, INT3 or any of 14 1-byte illegal instructions) and ***x*** can be set freely. This approach generally incurs a much lower runtime overhead than trap-based instrumentation. Indeed, the signal raising instructions are only hit if there is a jump directly to them, i.e. if the jump instruction overlaps two basic blocks. If it is fully contained in a basic block, then no signal will be raised during execution.

With those constraints, there could still be no space available for a trampoline, which can happen in layouts such as:

AbcDE

which are very restrictive. In that case, Chamith et al fall back to trap-based instrumentation and replace instruction **A** directly with an INT3, at the cost of runtime performance.

In our implementation, we introduced a new mechanism which we call instruction sliding. If instruction **A** is longer than 1 byte, we can replace its first byte with a no-op and try instrumenting the following byte. For instance, in layout:

AbcDE(f)

the two most significant bytes of the offset are restricted. However, if we replace byte **A** with a no-op (**N**), then the layout becomes

(N)BcDEf

and our placement range is significantly wider as we have full control over the most significant byte.

If this technique fails or cannot be used, we fall back on trap-based instrumentation.

GDB Integration Implementing the instrumentation in GDB, we took advantage of several internal mechanisms of the debugger. Using the `gdbarch` infrastructure of GDB, we were able to clearly separate the implementation of architecture specific features (such as the content of the trampoline or the instruction punning) from the generic features that can be reused when implementing the feature on other architectures. The infrastructure to handle signals is already present in GDB, as it uses signals to insert software breakpoints in the code. We were able to use this breakpoint infrastructure to handle the signals generated by instruction punning.

We also designed a specific memory management system which we use to determine the available placements for trampolines. The trampolines are only written on specifically mapped pages, however we favor sharing pages between trampolines when possible, to reduce the memory footprint of this tool. For each trampoline, we reserve 256 bytes, we can thus pack up to 16 trampolines per 4KB page. New pages are mapped using the `mmap MAP_FIXED` option. If a call fails because the page is not available, our system stores that information in order to avoid repeating failed calls.

4.5 Application Integration

Using our tool, we built several applications that demonstrate how existing programs can be extended, improving their performance and flexibility. The first application is a performance improvement for an internal feature of GDB: conditional breakpoints. In the other two examples, we extend the capabilities of state-of-the-art RV tools by integrating them

within our framework and we show how they benefit from efficient, targeted and dynamic instrumentation.

4.5.1 Fast Conditional Breakpoints

Breakpoints are at the core of interactive debugging, as they allow the user to interrupt the program once it reaches a specific instruction. To refine this basic functionality, most debuggers –GDB included– provide a way to add certain conditions on the breakpoint, meaning that execution will only be interrupted if these conditions are verified. However, the condition-checking in GDB is notoriously slow, and thus setting a conditional breakpoint in a critical part of the code can significantly slow down the program, even if the condition is never met. The reason for this slowdown is architectural. Indeed, the condition checking is executed within the debugger and not the underlying program. This requires switching between applications and thus raising a signal to the kernel.

Using our instrumentation framework, we were able to modify the behavior of conditional breakpoints so that condition checking occurs in the program context, removing the need for a context switch and a kernel signal. Only if the condition is met, a signal is raised and the debugger hands control to the user. This makes the condition checking faster by a factor of 5000, as discussed in Section 4.6, making interactive debugging with conditional breakpoints more efficient, while keeping the same flexibility.

4.5.2 Targeted Dynamic Address Sanitizer

Address Sanitizer (ASan) [18] is a popular memory leak and corruption checking tool maintained by Google. It detects errors such as buffer overflows and use-after-free on heap, stack and global variables. For global and stack variables, Address Sanitizer allocates red zones around variables at compile time that are poisoned at runtime. Any access to those red zones will result in an error. For heap variables, it works by using a custom memory allocator that maps additional shadow memory to metadata about allocated regions. It also uses compile-time instrumentation to insert checks on memory accesses, by reading the shadow memory to determine if the area is properly allocated. Its instrumentation is designed to be implemented by compilers, it only consists in a read for every memory access. All the shadow memory mapping and writing are handled by the shared library `libasan`, linked with the program.

The key downside to this tool is that it requires recompiling and instrumenting the whole binary in order to be applied, even when only a small fraction of it needs to be checked.

Using our framework, we designed a new approach to ASan which circumvents those limitations by taking advantage of the fully dynamic instrumentation integrated in the debugger. By letting the user target only the designated parts of the code, we gain in flexibility as well as in performance. This can prove useful when a program fails after a series of patches. Targeting ASan directly on the modifications to the source code will often point to the error, while incurring less slowdown (than if the whole binary is instrumented) and getting rid of the burden of recompiling.

The architecture of our dynamic implementation of Address Sanitizer is illustrated in Figure 4.3. It consists of a package for our backbone VSCode extension, which contains a Python script to pilot instrumentation, and the `libasan` library. The user can choose which parts of the program are to be instrumented. It can be a few lines, a file, or the whole source. The VSCode extension interacts with the `clangd` language server to determine the different memory accesses in the selected areas. The IDE then interacts with GDB to instrument the predetermined areas, and executes the program in a console. For now, errors are still reported through the console, but this extension could be improved to have them appear directly in the source files.

Our dynamic implementation of Address Sanitizer is not meant to replace the original version. Indeed, it does not analyse stack and global variables, and it is generally slower, when instrumenting the whole binary using our method, than recompiling it completely with the proper option. This is because, for each instrumented file, the language server is asked to parse the file and determine the locations of all memory accesses. However, it can be very useful when only partly instrumenting a binary, a targeting capability that the original tool lacks. The dynamic capabilities also allow developers to work on a single binary, rather than having a second build for Address Sanitizer. The same binary can be used across the software development chain, from design to verification and deployment.

4.5.3 Fast Data Watch

Data Watch [92] is a runtime verification tool designed to identify heap memory corruption instances during the execution of a program. It works by overriding the memory allocation functions of the C library (malloc, realloc, etc.) with wrappers. These wrappers store the original address returned by the allocation function, as well as the requested size of the allocation, and return a tainted address pointing to a protected memory page. When a tainted address is dereferenced, the system raises a signal (SIGSEGV on Linux). Data Watch uses a kernel module to catch that signal. Using the tainted address, it can determine what the original address was, as well as the offset of that memory access compared to the base

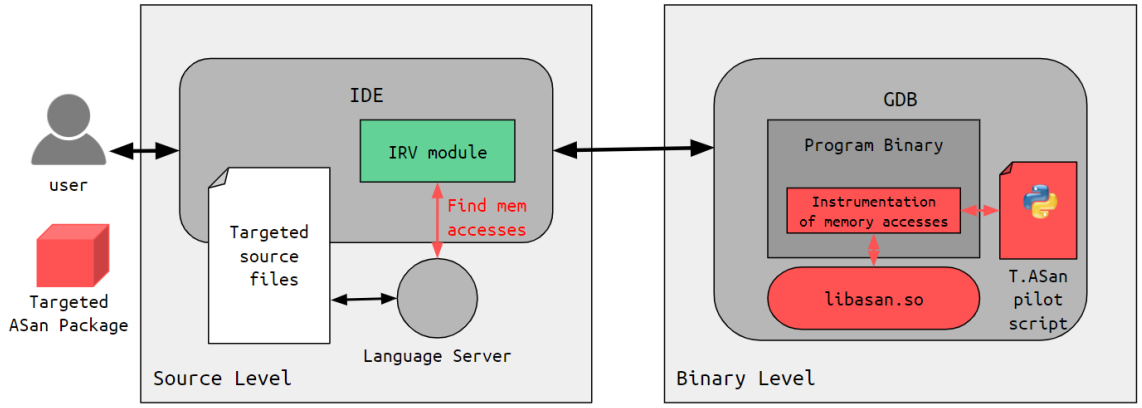

Figure 4.3 Targeted Dynamic Address Sanitizer architecture

tainted address. It can then compare this offset to the size of the allocated region, and return an error if there is an overflow. If the access is within bounds, the kernel module emulates the instruction with the correct address and returns control to the program.

The major problem with this tool on the Linux operating system is that each memory dereference generates a signal that has to be handled by the kernel, requiring a change of memory protection ring. This takes a long time and significantly slows down the program.

Based on our new framework, we were able to design a new version of Data Watch on x86_64 Linux which improves both the performance and usability of the tool. Our version, called Fast Data Watch (FDW) gets rid of the kernel module, and aims to keep a majority of checks within user space. Only the first execution of a dereference instruction will generate a signal. Subsequent executions will remain within the program context and generate minimal overhead by avoiding the change of protection ring. We can also make Fast Data Watch target a designated subset of variables by choosing which calls to the memory allocation functions should be overwritten.

The implementation occurs at multiple levels. First, a simple library is designed, which overrides memory allocation functions, and provides a new function called `memory_check_and_correct()`. This function checks that the tainted address received as argument is within bounds of the allocated object, and returns the corresponding untainted address.

At the debugger level, a signal handler is added to the SIGSEGV signal. When called, this handler analyses the faulty instruction, and determines which addresses were dereferenced. It then replaces that instruction with a jump to a trampoline which calls the `memory_check_and_correct()` function for those addresses (Figure 4.4). Still in the tram-

poline, the faulty instruction is executed again with the corrected addresses. The registers that were not written to are set back to their tainted value, and the program jumps back after the corrected instruction. Whenever that instruction is hit again later on, it will execute the trampoline and perform the correct memory checks before executing the original instruction, without raising a signal and going through the kernel.

On the IDE level, this application is integrated in our backbone application through a specific package. It contains the python script for signal handling as well as two versions of the library. The first version directly overrides all the memory allocation functions, while in the second one memory allocation calls need to be individually redirected to the modified version. This allows precise targeting of memory areas for analysis, in a similar fashion to targeted ASan.

FDW was implemented using a modified version of our new `patch` command, to be able to directly write machine code for efficiency and performance. A version using the unmodified command could be implemented but would need to use inline assembly to manipulate registers directly. The signal handling was performed using Python scripts and the capstone [6] library to analyse the instruction.

When compared to other tools such as regular Address Sanitizer, this memory corruption checker has the advantage of being fully dynamic and it does not need source code access, except to clarify bug reporting. One problem with this tool is handling kernel accesses to tainted pointers. We need to set a breakpoint at each system call to check that its arguments are not tainted. However, it is extremely difficult to insure that all the data accessed by the kernel will be untainted. For instance, the `ioctl` system call is very versatile and its behavior depends on the different drivers installed. One solution to this would be to use a kernel module, in parallel to FDW, to manage tainted pointers faults in the kernel.

4.5.4 Use cases

In Table 4.1, we illustrate where our different tools and framework belong in the domain of runtime verification.

In terms of canonical use cases, recompiling tools like Memcheck are relevant when performing heavyweight analysis of binaries before shipping. Indeed, tools like Memcheck cannot be activated during production as they have a very important performance and memory cost. Signal-based tools are more suited for interactive debugging and precise targeting. Indeed, due to the cost of executing a probe, keeping the number of generated signals low is essential to maintain a reasonable execution overhead. They are useful when trying to diagnose a bug

6. capstone-engine.org

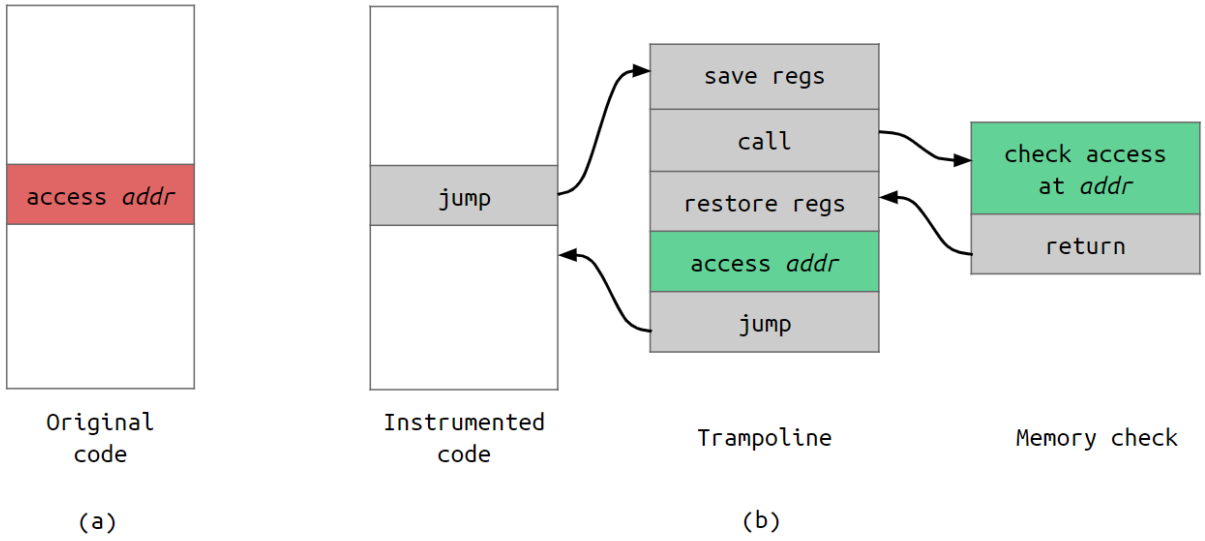

Figure 4.4 Fast Data Watch Trampoline: (a) original layout with tainted memory access (b) corrected layout using a trampoline

for which the user has a good idea about the likely source. Static instrumentation of programs aims to provide minimal performance overhead and be integrated in production programs when performance is not crucial. This is necessary to diagnose rare bugs that do not occur during testing. This is particularly useful in long running, online programs such as those running in servers. Trampoline-based tools aim to provide a middle ground between binary recompilation and static instrumentation. By being able to re-target dynamically where the analysis takes place, they can provide a smaller overhead than static instrumentation when the user knows which parts of the program may be problematic. This can be used for instance when modifying an existing program. During a certain lapse of time, it is possible to dynamically instrument the new additions to the program to check for errors during production. When no error has been detected after a certain time, it is also possible to remove the instrumentation, thus completely removing the performance overhead, without restarting the program.

4.6 Evaluation

Experimental Setup All measurements presented here were performed on a computer running Ubuntu 18.04 with Linux kernel 4.15.0-88. The CPU is an Intel i7-4790, with 32GB of 1600MHz DDR3 memory. The programs are compiled using GCC version 7.4.0, and we

Table 4.1 Categorisation of different instrumentation and RV tools and frameworks according to the technique used.

	Dynamic instrumentation			**Static instr.**
	Binary recompiling	Signal Injection	Trampoline-based	Direct injection
Instrumentation Framework	Pin	GDB	Dyninst LiteInst ***GDB patch***	clang
RV framework	Valgrind	Verde	***our framework***	
RV tool	Memcheck	Data Watch	***Targeted ASan*** ***Fast Data Watch***	ASan

In italic are the contributions of the present article.

patched GDB version 9.0.50.

4.6.1 Instrumentation Performance

To evaluate the instrumentation performance, we use the same benchmark proposed by Chamith et al [25] for their paper. It consists of several common tools found in the SPEC benchmark. This benchmark covers a variety of different cases for both single threaded (`bzip2`, `h264`, `perlbench`, `sjeng`) and multi threaded applications (`fluid` and `blackscholes`). The benchmarks represent different types of real applications, from compressing files (`bzip2`) to artificial intelligence (`sjeng`).

In our benchmark, we measure the execution time with and without instrumentation, as well as the time needed to set up the instrumentation. The instrumentation type selected was a procedure count, instrumenting every function entry and with probes that count the total number of calls to each function during the program execution. In Table 4.2 we show the number of probes installed per application and the memory overhead incurred. The impact is generally low, except in the case of the h264 benchmark, for which GDB allocates a large chunk of memory, independently of our instrumentation tool.

The results of the total run time and execution time analysis are presented in Figure 4.5. We were unable to successfully run LiteInst on the benchmark in our setup. We instead used the results from V. Zhao [32] to compare our approach to LiteInst and Pin, when the data is available. Note that the implementations of the procedure counter may differ slightly. However, this tool is simple enough that this difference should not be significant.

Our instrumentation framework performs slightly worse than state-of-the art instrumentation tools, while adding significant benefits in terms of flexibility and ease of use.

Table 4.2 Number of probes injected for each benchmark application. The amount of memory reserved for GDB and probes is also referenced and compared to the overall memory impact of the program.

Benchmark	Probes	Mem Overhead	Rel Overhead
bzip2	134	1.39MB	0.6%
blackscholes	36	24KB	0.04%
h264	644	61MB	204%
fluid	108	308KB	0.04%
perl	1996	20.5MB	6.6%
sjeng	188	1.88MB	1.04%

Table 4.3 Average installation and execution time for individual probes

	Average measured time
probe installation	30ms
probe execution	50ns

4.6.2 Applications performance

Fast Conditional Breakpoints To evaluate the performance of our implementation of fast conditional breakpoints, we tested it on a simple micro benchmark. We use the following test program, in order to measure the average execution time of a fast breakpoint condition check, compared to a regular breakpoint.

```
1 int main
2 {
3     int array[1000000];
4     for (int i = 0; i < 1000000; i++)
5     {
6         array[i]=i;
7     }
8 }
```

In order to check that all memory accesses are correct, we set a debugger breakpoint with condition:

$$(i < 0||i > 1000000)$$

This breakpoint never activates as its condition is never met. In figure **??**, we compare the execution times of the program with fast conditional breakpoints and regular conditional breakpoints. The baseline is the uninstrumented program run time.

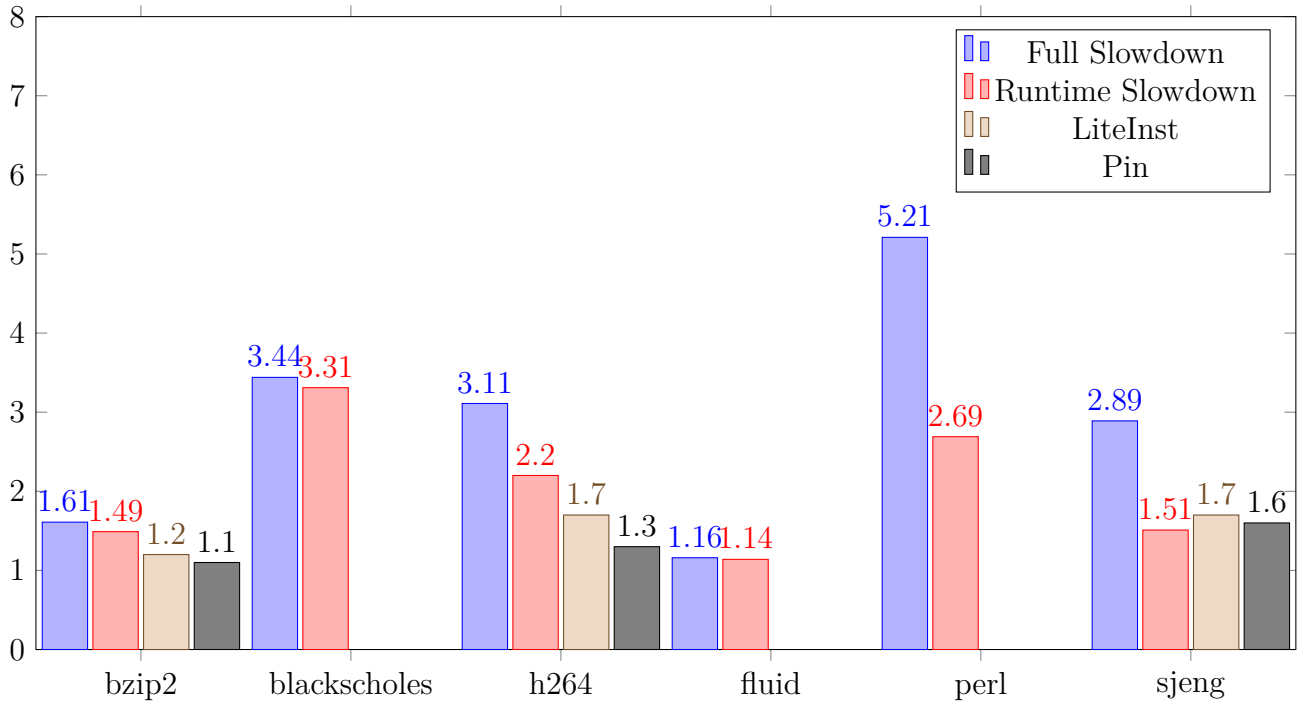

Figure 4.5 Instrumentation and runtime overhead of procedure counter, relative to uninstrumented execution time. Full slowdown includes both instrumentation and run time while *Runtime Slowdown* only measures program execution time. The slowdown factor compared to a uninstrumented run is shown here.

The execution is significantly faster when using fast conditional breakpoints, with a speedup of around 680 times compared to existing GDB conditional breakpoints. There is a significant slowdown compared to the uninstrumented baseline, however, which can be explained by the simplicity of this baseline. With a more complex program, for instance if we set $array[i]$ to a random value using `rand()`, the slowdown is lower (around 3 times).

From this data, we can interpret that the execution time of a Fast Conditional Breakpoint is about 30ns, or 100 CPU cycles. This improvement makes GDB conditional breakpoints usable in a wide variety of use cases, compared to the regular version which is often completely unusable.

Targeted Dynamic Address Sanitizer To evaluate our implementation of a dynamic targeted address sanitizer, we measured its performance when instrumenting parts or all of the `sjeng` benchmark, which simulates an AI chess player against a predefined move set. The partial instrumentation was performed on the `sjeng.c` source file. The execution times under different circumstances are presented in Figure 4.7. The run times presented were

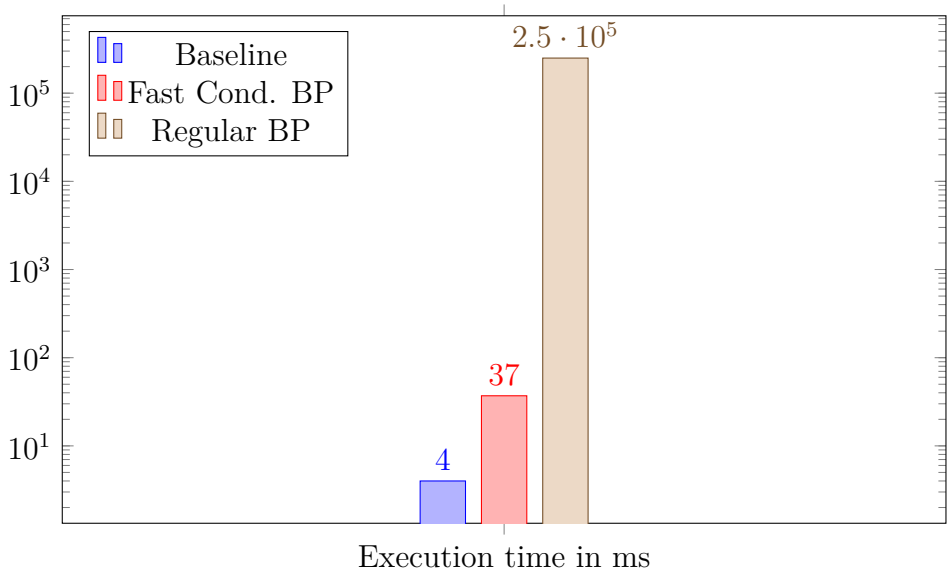

Figure 4.6 Comparison of GDB regular conditional breakpoints and fast conditional breakpoints on 1 000 000 condition checks

measured on 10 runs, and the standard deviation is shown on the graph. The number of injected probes during partial instrumentation was 25, and 1644 for full instrumentation. We checked the correctness of our tools by voluntarily introducing errors in different locations which were signaled during execution.

Our version of Address Sanitizer runs at virtually the same speed as the uninstrumented binary in the case of partial instrumentation. It even appears slightly faster, because in this measurement we do not consider the setup time of GDB, which includes loading the binary and shared libraries into memory. When looking at the full slowdown, including the time necessary to set up the instrumentation points, we observe a slowdown of around 1.5 times, which is faster than running a full analysis with Address Sanitizer. For full instrumentation, our approach is significantly slower than compile-time Address Sanitizer. This is due to the execution overhead of trampolines. When we take into consideration the setup time, the gap is even wider, as each file is individually processed in the compiler to generate the abstract syntax tree analysed by our tool.

Obviously, our application is not designed to replace Address Sanitizer. It aims to provide an alternative for targeted address sanitizing, which can be executed dynamically and at low cost. As for reference, on the dynamic alternative Valgrind Memcheck, we measured an average runtime of 790 seconds. This corresponds to a slowdown of 146 times, compared to uninstrumented execution and more than 2 times compared to our approach.

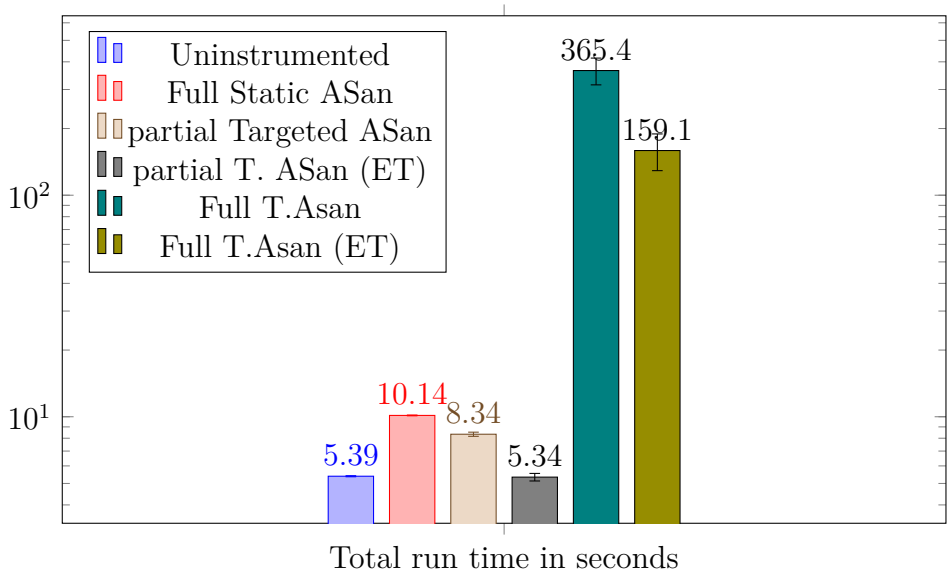

Figure 4.7 Comparison of regular Address Sanitizer vs our targeted version on the `sjeng` benchmark. ET stands for *execution time only*, which measures only the actual runtime of the program and not the setup time

Fast Data Watch In order to evaluate the correctness of our implementation of Fast Data Watch, we tested it on an existing bug of the *ffmpeg* binary (ticket 7394). This bug consists in a small chunk memory of memory lost, and a large one not freed at exit if an invalid input is given to the program. Using our tool, we were able to reproduce the Valgrind diagnosis and we found memory areas not freed at exit. However, our tool does not allow us to differentiate definitely lost memory (no pointers to it exist) from memory that is just never deallocated.

In terms of raw performance, we were not able to compare our version of Data Watch to the original kernel-based version, as they are implemented on different architectures. We measure the performance of our tool when instrumenting two programs representing extreme cases: *loop* and *unrolled*. Both use the same source file:

```
1 int main()
2 {
3   int *array =
4     (int*) malloc(100000 * sizeof(int));
5   for (int i = 0; i < 100000; i++)
6   {
7     array[i]=i;
8   }
9 }
```

In the case of the *unrolled* binary, the loop is completely unrolled so that no instruction is executed twice. This is the worst case for our program as each access generates a signal that has to be handled by the kernel and GDB. With both binaries, our tool reports that the variable `array` was not deallocated before program exit.

Table 4.4 indicates the execution times of the *loop* and *unrolled* binaries. For such simple cases, the slowdown is very important, even in the ideal case (78 times slowdown), because the base program most critical instructions are replaced with more than 200 instructions, in order to execute the trampoline and the memory bounds checking. That slowdown is significantly lower on more complex binaries. It is important to note that out of the 310ms, 190 are dedicated to loading GDB, which will not scale with the execution time of the binary. In the worst-case scenario (*unroll*), the slowdown is extremely important, because every load instruction generates a signal. In this scenario, using our approach is not recommended and using Valgrind Memcheck is significantly faster (around 4.5s of execution time).

Table 4.5 details the average measured times for probe installation and execution on our machine. The probe installation time increases with the number of already installed probes, as it needs to check for collisions on instrumented instructions. This measurement was obtained when installing 1000 probes.

4.7 Conclusion

In this article, we presented a new interactive and targeted runtime verification framework, which improves the performance of existing tools as well as increases their flexibility. As part of this framework, we designed a new instrumentation framework within the GNU debugger, which enables the simple and efficient instrumentation of programs. We also designed an IDE module acting as a unified interface for different runtime verification tools. We have shown that our tool offers performance close to the state-of-the-art, while adding flexibility and ease of use, making it more suited for widespread adoption.

In the future, we hope to improve the performance of our tool even more, bringing it on par with other less flexible instrumentation frameworks. We believe that there is significant

Table 4.4 Execution times of extreme cases when using our implementation of Fast Data Watch

Binary	Uninstrumented	Fast Data Watch
loop	4ms	310ms
unroll	4ms	57.6s

Table 4.5 Fast Data Watch average installation and execution time for individual probes

	Average measured time
probe installation	70ms
probe execution	500ns

headroom for further optimization. In terms of accessibility, IDE integration can be improved further, to display more data about the tools results, directly on and around the source code, and also to allow even more precise and automatic targeting of RV tools.

4.8 Acknowledgement

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

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Résultats Complémentaires : Traçage

Un des aspects principaux de la vérification à l'exécution (VE) d'un programme est la collecte efficace de données. Le traçage de programme est donc une composante essentielle de la VE, aussi bien en soutien d'autres outils d'analyse que seul. Cependant, aucun outil actuel n'allie réellement performance et flexibilité d'utilisation. Les outils statiques tels que LTTng [34] offrent des performances très intéressantes mais nécessitent une instrumentation statique des programmes, et donc une recompilation du binaire à chaque modification de l'instrumentation. Les outils de traçage dynamiques ont quant à eux de fortes limitations en termes de performance et de placement des points de trace (traceur GDB [26]) ou de format de trace (Extrae [45]).

En utilisant notre infrastructure de VE, il est possible d'adapter des outils de traçage statique pour les rendre totalement dynamiques, s'affranchissant ainsi des limitations des deux méthodes. De plus, le nouveau traceur dynamique peut profiter de l'interface graphique de l'environnement de développement pour offrir un ciblage fin des points de trace ainsi qu'une visualisation intégrée au code des points de trace et de leurs résultats.

5.1.1 Implémentation

Afin de démontrer les capacités de notre outil, nous avons créé un prototype de traceur entièrement dynamique basé sur une librairie de traçage statique dont nous fournissons l'instrumentation dynamique.

Cet outil de traçage reprend l'architecture par couche de notre infrastructure. Au niveau de l'éditeur, nous avons créé une application spécifique au traçage pour l'EDI Visual Studio Code. Cette application reçoit un module de traçage contenant une librairie de traçage, des scripts Python pour l'instrumentation ainsi qu'une liste de commandes afin d'initialiser la librairie de traçage. Les scripts sont chargés dans le débogueur et sont ensuite appelés par l'EDI lorsque l'utilisateur demande des actions spécifiques.

5.1.2 Évaluation

La librairie de traçage que nous avons choisi d'utiliser est Minitrace [1]. Il s'agit d'une librairie simple d'utilisation, qui stocke les données dans un tampon linéaire. Cette librairie est loin de l'état de l'art en matière d'exécution parallèle, car elle utilise un tampon commun dont l'accès est contrôlé par un verrou lors de l'écriture. Les données de traçage sont stockées au format *Trace Event* de Chrome, un format textuel utilisant un fichier JSON. L'intérêt principal de cette librairie dans notre approche de prototypage est sa flexibilité d'utilisation, mais ce travail devrait être porté sur un traceur complet tel que LTTng UST pour réellement être utilisable.

Pour l'évaluation, nous utilisons le même banc d'essai simple que pour les points d'arrêt conditionnels rapides afin de mesurer le temps d'exécution moyen d'un point de trace dynamique et de le comparer au même point de trace inséré de manière statique. A chaque point de trace la valeur de `array[i]` est enregistrée. Les résultats sont disponibles au tableau 5.1. Il est important de noter qu'étant donné que l'instrumentation se fait de manière interactive avec l'utilisateur, seul le temps d'exécution est représenté ici et non le temps nécessaire à l'instrumentation.

Tableau 5.1 Comparaison entre les temps d'exécution de points de trace insérés dynamiquement

	Temps total d'exécution	Temps d'exécution d'un point de trace
Référence	4ms	
Minitrace dynamique	180ms	176ns
Traceur GDB	515ms [2]	82μs
Minitrace statique	57ms	53ns
LTTng UST	136ms	132ns

Nous avons également comparé notre approche avec d'autres techniques de traçage statiques et dynamiques sur le même banc d'essai. Pour GDB, nous avons essayé de mettre en place le traçage standard basé sur les signaux (comme des points d'arrêt) ainsi que le traçage rapide qui utilise un système de trampoline. Dans la première approche, seuls 6160 points de trace ont été effectivement enregistrés sur les 1 000 000. Pour la seconde approche, GDB a rencontré une erreur de gestion de mémoire qui l'empêchait d'injecter un point de trace bien que l'instruction à instrumenter était de taille suffisante.

Pour comparer les outils, il est intéressant de noter ce qui est mesuré pour chacune des approches. Pour Minitrace, le binaire contient un appel inséré de façon statique ou dynamique

1. github.com/hrydgard/minitrace
2. Tous les points de trace n'ont pas fonctionné

à une fonction de collecte. Cette fonction prend en argument la valeur enregistrée (un entier sur 4 octets), ainsi que deux chaînes de caractères décrivant le type de point de trace ainsi que le nom de la variable tracée. La fonction de collecte récupère la date du système, et stocke cette valeur dans un tampon linéaire alloué à l'initialisation. Pour la librairie LTTng UST, le binaire contient un appel à la fonction de collecte injecté à la compilation, dont les arguments sont également deux identifiants ainsi que la valeur enregistrée. La fonction de collecte récupère la date système et transmet les données au *consumer daemon*, un second processus responsable d'enregistrer les données de trace sur le tampon circulaire sans verrou associé au processeur. Enfin, pour GDB, un point d'arrêt est placé dans le code, et à chaque interruption du programme par ce point d'arrêt, GDB collecte et enregistre les données demandées.

On note l'efficacité de la librairie Minitrace qui est plus de deux fois plus rapide que LTTng UST pour exécuter un point de trace statique, ceci en raison de la simplicité de ses mécanismes. Cette vitesse rend notre traceur dynamique presque aussi rapide que le traceur statique haute performance LTTng, mais il faut prendre en compte que l'écriture sur disque n'est pas comptée dans cette mesure, ni les temps d'initialisation des librairies. L'insertion dynamique d'un point de trace vient avec un coût d'environ 120ns par exécution de ce point de trace, mais la durée d'exécution reste du même ordre de grandeur que la version statique, contrairement au traceur intégré GDB qui multiplie le coût par plus de 1000.

5.2 Analyse des résultats obtenus

5.2.1 Instrumentation avec GDB

Dans la plupart des cas, l'empreinte mémoire de notre outil est raisonnable, même lorsque le nombre de points d'instrumentation est important (6% dans le cadre du programme Perl, pour presque 2000 points). Un problème que nous rencontrons est le comportement parfois obscur de GDB. Ainsi, dans le cadre du programme test *h264*, GDB réserve une quantité importante de mémoire dès le chargement du binaire (environ 30MB, soit déjà 100% de la mémoire utilisée par le programme test). La complexité du débogueur rend l'origine de cette utilisation de mémoire difficile à traquer.

En termes de temps de calcul, notre approche s'exécute presque aussi rapidement que les outils optimisés Pin et LiteInst, voire plus rapidement dans le cas du banc d'essai *sjeng*. Une meilleure optimisation de notre plateforme pourrait passer par un groupement des commandes. En effet, dans notre cas, une commande est envoyée par point d'instrumentation, et l'étape de lecture et d'interprétation est répétée inutilement.

5.2.2 Applications

Pour la version dynamique et ciblée d'Address Sanitizer, nos résultats montrent bien qu'elle n'est pas en mesure de remplacer la version statique, la durée d'exécution étant presque 15 fois plus importante lorsque tous les accès sont instrumentés. Ceci s'explique par le fait que les instructions nécessaires à l'instrumentation ajoutées par ASan sont très courtes. Les instructions que nous injectons pour sauvegarder les registres et appeler la fonction d'instrumentation ont donc un coût relatif beaucoup plus important que dans le cas d'une instrumentation complexe. Dans le cadre d'une instrumentation partielle, cependant, notre approche peut offrir de meilleures performances au niveau global tout en offrant la flexibilité de l'instrumentation dynamique, qui permet de maintenir un unique fichier binaire pour les tests et la publication.

Notre implémentation de Data Watch propose des résultats intéressants dans le cadre d'applications simples, mais sa gestion des appels systèmes est à améliorer. En effet, nous générons un point d'arrêt à chaque appel système, et corrigeons tous les registres impliqués avant et après l'appel. Ceci entraîne un coût conséquent dans le test sur des applications plus complexes. Ainsi, dans le cadre de *ffmpeg*, nous observons un ralentissement de plus de 15 fois, ce qui est presque aussi lent que d'utiliser Memcheck, pourtant plus exhaustive. Un travail d'optimisation reste donc nécessaire sur cette application.

Enfin, pour ce qui est du traçage, nous pensons que notre approche peut permettre une facilitation du traçage par l'intégration du traçage haute performance à plus d'applications. En effet, l'exécution dynamique d'un point de trace a un coût du même ordre de grandeur que l'exécution statique de celui-ci. On pourrait donc envisager une version totalement dynamique et interactive de LTTng UST, dont le coût serait environ 1,5 à 2 fois plus important que celui la version statique. Cela resterait utilisable dans la plupart des applications.

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Au cours de ce projet de recherche, nous avons produit une nouvelle plateforme pour la création et l'intégration d'outils de vérification dynamique de programmes. Celle-ci rend possible la création d'outils dynamiques interactifs et ciblés, sans compromettre la performance du programme débogué. Pour ce faire, elle repose sur une architecture séparée en deux domaines : le domaine des fichiers source et le domaine binaire.

Le domaine source repose sur un module que nous avons conçu pour l'éditeur de code modulaire Visual Studio Code, mais qui est également fonctionnel sur Eclipse Theia. Ce module permet de charger différents outils de VE et d'offrir une interface adaptée à chacun, via des commandes qui permettent d'interagir avec le domaine binaire ou des serveurs de langages.

Le domaine binaire contient le débogueur GDB, qui fournit l'infrastructure nécessaire à la manipulation de fichiers binaires et au chargement des librairies nécessaires aux outils de VE. Afin de compléter la capacité de GDB à agir comme serveur de modification binaire, nous avons implémenté un nouveau système d'instrumentation dans le débogueur. Il permet l'injection de code directement dans le fichier compilé pendant son exécution, par l'utilisation de trampolines et non de signaux noyau. Cette amélioration rend possible l'instrumentation dynamique performante via GDB, et nous nous en servons notamment pour reproduire une instrumentation normalement faite par le compilateur dans le cas d'outils statique comme Address Sanitizer.

En utilisant ces nouveaux outils, nous avons pu d'une part créer un système de points d'arrêt conditionnels rapides dans GDB, accélérant d'un facteur 5000 l'exécution de ces derniers. Nous avons également créé des versions ciblées et entièrement dynamiques des outils Address Sanitizer et Data Watch, offrant une alternative intéressante dans le cas d'une instrumentation partielle des programmes. Enfin, nous avons produit une plateforme de traçage dynamique qui permet la collecte efficace de données internes via une interface intégrée au code source.

Par nos travaux, nous avons démontré l'intérêt d'adopter une architecture plus modulaire pour les outils de vérification à l'exécution, permettant le partage des fonctionnalités de base. En simplifiant ainsi le développement d'outils de VE, une plateforme commune peut permettre aux développeurs d'outils de se focaliser sur les fonctionnalités avancées de leur programme, tout en offrant un large panel de fonctionnalités de base, comme l'instrumenta-

tion dynamique et une interface graphique. Une large adoption de ces fonctionnalités pourrait faciliter l'emploi des différents outils de VE et en populariser l'usage.

6.2 Limitations de la solution proposée

Il est important de noter que l'instrumentation dynamique de programmes telle que nous la proposons n'a pas vocation à remplacer l'instrumentation statique proposée par différents outils. En effet, les précautions qui doivent être prises dans le cadre d'une modification dynamique du code font que la performance de code ajouté statiquement sera toujours meilleure que celle de code ajouté dynamiquement. Cette différence étant due à un nombre constant d'instructions, le coût relatif de l'instrumentation dynamique sera autant plus important que le code injecté est rapide.

Vis à vis de la plateforme d'instrumentation en elle-même, son impact en mémoire peut être important car elle alloue en mémoire une à deux pages par point d'instrumentation, bien que la mémoire effectivement utilisée soit bien moindre. Ceci est en partie dû à la réutilisation des fonctionnalités internes de la commande *compile* de GDB. En effet, la mémoire étant libérée directement après l'utilisation dans ce cadre, l'économie de mémoire n'est pas une priorité.

Enfin, les applications que nous avons créées à partir de ASan et Data Watch sont des prototypes et ne sont pas exempts de bogues, notamment pour Data Watch dans le cadre d'interactions avec le noyau. Dans les tests que nous avons menés, l'utilisation de l'appel système *ioctl* était problématique car il ne possède pas de typage défini pour ses arguments. Une adresse non corrigée peut donc se glisser dans le noyau et provoquer une erreur de segmentation qui n'est pas rattrapée par GDB. Pour les points d'arrêt rapides, l'interface utilisateur laisse à désirer car elle indique le point d'arrêt à sa position en mémoire (dans la fonction d'instrumentation) et non là où l'utilisateur souhaitait la placer. Différencier la position effective en mémoire de ce point d'arrêt de sa position théorique serait nécessaire.

6.3 Améliorations futures

Une amélioration importante à notre plateforme en termes de facilité d'accès serait d'intégrer les modifications apportées à GDB au sein de l'arbre principal. Ceci nécessite que le code soit entièrement fonctionnel et conforme aux spécifications du projet, ce sur quoi nous travaillons.

Cette fonctionnalité d'instrumentation est elle-même à améliorer afin d'accélérer le placement et l'exécution des points d'instrumentation. La rapidité de placement des trampolines pourrait être significativement améliorée par une gestion plus avancée de la mémoire, qui

permettrait également de rassembler au maximum les trampolines et réduire l'impact en mémoire. Une modification de l'infrastructure de la commande *compile* permettrait également un impact moindre en mémoire.

Le module pour Visual Studio Code est très basique et pourrait bénéficier de nombreuses améliorations tant en performance qu'en fonctionnalités. Parmi les fonctionnalités les plus importantes à ajouter, l'intégration des résultats de VE au code peut offrir une vision intéressante à l'utilisateur. Nous avons déjà implémenté cette fonctionnalité dans le cas de l'outil de traçage dynamique, et offrir aux outils une interface permettant de prendre avantage de cette fonctionnalité via l'application généraliste nous paraît être une amélioration importante. Enfin, nous n'avons pas implémenté l'interface entre VSCode et GDB via le Debug Adapter Protocol. L'utilisation de ce protocole serait préférable car cela permettrait d'éviter la multiplication des canaux de communications.

Comme mentionné dans le paragraphe précédent, les applications que nous avons construites peuvent encore être largement améliorées. En plus de la gestion des appels systèmes pour Data Watch et de l'affichage des points d'arrêts, on peut citer la prise en charge des variables de la pile qui pourrait être implémentée pour Address Sanitizer. Ceci dit, cette fonctionnalité risque d'être moins performante puisque le volume d'instrumentation nécessaire serait plus important que pour les variables du tas.

La poursuite des présents travaux offre donc des perspectives intéressantes, qui faciliteront l'adoption large de notre outil.

RÉFÉRENCES

[1] Eclipse ide. [En ligne]. Disponible : eclipse.org/eclipseide/

[2] Visual studio. [En ligne]. Disponible : visualstudio.microsoft.com

[3] Git. [En ligne]. Disponible : git-scm.com

[4] Visual studio debugger. [En ligne]. Disponible : docs.microsoft.com/en-us/visualstudio/debugger/

[5] Vim. [En ligne]. Disponible : vim.org

[6] Notepad++. [En ligne]. Disponible : notepad-plus-plus.org

[7] Microsoft. (2015) Visual studio code. [En ligne]. Disponible : code.visualstudio.com

[8] R. Stallman *et al.*, "Debugging with gdb," *Free Software Foundation*, vol. 51, p. 02 110–1301, 2002.

[9] pdb - the python debugger. [En ligne]. Disponible : docs.python.org/2/library/pdb.html

[10] Debug adapter protocol. [En ligne]. Disponible : microsoft.github.io/debug-adapter-protocol/

[11] Language server protocol. [En ligne]. Disponible : microsoft.github.io/language-server-protocol/

[12] clangd. [En ligne]. Disponible : clang.llvm.org/extra/clangd/

[13] Electron. [En ligne]. Disponible : electronjs.org

[14] Node.js. [En ligne]. Disponible : nodejs.org/en/about/

[15] Microsoft. Vs code remote development. [En ligne]. Disponible : code.visualstudio.com/docs/remote/remote-overview

[16] Theia ide. [En ligne]. Disponible : theia-ide.org

[17] A. Griffith, *GCC : the complete reference*. McGraw-Hill, Inc., 2002.

[18] K. Serebryany *et al.*, "Addresssanitizer : A fast address sanity checker," dans *Presented as part of the 2012 USENIX Annual Technical Conference*, 2012, p. 309–318.

[19] Lldb. [En ligne]. Disponible : lldb.llvm.org

[20] J. Keniston, A. Mavinakayanahalli et V. Prasad, "Ptrace , utrace , uprobes : Lightweight , dynamic tracing of user apps," 2007.

[21] SunSoft. x86 assembly language reference manual. [En ligne]. Disponible : docs.oracle.com/cd/E19641-01/802-1948/802-1948.pdf

[22] C. C. Williams et J. K. Hollingsworth, “Interactive binary instrumentation,” dans *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, 2004, p. 32.

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

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

[25] B. Chamith *et al.*, “Instruction punning : Lightweight instrumentation for x86-64,” dans *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, p. 320–332.

[26] N. Mc Guire et W. Baojun, “A hardware architecture independant implementation of gdb tracepoints for linux,” dans *Proceedings of the 8 th Real-Time Linux Workshop*. Citeseer, 2007.

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

[28] C.-K. Luk *et al.*, “Pin : building customized program analysis tools with dynamic instrumentation,” *Acm sigplan notices*, vol. 40, n°. 6, p. 190–200, 2005.

[29] N. Nethercote et J. Seward, “Valgrind : a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan notices*, vol. 42, n°. 6, p. 89–100, 2007.

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

[31] V. Developers, “Valgrind user manual,” 2015.

[32] V. Zhao, “Evaluation of dynamic binary instrumentation approaches : Dynamic binary translation vs. dynamic probe injection,” 2018.

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

[34] M. Desnoyers et M. R. Dagenais, “The lttng tracer : A low impact performance and behavior monitor for gnu/linux,” dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, p. 209–224.

[35] M. Desnoyers, “Man page lttng-ust. 3,” 2012.

[36] Lttng live mode. [En ligne]. Disponible : lttng.org/docs/v2.11/#doc-lttng-live

[37] Lttng snapshot mode. [En ligne]. Disponible : lttng.org/docs/v2.11/#doc-taking-a-snapshot

[38] The common trace format. [En ligne]. Disponible : diamon.org/ctf/

[39] P.-M. Fournier, M. Desnoyers et M. R. Dagenais, “Combined tracing of the kernel and applications with lttng,” dans *Proceedings of the 2009 linux symposium*. Citeseer, 2009, p. 87–93.

[40] S. S. Shende et A. D. Malony, “The tau parallel performance system,” *The International Journal of High Performance Computing Applications*, vol. 20, n°. 2, p. 287–311, 2006.

[41] Open tracing. [En ligne]. Disponible : opentracing.io

[42] Jaeger : open source, end-to-end distributed tracing. [En ligne]. Disponible : jaegertracing.io

[43] Zipkin. [En ligne]. Disponible : zipkin.io

[44] B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[45] Extrae. [En ligne]. Disponible : tools.bsc.es/extrae

[46] M. Gebai et M. R. Dagenais, “Survey and analysis of kernel and userspace tracers on linux : Design, implementation, and overhead,” *ACM Computing Surveys (CSUR)*, vol. 51, n°. 2, p. 1–33, 2018.

[47] Paraver : a flexible performance analysis tool. [En ligne]. Disponible : tools.bsc.es/paraver

[48] The chromium projects : Chromium. [En ligne]. Disponible : chromium.org/Home

[49] The trace event profiling tool. [En ligne]. Disponible : chromium.org/developers/how-tos/trace-event-profiling-tool

[50] Eclipse trace compass. [En ligne]. Disponible : eclipse.org/tracecompass/

[51] G. Bastien. Trace compass ease scripting. [En ligne]. Disponible : github.com/tahini/tracecompass-ease-scripting

[52] Microsoft. Pe format. [En ligne]. Disponible : docs.microsoft.com/en-us/windows/win32/debug/pe-format

[53] M. J. Eager *et al.*, “Introduction to the dwarf debugging format,” *Group*, 2007.

[54] P. Krishnan, “Hardware breakpoint (or watchpoint) usage in linux kernel,” dans *Proceedings of the Linux Symposium*. Citeseer, 2009, p. 149–158.

[55] Q. Zhao *et al.*, “How to do a million watchpoints : Efficient debugging using dynamic instrumentation,” dans *International Conference on Compiler Construction*. Springer, 2008, p. 147–162.

[56] A. Vasudevan et R. Yerraballi, “Stealth breakpoints,” dans *21st Annual Computer Security Applications Conference (ACSAC’05)*. IEEE, 2005, p. 10–pp.

[57] M. N. Gagnon, S. Taylor et A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security & Privacy*, vol. 5, n°. 3, p. 82–84, 2007.

[58] P. B. Kessler, "Fast breakpoints : Design and implementation," *ACM SIGPLAN Notices*, vol. 25, n°. 6, p. 78–84, 1990.

[59] Windbg. [En ligne]. Disponible : windbg.org

[60] B. Boothe, "Efficient algorithms for bidirectional debugging," dans *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, p. 299–310.

[61] J. Engblom, "A review of reverse debugging," dans *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*. IEEE, 2012, p. 1–6.

[62] rr : lightweight recording & deterministic debugging. [En ligne]. Disponible : rr-project.org

[63] N. Sidwell *et al.*, "Non-stop multi-threaded debugging in gdb," dans *GCC Developers' Summit*, vol. 117. Citeseer, 2008.

[64] H. Pan *et al.*, "Controlling program execution through binary instrumentation," *ACM SIGARCH Computer Architecture News*, vol. 33, n°. 5, p. 45–50, 2005.

[65] Gdb/gcc compile and execute project. [En ligne]. Disponible : sourceware.org/gdb/wiki/GCCCompileAndExecute

[66] M. Dmitriev, "Profiling java applications using code hotswapping and dynamic call graph revelation," dans *Proceedings of the 4th International Workshop on Software and Performance*, 2004, p. 139–150.

[67] E. de Claro February *et al.*, "ncurses."

[68] Ddd - data display debugger. [En ligne]. Disponible : gnu.org/software/ddd/

[69] J.-C. Delay, "Gem# 119 : Gdb scripting—part 1," *ACM SIGAda Ada Letters*, vol. 34, n°. 1, p. 48–52, 2015.

[70] R. Jakse *et al.*, "Interactive runtime verification—when interactive debugging meets runtime verification," dans *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, p. 182–193.

[71] N. Chatterjee *et al.*, "Debugging multi-threaded applications using pin-augmented gdb (pgdb)," dans *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015, p. 109.

[72] Q. Zhao *et al.*, “How to do a million watchpoints : Efficient debugging using dynamic instrumentation,” dans *International Conference on Compiler Construction*. Springer, 2008, p. 147–162.

[73] J. Silva, “A survey on algorithmic debugging strategies,” *Advances in engineering software*, vol. 42, n°. 11, p. 976–991, 2011.

[74] E. Y. Shapiro, “Algorithmic program diagnosis,” dans *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1982, p. 299–308.

[75] E. Av-Ron, *Top-down diagnosis of Prolog programs*, 1984.

[76] B. Pope, “Declarative debugging with buddha,” dans *International School on Advanced Functional Programming*. Springer, 2004, p. 273–308.

[77] R. Caballero, C. Hermanns et H. Kuchen, “Algorithmic debugging of java programs,” *Electron. Notes Theor. Comput. Sci.*, vol. 177, p. 75–89, juin 2007. [En ligne]. Disponible : https://doi.org/10.1016/j.entcs.2007.01.005

[78] H. Grgic, B. Mihaljević et A. Radovan, “Comparison of garbage collectors in java programming language,” dans *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2018, p. 1539–1544.

[79] T. Perl, “Python garbage collector implementations cpython, pypy and gas,” 2012.

[80] H.-J. Boehm et M. Weiser, “Garbage collection in an uncooperative environment,” *Software : Practice and Experience*, vol. 18, n°. 9, p. 807–820, 1988.

[81] G. Xuetao *et al.*, “Quantitative analysis of boehm’s gc.”

[82] Clang : a c language family frontend for llvm. [En ligne]. Disponible : clang.llvm.org

[83] T. Kremenek. (2008) Finding software bugs with the clang static analyzer. [En ligne]. Disponible : llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf

[84] S. Cherem, L. Princehouse et R. Rugina, “Practical memory leak detection using guarded value-flow analysis,” dans *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, p. 480–491.

[85] Y. Sui, D. Ye et J. Xue, “Static memory leak detection using full-sparse value-flow analysis,” dans *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, p. 254–264.

[86] Google. Leaksanitizer. [En ligne]. Disponible : github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer

[87] Leak sanitizer vs heap checker. [En ligne]. Disponible : github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizerVsHeapChecker

[88] Massif : a heap profiler. [En ligne]. Disponible : valgrind.org/docs/manual/ms-manual.html

[89] Heaptrack. [En ligne]. Disponible : github.com/KDE/heaptrack

[90] J. Xu *et al.*, "Automatic diagnosis and response to memory corruption vulnerabilities," dans *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, p. 223–234.

[91] efence(3)- linux man page. [En ligne]. Disponible : linux.die.net/man/3/efence

[92] J. Puncher. (2017) Data watch. Ciena. [En ligne]. Disponible : tracingsummit.org/ts/2017/files/TS17-datawatch.pdf

[93] K. Serebryany et T. Iskhodzhanov, "Threadsanitizer : data race detection in practice," dans *Proceedings of the workshop on binary instrumentation and applications*, 2009, p. 62–71.

[94] Google. Threadsanitizer deadlock detector. [En ligne]. Disponible : github.com/google/sanitizers/wiki/ThreadSanitizerDeadlockDetector

[95] Helgrind : a thread error detector. [En ligne]. Disponible : valgrind.org/docs/manual/hg-manual.html

[96] N. Nethercote et J. Seward, "Valgrind : A program supervision framework," *Electronic notes in theoretical computer science*, vol. 89, nº. 2, p. 44–66, 2003.

[97] F. Chen et G. Roşu, "Mop : an efficient and generic runtime verification framework," dans *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, 2007, p. 569–588.

[98] R. Jakse *et al.*, "Interactive runtime verification—when interactive debugging meets runtime verification," dans *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, p. 182–193.