



Titre: Techniques d'analyse ciblée de mémoire en temps réel pour les
Title: systèmes embarqués

Auteur: David Piché
Author:

Date: 2023

Type: Mémoire ou thèse / Dissertation or Thesis

Référence: Piché, D. (2023). Techniques d'analyse ciblée de mémoire en temps réel pour les
Citation: systèmes embarqués [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie.
<https://publications.polymtl.ca/55856/>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/55856/>
PolyPublie URL:

**Directeurs de
recherche:** Michel Dagenais
Advisors:

Programme: Génie informatique
Program:

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

**Techniques d'analyse ciblée de mémoire en temps réel pour les systèmes
embarqués**

DAVID PICHÉ

Département de génie informatique et génie logiciel

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

Septembre 2023

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

**Techniques d'analyse ciblée de mémoire en temps réel pour les systèmes
embarqués**

présenté par **David PICHÉ**

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

Heng LI, président

Michel DAGENAIS, membre et directeur de recherche

Richard GOURDEAU, membre

DÉDICACE

*À mes parents,
qui ont nourri ma curiosité et mon esprit scientifique.*

REMERCIEMENTS

Je tiens premièrement à remercier mon directeur de recherche, Michel Dagenais, qui m’a conseillé tout au long de mon projet de maîtrise. Il a su me transmettre son enthousiasme dans ce domaine. Il m’a ainsi enseigné beaucoup de connaissances intéressantes liés au domaine de gestion de mémoire. Je remercie aussi Mohammad Nassiri qui m’a guidé durant le projet.

Je remercie Olivier Dion pour son aide à l’intégration de son outil « Libpatch » à ma conception d’outil. Je le remercie aussi pour nos nombreuses rencontres, en plus de ses nombreux correctifs sur sa librairie afin de répondre aux besoins de mon projet.

Je remercie mes collègues de laboratoires, qui m’ont bien tenu compagnie tout au long de mon parcours.

Enfin, je tiens à remercier chaque lecteur pour leur temps dédié à la lecture de ce mémoire.

RÉSUMÉ

Les erreurs lors des accès en mémoire pendant l'exécution d'un programme sont toujours d'actualité 50 ans après la création du langage de programmation C. Toutefois, des outils de vérification de mémoire peuvent détecter ces erreurs mémoires. Dans le contexte des systèmes embarqués, il est impératif de minimiser le surcoût et l'invasivité de l'approche.

Plusieurs outils proposent une analyse statique du programme, en détectant les erreurs mémoires dans le code source, hors exécution du programme. Dans le contexte précédemment décrit, il est plutôt préféré d'analyser le programme lors du temps exécution. En effet, l'accès au code source n'est pas requis, contrairement à lors de l'utilisation de l'analyse statique du programme. De plus, les erreurs de mémoire qui découlent de mémoire allouée dynamiquement représentent un défi considérable à détecter hors exécution. Ainsi, l'outil doit effectuer la vérification de mémoire en temps d'exécution.

L'objectif de ce mémoire se résume donc sur la conception de techniques visant à réduire le surcoût de temps et de mémoire lié à ces outils, dans un contexte d'utilisation de systèmes embarqués. Il faut aussi s'assurer que la technique soit la moins invasive possible.

Pour ce faire, nous introduisons une première technique en utilisant `mprotect` afin de protéger des pages de mémoire lors de l'allocation des zones mémoire. Quand une de ces pages est accédée lors d'accès mémoire, l'utilisation de *PTrace* permet ensuite à autre fil d'exécution de :

- Désassembler l'instruction
- Extraire l'adresse en question
- Effectuer la vérification des bornes adressables
- Enlever la protection de la (des) page(s) concernée(s)
- Ré exécuter l'instruction
- Remettre la protection de la (des) page(s) concernée(s)

Cette approche est malheureusement très coûteuse en mémoire et en temps. Cependant, le surcoût en mémoire peut être réduit en permettant à différents objets protégés de se trouver sur la même page, si la taille le permet.

Une technique alternative pour protéger la mémoire lors de l'allocation est d'utiliser des étiquettes de pointeurs. En utilisant les 16 bits inutilisés de l'architecture x86_64, on peut enregistrer une étiquette unique pour chaque objet alloué. Une table de métadonnées enregistre les données pertinentes pour la vérification des bornes, et l'étiquette sert donc d'index

pour cette table.

Dans les deux cas, le processus de vérification de mémoire ne se fait que pour les objets protégés. Bien que notre approche protège en ce moment par défaut toutes les allocations d’objets, cela laisse la possibilité aux développeurs de ne protéger qu’un sous-ensemble des allocations, réduisant ainsi le surcoût en temps et en mémoire.

Cependant, le surcoût en temps reste élevé, du fait, entre autres, que les instructions sont toujours désassemblées, même s’ils l’ont déjà été précédemment.

Nous proposons donc d’utiliser le travail d’un collègue de laboratoire, la librairie *Libpatch*, afin d’assurer une baisse significative de surcoût lors de la répétition d’instructions. En effet, lorsqu’une instruction est désassemblée pour la première fois, *Libpatch* modifie le code machine afin d’y insérer des « probes » et émuler l’instruction pour les accès futurs. *Libolx*, une version minimaliste de *Libpatch*, est aussi utilisée comme technique alternative.

Bien que les approches ne sont pas tout à fait complètes, des études de fréquence des allocations et accès mémoire, sur des bancs d’essai normalisés, et des mesures de performance de plusieurs alternatives sur un micro test, confirment le potentiel de cette approche. En effet, avec l’approche *Libpatch*, pour des exécutions subséquentes d’instructions, on ne dénote un surcoût de vérification d’accès mémoire réduit, passant de 1,3 microseconde à environ 65 nanosecondes.

Ce travail jette donc les bases d’une boîte à outils, versatile et efficace, de techniques pour vérifier la validité des accès en mémoire, dans un contexte de systèmes embarqués ou temps réel. L’usager peut sélectionner finement le sous-ensemble des objets à vérifier, ce qui lui permet de réduire d’autant le surcoût à l’exécution. Il peut aussi choisir entre quelques approches différentes pour effectuer la vérification, approches présentant des compromis différents en termes d’invasivité et de type de surcoût. Il reste cependant plusieurs aspects qui pourront être encore plus optimisés, lors de travaux subséquents, et un certain nombre de cas qui ne sont pas encore pris en compte, par exemple avec la gestion des arguments protégés des appels système.

ABSTRACT

Memory access errors during execution are still a concern 50 years after the C programming language was created. However, memory verification tools can help detect these memory errors. In the context of embedded systems, it is of utmost importance to ensure low overhead and invasiveness of the tool.

Many tools offer a static analysis of the program, by detecting memory errors in the source code while the program is not running. In the case of embedded systems, however, it is preferred to analyze the program while it is running. Indeed, access to the source code is not required, as it would be with static analysis. In addition, memory errors related to dynamically allocated memory are challenging to predict before execution. As such, memory analysis must be done at runtime.

The main objective of this thesis is to design new techniques in order to reduce the performance and memory overhead related to these tools, in hopes of applying them to embedded systems. The technique must also ensure low invasiveness.

To do this, we introduce a first technique using `mprotect` to protect memory pages when allocating memory. When a memory access instruction accesses one of these aforementioned protected memory pages, the use of *PTrace* enables another thread to:

- Disassemble the instruction
- Extract the address in question
- Perform bounds checking
- Remove protection of the affected memory page(s)
- Re-execute the instruction
- Restore protection of the affected memory page(s)

This approach is unfortunately very costly both in memory and time overheads. However, the memory overhead can be reduced by allowing different protected objects to coexist in the same memory page if size permits.

An alternative technique for protecting memory at allocation is to use pointer taints. By using the unused 16 bits in the x86_64 architecture, we can store a unique taint for each allocated object. A metadata table stores relevant information for bounds checking, with the taint serving as the index for this table.

In both cases, this memory verification process is only done for protected objects. Even if our current approach protects every memory allocation by default, developers can protect a

subset of allocations, reducing both time and memory overheads.

However, performance overhead stays high because, among others, instructions are disassembled each time, even if they were previously disassembled.

We propose using the work of a colleague of the DORSAL laboratory, the *Libpatch* library, in order to significantly reduce overhead when instructions are executed multiple times. Indeed, when an instruction is disassembled for the first time, *Libpatch* modifies the machine code to insert probes and emulate the instruction for subsequent accesses. *Libolx*, a minimalist version of *Libpatch*, is also used as an alternative.

Even though the different approaches are not yet complete, studies of allocation and memory access frequencies on normalized benchmarks, and performance overhead measurements of multiple alternatives on a micro-benchmark, confirm the potential of this approach. Indeed, with the *Libpatch* approach, we observe a reduced overhead for memory verification for subsequent instruction executions, from 1.3 microseconds to about 65 nanoseconds.

This work poses the basis of an efficient and versatile toolbox of techniques to verify memory access validity in the context of embedded or real-time systems. The user can select the desired subset of memory objects to verify, which allows it to greatly reduce runtime overhead. He can also choose between different approaches to verify memory, with different compromises in terms of overheads and invasiveness. However, many aspects can still be optimized by future work, and a number of cases are not yet accounted for, for instance with the handling of protected arguments in system calls.

TABLE DES MATIÈRES

DÉDICACE	iii
REMERCIEMENTS	iv
RÉSUMÉ	v
ABSTRACT	vii
TABLE DES MATIÈRES	ix
LISTE DES TABLEAUX	xii
LISTE DES FIGURES	xiii
LISTE DES SIGLES ET ABRÉVIATIONS	xiv
CHAPITRE 1 INTRODUCTION	1
1.1 Définitions et concepts de base	1
1.2 Objectifs de recherche	4
1.3 Plan du mémoire	4
CHAPITRE 2 REVUE DE LITTÉRATURE	5
2.1 Zone rouge	5
2.2 Mémoire fantôme (Shadow Memory)	6
2.3 Étiquette de Pointeurs	8
2.4 Instrumentation	21
2.5 Conclusion de la revue de littérature	24
CHAPITRE 3 MÉTHODOLOGIE	26
CHAPITRE 4 ARTICLE 1 : TARGETED MEMORY RUNTIME ANALYSIS TECH- NIQUES FOR EMBEDDED SYSTEMS	28
4.1 Abstract	28
4.2 Introduction	29
4.3 Related Work	30
4.3.1 Redzones	30

4.3.2	Shadow Memory	31
4.3.3	Pointer Tainting	32
4.3.4	Instrumentation	33
4.3.5	DataWatch for Embedded Systems	34
4.4	Concepts	35
4.5	Implementation	37
4.5.1	Memory Protection	38
4.5.2	Disassembly	39
4.5.3	Single-step	40
4.6	Experimental study and Results	44
4.6.1	Experimental setup	44
4.6.2	Allocation Distribution	45
4.6.3	Instruction Distribution	46
4.6.4	Overhead	47
4.6.5	Selective Protection Based on the Object Size	53
4.7	Discussion	54
4.8	Conclusion	56
CHAPITRE 5 DISCUSSION GÉNÉRALE		58
5.1	Limitations et améliorations futures de Libpatch	58
5.2	Groupes de distribution d'allocations	58
5.2.1	505.mcf_r	58
5.2.2	519.lbm_r	59
5.2.3	526.blender_r	59
5.2.4	531.deepsjeng_r	60
5.2.5	541.leela_r	60
5.2.6	544.nab_r	60
5.3	Considérations pour le respect des objectifs	61
5.3.1	Minimiser le surcoût en temps	61
5.3.2	Minimiser le surcoût en mémoire	62
5.3.3	Minimiser l'invasivité de l'approche	63
5.4	Application des techniques à d'autres architectures	64
CHAPITRE 6 CONCLUSION		72
6.1	Synthèse des travaux	72
6.2	Limitations de la solution proposée	72
6.3	Améliorations futures	73

RÉFÉRENCES	74
----------------------	----

LISTE DES TABLEAUX

Table 4.1	Distribution of allocated object sizes for each benchmark	48
Table 4.2	Instruction Distribution	48
Table 4.3	Unique instructions accessing protected objects, total number of instruction executions accessing protected memory objects, and base execution time (without any runtime verification).	49
Table 4.4	Micro-benchmark results	52
Table 4.5	For each approach, the time relative to the base execution time is provided	53

LISTE DES FIGURES

Figure 2.1	Optimisation de la disposition de zones de mémoires pour réduire le nombre de zones rouges. Cette approche garantit, pour N objets, $N + 1$ zones rouges.	6
Figure 2.2	Représentation de mémoire fantôme	7
Figure 2.3	Représentation des étiquettes de pointeurs dans la représentation 64 bits de pointeurs	9
Figure 4.1	Distribution of allocated objects sizes (bytes) for xalancmbk	46
Figure 4.2	Number of allocated objects in time for xalancmbk	46
Figure 4.3	Number of allocated bytes in time for xalancmbk	47
Figure 4.4	Code snippet of the micro-benchmark	50
Figure 4.5	Assembly instructions of repeated memory access instructions in micro-benchmark	50
Figure 5.1	Distribution des tailles d'objets alloués pour mcf	59
Figure 5.2	Nombre d'objets alloués dans le temps pour mcf	60
Figure 5.3	Nombre d'octets alloués dans le temps pour mcf	61
Figure 5.4	Distribution des tailles d'objets alloués pour lbm	62
Figure 5.5	Nombre d'objets alloués dans le temps pour lbm	63
Figure 5.6	Nombre d'octets alloués dans le temps pour lbm	64
Figure 5.7	Distribution des tailles d'objets alloués pour blender	65
Figure 5.8	Nombre d'objets alloués dans le temps pour blender	66
Figure 5.9	Nombre d'octets alloués dans le temps pour blender	66
Figure 5.10	Distribution des tailles d'objets alloués pour deepsjeng	67
Figure 5.11	Nombre d'objets alloués dans le temps pour deepsjeng	67
Figure 5.12	Nombre d'octets alloués dans le temps pour deepsjeng	68
Figure 5.13	Distribution des tailles d'objets alloués pour leela	68
Figure 5.14	Nombre d'objets alloués dans le temps pour leela	69
Figure 5.15	Nombre d'octets alloués dans le temps pour leela	69
Figure 5.16	Distribution des tailles d'objets alloués pour nab	70
Figure 5.17	Nombre d'objets alloués dans le temps pour nab	70
Figure 5.18	Nombre d'octets alloués dans le temps pour nab	71

LISTE DES SIGLES ET ABRÉVIATIONS

IETF Internet Engineering Task Force

OSI Open Systems Interconnection

CHAPITRE 1 INTRODUCTION

Bien que le langage de programmation C ait été créé dans les années 1970, et le langage C++ vers 1985, les problèmes de corruption de mémoire restent toujours un problème de taille pour leurs applications. Ceci est dû au fait que ces langages laissent aux développeurs la gestion manuelle de la mémoire, à l'inverse de plusieurs autres langages qui auront, par exemple, un ramasse-miette qui s'occupera de cette tâche. Cette approche laisse plus de liberté aux développeurs, en ce qui a trait à ce domaine, mais cela ouvre inévitablement la porte à des erreurs qui peuvent causer de la corruption de données ou des vulnérabilités. Parmi les erreurs typiques de mauvaise gestion de mémoire, on peut nommer le débordement de mémoire (en anglais «overflow» ou «underflow») l'utilisation de la mémoire après libération («use-after-free») et plusieurs autres. Il existe plusieurs outils pour le débogage d'erreurs de mémoire, mais leur application pour les systèmes embarqués reste limitée, puisque les outils actuels sont trop invasifs ou occasionnent un coût additionnel en mémoire et en temps trop grand.

Le but de ce mémoire est donc de proposer une boîte à outils de débogage de mémoire, afin de réduire les coûts en mémoire et en temps, et donc de rendre l'application de tels outils possible sur des systèmes embarqués.

1.1 Définitions et concepts de base

Vérification du programme en temps d'exécution

La pratique de la vérification du programme en temps d'exécution (*Runtime Verification*), comme son nom l'indique, consiste premièrement en la collecte de données du programme, lors de son exécution. Utilisant ces informations récupérées, cette pratique consiste ensuite en l'analyse de ces informations, pendant l'exécution, pour détecter certains comportements, avec la possibilité d'intervenir sur l'exécution du programme.

Cette pratique s'inscrit dans un contexte de détection d'erreurs difficiles à détecter avec des moyens usuels, comme avec les jeux d'essais. Même en cas d'échec du programme, les informations collectées à l'aide de l'instrumentation en temps réel et son analyse pourront guider le développeur sur la bonne voie pour la résolution du problème.

Dans le cadre de cette maîtrise, la vérification du programme en temps d'exécution va permettre à l'outil d'inspecter chaque accès mémoire afin de vérifier leur validité. Pour pouvoir intervenir sur l'exécution du programme, il faudra utiliser l'instrumentation lors du temps

d'exécution.

Instrumentation

L'instrumentation consiste en la pratique d'insertion d'instructions dans le programme. L'instrumentation est souvent utilisée à fin de collecte d'information, où les instructions insérées serviront à la collecte d'informations de mémoire, de performance, etc. On distingue deux types d'instrumentation : statique et dynamique :

- L'instrumentation statique consiste en l'insertion des instructions dans le code source du programme lui-même. Par exemple, mettre des instructions d'impression («`print`») dans le code source, afin de récolter des données lors de l'exécution du programme, est une forme d'instrumentation statique.
- L'instrumentation dynamique insère les instructions lors de l'exécution même du programme.

Dans le but de collecte d'informations aux fins de débogage, le terme « journalisation » est souvent utilisé pour ce qui est de la collecte d'information. Toutefois, le terme « traçage » est préféré pour la collecte d'évènements de bas niveau, comme les évènements de noyau.

Dans le cadre de notre recherche, l'instrumentation sera une technique à utiliser dans le cadre de la vérification du programme en temps d'exécution («`Runtime Verification`»).

Allocation manuelle

Le langage de programmation C octroie aux développeurs la responsabilité d'effectuer les allocations et désallocations mémoire manuellement. Plusieurs fonctions de la librairie C sont donc disponibles à cet effet. Les fonctions principalement utilisées sont `malloc`, `calloc`, `realloc`, `memalign` et `free`. Pour l'allocation de mémoire d'une structure T , l'appel à `malloc` permet d'allouer l'espace nécessaire sur le tas (*heap*), mémoire réservée pour le programme. Le paramètre de retour représente l'adresse de la mémoire sur le *heap*, qui sera stockée dans un pointeur. Les accès et modifications de cette structure nouvellement allouée utilisent ce pointeur pour localiser cet objet dans la mémoire, utilisant les opérateurs «`*`» et «`->`» pour accéder et modifier les données enregistrées dans la structure. Ensuite, lorsque la mémoire associée à la structure n'est plus requise, un appel à `free` est fait pour libérer la mémoire pour le futur. Un échec à bien libérer la mémoire à temps peut laisser la mémoire dite « inatteignable », si tout pointeur ayant comme information l'adresse du bloc de mémoire est effacé. Ceci peut causer un défaut de mémoire de tas un peu plus tard. Cette erreur de

mémoire, nommée fuite de mémoire, s'inscrit malheureusement dans une longue liste d'erreurs liées à la mémoire.

Dans d'autres langages, un ramasse-miette est mis en place afin d'automatiquement gérer la mémoire. Il prend la forme d'un fil d'exécution de service («daemon») qui désalloue la mémoire sur le tas lorsqu'elle n'est plus atteignable, par exemple lorsque toute référence à son adresse est effacée du programme.

La détection des erreurs liées à l'allocation manuelle de la mémoire se révèlent particulièrement exigeante.

Invasivité

Dans ce mémoire, l'invasivité d'un outil est définie par sa perturbation occasionnée au programme. Un bon exemple est l'implémentation d'un correctif noyau, qui représente une perturbation importante à l'implémentation d'un système, demandant de recompiler le noyau Linux. On exclut donc ce terme du terme de surcoût en mémoire ou en temps d'un programme. Dans le contexte des systèmes embarqués, l'invasivité d'une approche nuit à son adoption, puisqu'il n'est pas toujours possible de recompiler le noyau du système par exemple.

Vérification mémoire

Voici quelques termes utilisés dans le contexte de vérification des limites de zones de mémoires.

Débordement

Le débordement de mémoire définit un type d'erreur où un accès mémoire, initialement prévu pour un certain objet, dépasse les bornes de cet objet. Un exemple classique est un accès à un index illégal d'un tableau, par exemple l'élément $n+1$ d'un tableau qui n'en contient que n . L'accès à cet « index » pointe alors à une autre zone mémoire.

Addressabilité

L'addressabilité d'une zone de mémoire décrit si cette dernière est bien allouée par le système, et donc disponible pour y écrire des données.

Validité

Dans le contexte de ce mémoire, la validité d'un accès mémoire se traduit par la vérification que cet accès mémoire soit bien fait à la bonne zone mémoire. Il vérifie donc que l'accès mémoire prévu pour un objet soit bien effectué dans ses bornes, et non dans une zone de mémoire inaccessible ou une zone de mémoire appartenant à un autre objet.

Fuite

Une fuite de mémoire est un type de problème de mémoire récurrent. Dans le langage C/C++, une allocation dynamique de mémoire laisse au développeur un pointeur, une référence à cette zone de mémoire allouée. Négliger la libération de cette zone de mémoire, lorsqu'elle n'est plus utile (qui peut être causé par la perte de ce pointeur), constitue une fuite de mémoire. La fuite de mémoire est un problème qui ne peut qu'augmenter au fil du temps d'exécution.

1.2 Objectifs de recherche

Les travaux effectués dans ce mémoire tentent de répondre à la question suivante :

Est-il possible de développer des outils de détection d'erreurs de mémoire en temps réel qui sont applicables de façon réaliste aux systèmes embarqués en matière de surcoût ?

Les objectifs de recherche suivants sont donc extraits de la question, résultants en ces 3 objectifs :

- Minimiser le surcoût en temps et en mémoire
- Assurer une détection d'erreurs de mémoire adéquate
- Minimiser l'invasivité de l'approche sur le système

1.3 Plan du mémoire

Le chapitre 2 produit un portrait des différents outils existants de débogage de mémoire, avec les différentes techniques qui leur sont communes. Ensuite, le chapitre 3 présente la méthodologie utilisée dans le cadre du processus englobant la conception et la collecte des résultats. Le chapitre 4 présente l'article « Targeted Memory Runtime Analysis Techniques for Embedded Systems », qui résume les techniques développées dans le cadre de cette maîtrise. Le chapitre 5 détaille des résultats additionnels omis de l'article. Finalement, le chapitre 6 explore les limitations actuelles et propose des améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre peint un portrait de l'état de l'art des outils de débogage mémoire. Ces différents outils utilisent des techniques similaires pour la détection d'erreurs de mémoire. Ces techniques seront donc résumées, avec quelques exemples d'outils utilisant ces techniques, puis tous les outils sont détaillés avec leurs innovations et leurs surcoûts de mémoire et de performance. Ensuite, différents outils d'instrumentation dynamique, nécessaire pour la plupart des outils de débogage mémoire présentés, seront détaillés.

2.1 Zone rouge

Les zones rouges sont des zones de mémoire marquées comme étant non adressables, pour détecter les erreurs de débordement de zone de mémoire. En pratique, ces zones de mémoire sont réservées lors de l'allocation d'un objet par une implémentation spéciale des fonctions d'allocation. Ces zones rouges sont définies directement après et/ou avant la zone de mémoire allouée pour l'objet. Dans ce cas, l'outil de débogage peut détecter un accès hors limite allant jusqu'à la taille même de ces zones rouges. Un dilemme s'impose donc sur la taille de cette zone rouge. Une taille plus grande permet à des accès hors limite plus grands d'être détectés sans problème, mais cela augmente significativement le surcoût de mémoire de tas. À l'inverse, une taille plus petite a un surcoût de mémoire plus petit, mais il ne garantit pas la détection d'accès mémoires hors limite aussi grands.

Par exemple, supposons qu'un vecteur d'entiers est accédé de manière linéaire en incrémentant de 1 son index à chaque tour de boucle. Si le calcul de fin de boucle est erroné et on tente d'accéder un index qui dépasse de 1 la taille du vecteur, la moindre zone rouge, de 1 octet ou plus, pourra détecter ce débordement à l'entier tout de suite après le vecteur. Par contre, si on accède 1 entier sur 4 dans le vecteur, un débordement pourrait accéder une case qui n'est pas immédiatement adjacente au vecteur. Dans ce cas, la zone rouge devrait avoir une taille supérieure à 3 entiers afin de détecter ce débordement. Le même principe s'appliquerait si on avait un vecteur de structures de données et on accédait un seul champ dans chaque structure de donnée du vecteur.

L'outil *ElectricFence* [1] est un outil simple n'utilisant que cette technique de zone rouge afin d'éviter, entre autres, les erreurs de débordement de mémoire. L'outil arrondit chaque allocation à la page près, puis rajoute une page de mémoire inaccessible directement après (ou avant) l'objet alloué.



FIGURE 2.1 Optimisation de la disposition de zones de mémoires pour réduire le nombre de zones rouges. Cette approche garantit, pour N objets, $N + 1$ zones rouges.

Lorsque l'outil entoure chaque objet alloué par deux zones rouges, une optimisation peut être employée afin de réduire le nombre total de zones rouges. En effet, en supposant deux objets alloués un à la suite de l'autre sur la mémoire de tas, il n'y a que 3 zones rouges requises, puisqu'une seule zone rouge peut séparer les 2 objets alloués. Ainsi, supposant N le nombre d'objets alloués, le nombre total de zones rouges requis passe de $2N$ à $N+1$. La figure 2.1 montre une représentation spatiale de la mémoire implémentant cette optimisation. *AddressSanitizer* [2] utilise cette optimisation afin de garantir $N+1$ zones rouges et ainsi réduire le surcoût de mémoire.

2.2 Mémoire fantôme (Shadow Memory)

La mémoire fantôme, dite *Shadow Memory*, est une technique visant à représenter la mémoire complète d'un programme par un sous-ensemble de mémoire dit *shadow*. Ce sous-ensemble contient donc certaines informations sur tout le programme. Cependant, l'implémentation du sous-ensemble diffère beaucoup entre les différents outils qui utilisent cette technique. La figure 2.2 présente une mémoire fantôme qui représente chaque zone de mémoire de T bits par A bits pour sa mémoire fantôme associée.

Le premier élément à considérer est le type d'information à entreposer dans la mémoire fantôme. Plusieurs outils de débogage de mémoire utilisent cette technique afin de stocker l'information sur l'état de la mémoire. De cette manière, l'outil peut s'assurer qu'aucun accès ne sera fait sur de la mémoire non allouée. Certains outils comme *AddressSanitizer* [2] n'indiquent que si une zone de mémoire est adressable ou non. D'autres outils, comme *Memcheck* [3, 4] et *Dr. Memory* [5], détaillent cette information en indiquant si la zone de mémoire est définie, c'est-à-dire initialisée. En effet, une zone de mémoire adressable est allouée, mais elle n'est pas forcément initialisée avec une valeur.

Ensuite, il faut considérer la taille même de la mémoire fantôme. Un équilibre doit être

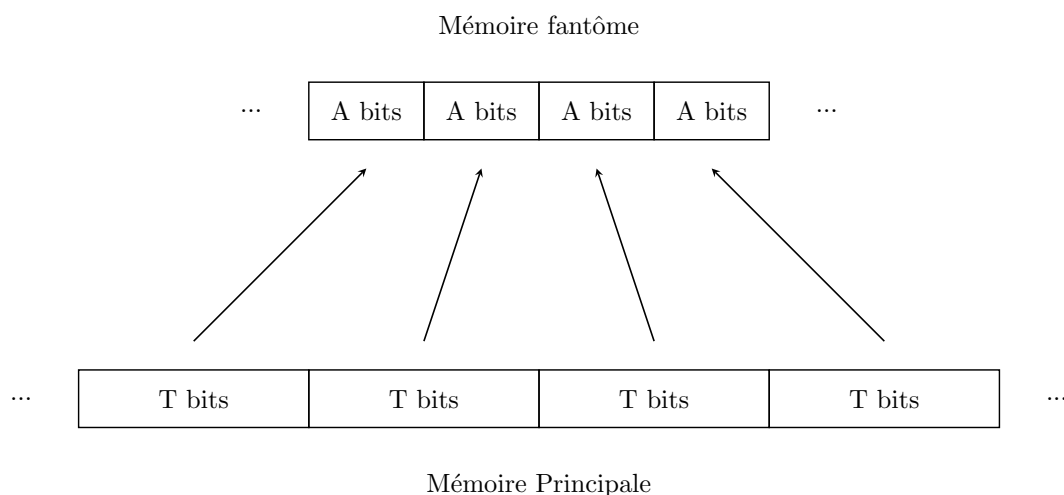


FIGURE 2.2 Représentation de mémoire fantôme

établi entre la taille et la précision de ce sous-ensemble. En effet, une implémentation naïve pourrait considérer enregistrer les informations de débogage bit par bit. Ceci donne à l'outil une précision au bit près, mais il souffre d'un surcoût de mémoire plus conséquent. À l'inverse, stocker l'information pour un sous-ensemble, pour chaque 4 octets de la mémoire, réduit le surcoût d'un facteur de 32, mais sa précision est grandement diminuée. En effet, il est possible qu'une certaine portion du 4 octets de mémoire représenté ne soit pas tout adressable, alors que l'autre portion l'est.

Une attention particulière doit être portée à la gestion de la mémoire fantôme. Lors de l'allocation d'un nouvel objet, une implémentation spéciale des fonctions d'allocation/désallocation (`malloc`, `calloc`, `realloc`, `free`, etc.) permet la mise à jour de la mémoire fantôme.

Bien que l'outil puisse détecter un accès hors limite, dans la mesure où l'accès se fait sur de la mémoire non adressable, la plupart des outils implémentant cette technique ne conservent pas d'information pour différencier les différents objets alloués. En conséquence, sans autre considération, il n'y aurait pas de différence entre un accès dans les limites de l'objet et un accès hors limite, mais qui tombe dans les bornes d'une autre zone de mémoire adressable.

Pour remédier à ce problème, bon nombre d'outils utilisent cette technique en conjonction avec des zones rouges. En utilisant ces zones rouges, les outils héritent du même dilemme concernant la taille des zones rouges à employer.

La plupart des erreurs de débordement de mémoire tampon sont linéaires, ce qui implique qu'ils sont détectables facilement avec cette technique de zones rouges. Cependant, un certain

nombre de ces erreurs sont dites non linéaires, ce qui peut poser problème pour sa détection. Afin de vérifier l'état de la mémoire fantôme avant chaque accès mémoire, il faut avoir une instrumentation avant chacun de ces accès.

2.3 Étiquette de Pointeurs

Cette prochaine technique consiste à entreposer des informations dans le pointeur de l'objet lui-même. Ces informations sont utilisées pour effectuer le contrôle des bornes lors d'accès mémoire.

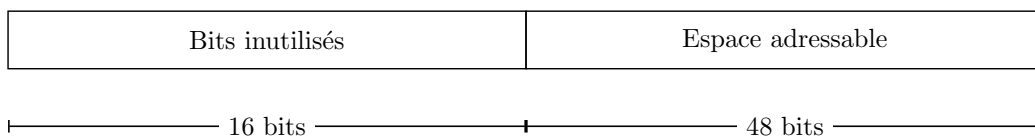
Pour stocker des données dans le pointeur même de l'objet alloué, plusieurs méthodes peuvent être employées. Une de ces méthodes, possible sous l'architecture `x86_64`, prend avantage du fait que les bits 47 à 63 sont inutilisés. Cependant, un accès mémoire fait à une adresse dont la valeur de ses bits 47 à 63 n'est pas les mêmes (soit tous avec une valeur de 1, soit tous avec une valeur de 0), génère un signal `SIGSEGV` ou `SIGBUS`. Cette approche n'est évidemment pas disponible pour des systèmes 32 bits, qui ne possèdent peu ou pas de bits inutilisés, mais il n'y a que très peu de ces systèmes aujourd'hui.

Une première considération d'implémentation doit être portée sur la nature de l'information stockée. Plus l'information enregistrée est détaillée, plus de bits seront requis dans le pointeur pour représenter cette information, ce qui peut limiter l'espace adressable.

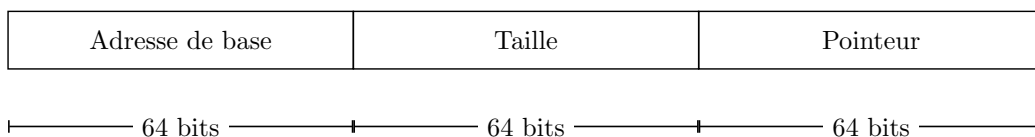
Une première option est d'associer chaque allocation à une étiquette de petite taille. Bien que plusieurs objets auront la même étiquette, cela requiert moins de bits du pointeur à occuper. L'outil *Hardware-Assisted AddressSanitizer* [6], spécifique à l'architecture `AArch64`, écrit une étiquette de 8 bits. Cet outil utilise aussi la mémoire fantôme afin d'associer chaque zone de mémoire de 16 octets avec une des étiquettes. Enfin, lors de l'accès mémoire, l'étiquette du pointeur ainsi que la zone mémoire accédée peuvent être comparées pour valider l'accès. Cependant, il y a toujours un risque d'un faux négatif, dans ce cas-là d'environ $1/256$, ce qui reste relativement petit. Dans le cas de l'extension matérielle de *SPARC ADI*, qui utilise une étiquette similaire, l'étiquette prend 4 bits, ce qui monte le risque à $1/16$, soit 6%.

À l'inverse, une autre option à considérer pourrait être d'enregistrer toutes les informations nécessaires au contrôle des limites de zones adressables dans le pointeur. Ceci impliquerait donc d'y mettre l'information sur l'adresse de base, et la taille de l'objet. Pour avoir assez d'espace, il faut avoir recours à des soi-disant *fat pointer*, ou pointeurs gras. Ces pointeurs sont conçus pour occuper trois mots de 64 bits au lieu d'un seul, afin de pouvoir y enregistrer l'information nécessaire, ajoutant l'adresse de base de la zone de mémoire, ainsi que sa taille. Cette représentation de pointeurs sur mesure implique sans doute un surcoût en mémoire du

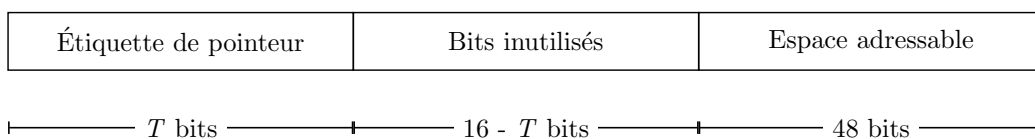
a) Représentation des pointeurs sous x86_64



b) Représentation possible d'un pointeur gras de 3 mots (3 octets)



b) Étiquette de pointeur placée dans les bits inutilisés



c) Étiquette de pointeur placée dans l'espace adressable

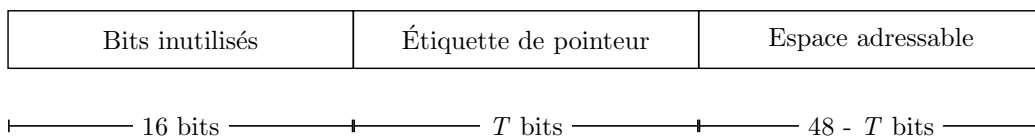


FIGURE 2.3 Représentation des étiquettes de pointeurs dans la représentation 64 bits de pointeurs

fait que chaque pointeur occupe désormais 3 fois plus de mémoire.

Cependant, contrairement à la technique précédente, cette méthode a l'avantage de fournir ces informations précises sur leur taille. Ainsi, peu importe si l'erreur de débordement de mémoire est linéaire ou pas, elle sera toujours détectée si elle ne respecte pas la condition suivante :

$$base < mem < base + taille$$

Où *base* représente l'adresse de *base* de la zone allouée, *taille* représente sa taille, et *mem* représente le pointeur utilisé pour l'accès mémoire, en ne prenant uniquement les 64 bits qui représentent l'adresse.

Bien que *Cyclone* [7] soit un dialecte du langage C, et non un outil de débogage mémoire, il introduit tout de même les pointeurs gras afin de combattre les erreurs de débordement de zones de mémoire.

Cependant, une telle approche réduit aussi sa compatibilité. Une telle représentation pour les pointeurs demande un certain effort à intégrer. Plusieurs avancements ont été faits pour garder la même représentation du pointeur tout en gardant sa capacité de fournir les informations sur l'adresse de base et la taille des zones de mémoires allouées. L'approche *Low-Fat Pointers* [8,9], originalement proposée à un niveau matériel [10], a été adaptée pour utilisation logicielle, en organisant la mémoire de tas intelligemment afin d'extraire ces informations. Les informations relatives à l'adresse de base et la taille des objets peuvent être aussi stockées dans un espace disjoint, mais cela requiert d'établir une correspondance entre le pointeur et ses métadonnées. *MESH* [11] utilise une table de métadonnées pour enregistrer les informations de bornes d'allocations, en utilisant l'étiquette comme index pour récupérer les bonnes données.

Memcheck

Memcheck [3,4] est un outil de la suite *Valgrind* qui se spécialise dans la détection d'erreurs mémoires courantes dans les programmes C/C++. Cet outil compte pour plus de 80 % de l'utilisation de la suite d'outils de *Valgrind*.

La mémoire fantôme de *Memcheck* stocke 2 types d'information :

- Pour chaque octet de mémoire, un bit A (A pour Adressabilité) fantôme indique si l'octet est adressable ou non. Cette information permet à *Memcheck* de repérer les débordements de mémoire.
- Pour chaque bit de mémoire, un bit V (V pour Validité) fantôme indique si le bit est défini. Ceci permet une précision au bit près.

Puisqu'il est inutile d'examiner les bits de validité pour un octet si son bit d'adressabilité pointe au fait qu'il est inadressable, cela génère donc 257 états pour chaque octet du programme, un état indiquant que l'octet est inadressable et les 256 autres états détaillant la validité de chacun des 8 bits de l'octet.

La mémoire fantôme est implémentée comme une table à deux niveaux : le premier niveau étant appelé *primary map (PM)*, est constitué d'un tableau de 64K entrées qui contiennent des pointeurs au deuxième niveau, le *secondary map (SM)*, qui contient donc la mémoire fantôme pour un bloc de 64 Ko de mémoire. Un sous-ensemble du deuxième niveau de la table est appelé le *distinguished secondary map (DSM)*, et marque la mémoire qu'il représente

comme non adressable, sans possibilité de modification, du moins sans changer le pointer du PM à un SM.

L'implémentation de la mémoire fantôme avec son instrumentation est faite avec une technique simple, mais lente (M0), à laquelle on peut appliquer jusqu'à 4 optimisations (M1-M4) pour réduire le temps de surcoût.

La première optimisation modifie la fonction qui sert à charger plusieurs octets de la mémoire fantôme en même temps, assumant qu'ils soient tous alignés et adressables.

Dans certains cas, toute la mémoire fantôme dans une certaine zone doit être mise à la même valeur. La deuxième optimisation consiste donc à combiner 3 améliorations sur ces opérations sur la mémoire fantôme :

- Les octets de la mémoire fantôme à changer peuvent être vectorisés.
- Lorsque la valeur d'une certaine zone de mémoire fantôme SM doit être changée à la valeur *non adressable*, le remplacer par une zone DSM non adressable s'occupe de tout ce changement en une opération.
- En rajoutant des zones de mémoire fantôme DSM pour les valeurs d'adressabilité *Non Défini* et *Défini*. Ces zones de mémoire fantôme DSM ont des cas d'utilisation bien utiles, puisqu'un DSM marqué défini peut abriter des régions de données en lecture seule ou des longs segments de codes.

Puisque les changements des pointeurs de mémoire de pile sont souvent associés à des tailles constantes de 4, 8, 12, 16 ou 32 octets, les fonctions changeant la mémoire fantôme dans une zone (range-setting operations) sont optimisées pour ces tailles spécifiques.

Finalement, la dernière optimisation redéfinit la distribution des bits fantômes A et V, ne nécessitant que deux bits AV pour chaque octet.

Ces optimisations permettent de réduire le surcoût associé à plusieurs des opérations couramment utilisées dans le cadre de la gestion de la mémoire fantôme.

Memcheck faisant partie de la suite d'outils de *Valgrind*, cette dernière sert de cadre pour effectuer l'instrumentation binaire dynamique, lors de l'exécution du programme. L'instrumentation dynamique faite à l'exécution est un avantage dans notre contexte, puisqu'il demeure moins invasif et nécessite moins de configuration au préalable.

L'approche avec les différentes optimisations résulte en un facteur de surcoût de temps de 22,2, et un surcoût mémoire d'un facteur de 1,26. Ce surcoût important n'est malheureusement pas compatible avec l'utilisation pour des systèmes embarqués en temps réel.

Memcheck permet de détecter un grand nombre de types d'erreurs mémoire. Voici une liste

résumant ces différents types d'erreurs détectées par *Memcheck* [12] :

- Fuites de mémoire
- Débordement de mémoire en lecture/écriture
- Utilisation de valeurs non initialisées
- Utilisation de valeurs non adressables ou non initialisées dans des appels système
- Désallocations illégales
- Fonction de désallocation de bloc inappropriée
- Blocs de source et de destination qui se chevauchent (pour des fonctions comme `strncpy`)

La popularité de *Memcheck* fait de cet outil une source d'inspiration pour un grand nombre des outils détaillés dans cette section.

Dr. Memory

Dr. Memory est un outil de vérification de mémoire qui détecte les erreurs liées à la mémoire sur **Windows** ainsi que sur **Linux**.

Pour chaque octet du programme, 2 bits lui sont alloués en tant que mémoire fantôme. Ces 2 bits permettent d'indiquer un des 3 états de mémoire dans lequel se trouve cet octet : non adressable, non initialisé et défini.

Dr. Memory utilise *DynamoRIO* pour ses besoins d'instrumentation. Ce système est utilisé afin d'insérer les instructions d'instrumentation en temps d'exécution.

Deux principales implémentations d'instrumentation sont utilisées en complémentarité, l'une surnommée *fastpath*, pour les actions où la performance doit être optimisée à l'aide de code machine fabriqué à la main, et *slowpath*, pour des actions moins importantes qui ne requièrent pas une telle attention, comme les instructions complexes et les rapports d'erreur.

Bon nombre d'optimisations sont apportées au traitement d'accès mémoire qui se retrouvent dans le *fastpath* comme dans le *slowpath* :

- L'adressabilité d'une région de mémoire est vérifiée avec une table de correspondance.
- Comme *Memcheck*, des régions spéciales de mémoire sont dédiées à la mémoire fantôme ayant la même valeur (*non-adressable* pour l'espace non alloué ou *défini* pour les bibliothèques). Ces régions sont définies en lecture seule afin d'éviter de devoir effectuer des vérifications lors d'écritures.
- Le nombre de débordements *spill* de registres, lorsque la valeur d'un registre est écrite en mémoire afin de laisser la place pour écrire une autre valeur dans ce même registre, est réduit en analysant chaque bloc de base, partageant ainsi les registres qui doivent

être sauvegardés temporairement *spill*.

- L'utilisation d'un encodage spécial de pile d'appels binaire réduit considérablement le surcoût de temps lors d'impression de texte dans la mémoire tampon.
- Les différentes fonctions de recherche et de copie de chaînes de caractères sont utilisées avec une boucle pour chaque mot.
- Pour calculer l'adresse de la mémoire fantôme pour une adresse cible, la distance de déplacement est enregistrée, plutôt que l'adresse absolue, réduisant l'espace requis pour la table de traduction.

Sur la suite de tests de performance *SPEC CPU 2006* [13], *Dr. Memory* a un surcoût de temps d'un facteur de 10,2 en moyenne, ce qui était environ 2 fois meilleur que l'outil *Memcheck*, lorsque l'article a été publié. Cependant, le surcoût demeure encore bien trop élevé pour toute application dans les systèmes embarqués en temps réel.

AddressSanitizer

AddressSanitizer (ASAN) [2] est un outil de débogage d'erreurs de mémoire développé par Google en 2012. Il emploie un bon nombre de techniques similaires à *Memcheck*, comme la mémoire fantôme, mais il optimise ces techniques existantes et utilise une autre approche afin d'assurer de meilleures performances.

La mémoire fantôme d'*ASAN* alloue un octet de mémoire fantôme pour chaque 8 octets de la mémoire du programme. Cet octet signale l'accessibilité de chacun des 8 octets de la mémoire. Ceci étant dit, cette technique garantit donc un surcoût de mémoire équivalent à un huitième de la mémoire de l'application. Contrairement à *Dr. Memory* ou *MemCheck*, il n'y a pas de distinction entre un octet inadressable et un octet non défini.

Lors d'accès mémoire, *ASAN* calcule les octets de mémoire fantôme correspondants à l'accès mémoire, puis vérifie la valeur de ce(s) octet(s) pour valider l'accès mémoire. Pour s'assurer de pouvoir faire ces vérifications pour chaque accès mémoire, *ASAN* insère l'instrumentation lors de la compilation du programme, ce qui nécessite une recompilation complète du programme. Une telle recompilation peut ne pas être possible pour les systèmes en temps réel et les systèmes embarqués.

Pour empêcher les débordements de mémoire, *ASAN* entoure les allocations de mémoire par deux zones rouges. Pour n objets, *ASAN* garantit $n+1$ zones rouges. Ces deux zones de mémoire ont par défaut une taille de 128 octets, bien que leur taille puisse être ajustée. Un certain équilibre doit être atteint, puisqu'une taille de mémoire zone rouge plus grande implique un surcoût en mémoire plus important, alors qu'une taille de mémoire zone rouge

trop petite nuirait à la capacité de détecter les débordements de mémoire.

Pour détecter les erreurs d'utilisation de mémoire après désallocation, chaque zone de mémoire nouvellement désallouée est marquée inaccessible pour un certain temps, dans une zone de quarantaine. Cette zone de quarantaine est implémentée sous la forme d'une queue premier entré, premier sorti de 256 méga-octets.

Pour s'assurer de faire ces allocations spéciales de mémoire, une bibliothèque d'exécution spéciale s'occupe de l'allocation de la mémoire fantôme et des zones de mémoire interdites.

L'outil garantit la présence d'aucun faux positif, bien qu'il puisse y avoir de faux négatifs avec, entre autres, les erreurs de débordement de mémoire dépassant la taille des zones rouges.

Cet outil obtient un surcoût de temps d'un facteur de 1,71, et une mémoire 3,34 fois plus élevée. Le surcoût de mémoire provient principalement de l'ajout de zones rouges. Comme mentionné auparavant, il réussit ces diminutions de surcoût par son instrumentation à la compilation, ce qui est toutefois plus invasif et peut être difficile à implémenter dans le contexte des systèmes embarqués et en temps réel. Le fait que *ASAN* ne stocke pas d'information, pour distinguer les zones de mémoire adressables et définies des zones de mémoire adressables et non définies, limite le type d'erreurs qu'il peut détecter. Contrairement à *Memcheck*, il ne peut pas détecter efficacement l'utilisation de valeurs non définies.

Low-Fat Pointers

Cette approche [8] développée par Gregory J. Duck, et Roland H. C. Yap, détaille une approche utilisée pour garder les avantages des pointeurs gras tout en restaurant la représentation 64 bits des pointeurs. Pour ce faire, la mémoire de tas est séparée en régions de 4 Go. Ainsi, la région mémoire correspondante du pointeur peut être déterminée avec la formule suivante :

$$index(p) = p >> 32$$

Chaque région (sauf la première région, qui est réservé pour les informations propres au programme comme `.data` et `.text`) est associée à une certaine taille d'allocation. Les tailles d'allocation associées peuvent être changées lors de l'implémentation, mais le prototype utilisé a une taille associée de 16 octets pour la première zone, puis 16 octets de plus pour les zones subséquentes, puis 2 fois plus d'octets lorsque la taille associée atteint les 8 Ko. Chaque allocation est arrondie à une de ces tailles (à la hausse), puis mise dans sa région de mémoire correspondante. En sachant la taille d'allocation représentée par la région de

mémoire appartenant au pointeur, on peut donc déduire sa taille. Son adresse de base peut être déduite de l'alignement de la région de mémoire à laquelle il est associé.

L'instrumentation au temps de compilation, à l'aide d'un plugiciel de *LLVM*, s'occupe de mettre à jour les méta-informations sur les pointeurs et la vérification du respect des limites des zones de mémoires. Cette dernière est assurée en instrumentant à la fois chaque opération *load* ou *store*, et chaque opération d'arithmétique de pointeur, en assumant que le pointeur résultant est utilisé ensuite :

- Lors de forçage de type du pointeur à un entier
- Lorsque le pointeur est stocké à la mémoire
- Lorsque le pointeur est passé à une fonction
- Lorsque le pointeur est utilisé comme valeur de retour de la fonction

L'instrumentation de chaque opération d'arithmétique de pointeur permet à l'outil de s'assurer que chaque pointeur reste dans son espace d'allocation. Dans ce cas, l'outil évite une situation fâcheuse où une opération d'un pointeur dans la première région de mémoire se solde en un pointeur dans la deuxième région de mémoire, ce qui aurait comme conséquence de fausser les opérations de récupération des informations sur la taille et l'adresse de base. Ceci évite aussi le cas où un débordement de pointeur fait pointer vers un autre objet, et évite conséquemment le recours à des zones rouges.

Trois différentes optimisations sont proposées afin de réduire le surcoût de temps. La première optimisation consiste à remplacer l'opération de division requise pour déterminer l'adresse de base du pointeur par une opération de multiplication, tirant avantage du fait que les tailles d'allocations sont constantes et prédéterminées avant l'exécution du programme. Cependant, l'arithmétique en virgule fixe utilisée pour convertir cette opération de division peut subir des imprécisions, ce qui peut provoquer des faux négatifs lors du contrôle des limites de zones mémoire pour les allocations de plus de 2 Mo. Pour y remédier, un remplissage de 4 Ko est ajouté aux allocations de plus de 2 Mo, afin que les erreurs de précision n'impactent que ce paquet de 4 Ko.

La deuxième optimisation restreint les tailles d'allocations à des puissances de deux. Ainsi, l'opération requise pour obtenir la taille de l'objet à partir du pointeur n'est qu'un simple bit de masque.

La troisième optimisation restreint les vérifications de limites de zones de mémoire à des opérations d'écriture de mémoire.

En testant la suite de test de performance *SPEC 2006* [13], le surcoût de mémoire est de 1,03 et le surcoût de temps est de 2,5. Cependant, avec les optimisations, le surcoût de temps

peut être réduit jusqu'à 1,13, tout en augmentant le surcoût de mémoire jusqu'à un facteur de 1,11 seulement.

L'approche avait été conçue initialement pour la mémoire de tas, mais elle a été étendue pour la protection des objets dans la mémoire de pile [9]. Ils définissent donc un nouvel allocateur, nommé le *low-fat stack allocator*. Les régions de mémoire associées aux différentes tailles d'allocations sont maintenant divisées en 2 sous-sections : une première sous-section pour les objets de la mémoire de tas, étant alloués à partir du début de la région, alors que la deuxième sous-section pour les objets de la mémoire de tas sont alloués à partir de la fin de la région. On se retrouve donc avec plusieurs sous-sections de mémoire de pile, ce qui est possible avec la réflexion de pointeurs d'objets de mémoire de pile, une technique conçue qui permet d'allouer la mémoire de pile dans ces sous-régions, ce qui fait de ces pointeurs *low-fat*, et leur garantit la protection de mémoire décrite plus haut. L'article prévoit des régions de 32 Go, avec 28 Go pour la sous-section de la mémoire de tas et 4 Go pour la sous-section de la mémoire de pile.

Trois autres optimisations sont proposées pour réduire les surcoûts liés à cette extension. Une optimisation sur l'allocation d'objets de mémoire fixe permet de réduire le nombre d'instructions requis pour l'allocation spéciale. L'allocateur *low-fat* pour objets de mémoire de pile peut aussi être ignoré entièrement, si son pointeur n'est pas passé à une autre fonction, et que tous les accès à cet objet peuvent être déterminés statiquement comme étant dans les limites de l'objet. Cette optimisation prend avantage de *LLVM*, afin de déterminer ces conditions. Les dernières optimisations optimisent la vérification des bornes, en utilisant une méthode plus rapide, en supprimant les vérifications redondantes, et en restreignant les vérifications à un sous-ensemble des opérations, comme les écritures de mémoire.

La suite de test de performance *SPEC 2006* [13] est utilisée à nouveau afin de tester la performance de l'approche avec cette extension. Le surcoût de temps atteint un facteur moyen de 1,63 et le surcoût de mémoire atteint un facteur de 1,35. Les trois optimisations mentionnées plus haut réduisent les surcoûts de temps et de mémoire à des facteurs de 1,17 et 1,15 respectivement. Par contre, allouer une certaine section pour une taille d'allocation peut poser problème si un très grand nombre d'allocations de même taille est fait, du fait que la section de 4 Go dédiée pourrait ne pas suffire pour les stocker. Les allocations très grandes (plus de 1 Go) ne peuvent aussi pas être protégées par cette approche.

SoftBounds

SoftBounds [14] se décrit comme un outil de transformation lors de compilation, pour diminuer les erreurs de mémoire. *SoftBounds* stocke les informations sur la taille et l'adresse de base de chaque objet dans un espace disjoint. Il s'assure ensuite de récupérer ces informations dans une table de correspondance. L'espace disjoint utilisé pour l'entreposage des métadonnées est implémenté soit comme une table de hachage, ou un espace fantôme. La table de hachage requiert une entrée supplémentaire pour un tag et plus d'instructions pour récupérer les métadonnées, alors que l'espace fantôme requiert moins d'instructions, mais restreint 2 bits de la taille de l'espace adressable virtuel.

Le prototype présenté utilise 23 tests de performance d'un mélange de tests de performance des suites *SPECint*, *SPECfp* et *Olden* [15]. Il présente un surcoût de temps d'un facteur moyen de 1,93 pour la table de hachage ou 1,67 utilisant l'espace fantôme. Cependant, un test séparé est fait, ne faisant que le contrôle des bornes lors d'opérations *store*. L'article explique que les bogues liés avec les opérations de stockage sont plus insidieux, ce qui explique cette priorité afin d'améliorer la performance. Le surcoût en performance est donc réduit à un facteur de 1,54 et 1,22, respectivement. Il présente finalement un surcoût de mémoire d'un facteur moyen de 1,87 avec la table de hachage, ou 1,64 avec l'espace fantôme.

Hardware-Assisted AddressSanitizer

Hardware-Assisted AddressSanitizer (HWASAN) [6] est une variante de l'outil AddressSanitizer spécifique à l'architecture **AArch64**. Il tire avantage du champ de données *Top Byte Ignore* du registre TCR_EL pour permettre aux 8 bits, de 55 à 63, d'ignorer l'exigence précédemment mentionnée que les 16 bits inutilisés (bits 47 à 63) doivent tous avoir la même valeur. Ainsi, il peut associer une étiquette à chaque pointeur d'objet alloué, sans que cela ne perturbe l'exécution en générant des interruptions.

Pour chaque 16 octets de mémoire, une étiquette de 1 octet lui est associée lors de l'allocation d'une nouvelle zone de mémoire. Cela requiert que chaque allocation soit alignée sur 16 octets, pour que chaque zone de mémoire de 16 octets n'ait qu'une seule étiquette associée.

L'outil utilise *LLVM* aux fins d'instrumentation. L'instrumentation est donc faite lors de la compilation du programme.

Un avantage considérable de cette approche est que l'utilisation de zones rouges et de quarantaines de *AddressSanitizer* n'est plus nécessaire, ce qui réduit de façon considérable le surcoût de mémoire. En effet, comme mentionné précédemment, les zones rouges représentent une

partie considérable du surcoût de mémoire d'un facteur de 3,34. Le surcoût de mémoire est donc estimé en fonction de la mémoire fantôme, prenant 6,25% de mémoire additionnelle, avec un certain surplus pour l'alignement. L'outil n'a besoin que de comparer l'étiquette reliée à la zone de mémoire (stockée dans la mémoire fantôme) avec l'étiquette contenue dans le pointeur concerné par l'accès mémoire.

L'étiquette ne pouvant avoir que 256 valeurs différentes, la même étiquette peut être associée à plusieurs objets, ce qui implique un taux de faux négatif de 0,39%. Cela s'applique à la détection d'erreurs de débordement de limites de zone de mémoire. Pour la détection d'accès aux zones de mémoire désallouées, il est facile d'avoir une valeur d'étiquette «invalid» réservée, à laquelle aucun pointeur ne correspondra. La méthode de détection d'erreurs de mémoire est donc dite probabiliste. Bien sûr, un inconvénient additionnel de cet outil est sa dépendance à l'architecture AArch64 pour sa fonctionnalité *Top Byte Ignore*.

SPARC Application Data Integrity (ADI)

SPARC ADI [6] est une extension matérielle supportée sur les processeurs SPARC M7/M8 sur le système d'exploitation Solaris [16]. Dans une approche similaire à *Hardware-Assisted AddressSanitizer*, chaque 64 octets de mémoire est associé avec une étiquette de 4 bits. Ce même octet peut être stocké dans les hauts bits inutilisés des pointeurs. Lors d'accès mémoire, l'étiquette de la zone de mémoire accédée et l'étiquette du pointeur sont comparées, une divergence occasionnant un message d'erreur et un signal SIGSEGV.

Comme pour l'outil *HWASAN*, l'étiquette ne pouvant avoir que 16 valeurs différentes, la même étiquette peut être associée à plusieurs objets, ce qui implique un taux de faux négatif de 6,25%. La méthode de détection d'erreurs de mémoire est donc aussi dite probabiliste. Bien que la fonctionnalité ADI de SPARC ait à sa disposition 2 fois moins de mémoire pour y enregistrer l'étiquette, cela représente une baisse de précision considérable.

MESH

MESH [11] est un outil de débogage de mémoire qui utilise aussi des étiquettes de pointeurs. Cependant, afin d'enregistrer les informations relatives aux allocations mémoire sans avoir recours à des pointeurs gras, il utilise une table de métadonnées. Les informations enregistrées pour chaque allocation sont sa borne inférieure, sa borne supérieure, ainsi qu'un drapeau de validité. L'étiquette du pointeur sert donc comme index dans la table de métadonnées.

Lors de l'allocation d'un objet, ses informations de bornes sont stockées dans la table à un

index précédemment libre, le drapeau de validité est changé à *vrai*, et son index est ajouté au pointeur retourné en tant qu'étiquette. Lors de la désallocation d'un objet, le drapeau de validité est changé à faux, et les valeurs de bornes sont réinitialisées à 0. Ce drapeau permet donc à l'algorithme d'identifier les entrées libres dans la table de métadonnées. L'étiquette est donc unique pour chaque objet alloué. Cela implique que le nombre d'objets alloués protégés par cette approche ne peut pas dépasser $2^T - 1$, où T représente le nombre de bits que prend l'étiquette. Tous les objets protégés de cette façon, ainsi que la table de métadonnées, sont conservés dans un segment de mémoire dédié, afin d'assurer le support des objets qui ne sont pas protégés. L'étiquette 0 est donc inutilisée, pour ne pas perturber les accès mémoire à des objets non protégés.

Bien que MESH ne puisse être utilisé que pour protéger les zones de mémoire de tas, il est suggéré de le combiner avec des approches qui protègent les allocations de zones de mémoire de pile, comme *SafeStack* [17].

L'instrumentation est utilisée avec *LLVM* pour chaque instruction de chargement et de stockage. Si le pointeur utilisé dans l'accès mémoire n'a pas d'étiquette, MESH ne vérifie que si l'accès mémoire ne se fait pas dans le bloc de mémoire réservé pour les objets protégés.

Le surcoût de l'approche se mesure par la taille de la table de métadonnées, qui sera d'au plus 2 Mo pour *x86_64*, qui utilise 17 bits d'étiquette. Pour mesurer le surcoût de temps, la suite de tests *CoreMark* [18], conçue pour les systèmes embarqués, lui donne un surcoût d'un facteur de 2,7, alors que la suite de tests *ApacheBench*, associée au moteur web *Nginx*, lui donne un surcoût d'un facteur de 1,056 pour l'architecture *x86_64*.

Les accès mémoire répétés de *CoreMark* expliquent ce surcoût considérable. Dans notre contexte, bien que le surcoût en mémoire soit moindre, un tel surcoût de temps demeure inacceptable pour les systèmes embarqués. Un des avantages de cette approche, toutefois, est que MESH supporte le traitement multi fils.

ElectricFence

ElectricFence [1] est un outil simple de détection d'erreurs de mémoire. Il vise à résoudre les débordements de limites d'objets, et les erreurs liées à l'accès à la mémoire récemment désallouée. À cette fin, lors de chaque allocation, la page de mémoire suivante (ou précédente, au choix de l'utilisateur) de l'objet alloué est marquée inaccessible avec le drapeau `PROT_NONE`, avec l'appel système `mprotect` [19]. De plus, l'allocation même est arrondie à la page près, et les accès mémoires sont autorisés avec le drapeau `PROT_READ` et `PROT_WRITE` de `mprotect`. L'outil offre aussi une protection contre les allocations sur des zones mémoires

récemment désallouées par la fonction **free**, en interdisant l'accès à cette page de mémoire pour un certain temps. Puisqu'un accès à cette page de mémoire marquée inaccessible génère une erreur SIGSEGV, il n'y a aucun besoin d'instrumentation additionnelle lors des accès mémoire.

Puisque chaque allocation est arrondie à la page près, et qu'il y a donc une certaine partie de page qui est inutilisée par l'allocation, cette dernière est mise en place afin de se positionner à la fin de la page, lorsque le développeur veut détecter des débordements de mémoire sur une adresse supérieure aux bornes de l'objet. À l'inverse, cette page de garde sera mise avant l'objet, et l'objet placé au début de la page, lorsque le développeur veut détecter des débordements de mémoire sur une adresse inférieure. Cela signifie toutefois qu'il est impossible de détecter ces deux types d'erreurs en même temps avec cet outil.

Une page marquée inaccessible, pour chaque objet alloué dynamiquement, entraîne un surcoût de mémoire très considérable, surtout lorsque le programme fait un grand nombre de petites allocations mémoire. En effet, pour une allocation de 4 octets, 2 pages de mémoire se retrouvent réservées par l'outil, une page inaccessible et l'autre page qui contient l'objet alloué.

BaggyBounds

BaggyBounds [20] est un autre outil qui utilise une allocation limitée à des puissances de 2. Cette approche a comme avantage qu'il n'est pas nécessaire pour *Baggybounds* d'enregistrer les informations relatives à l'adresse de base et la taille, puisque les allocations sont toutes alignées et arrondies à une taille de puissance de deux supérieure. Ce qui sera enregistré sera plutôt le logarithme de la taille allouée en base 2. Dans les systèmes 64 bits, les bits 38 à 42 sont utilisés pour représenter ce logarithme, ce qui laisse 38 bits pour l'espace adressable. Ces 5 bits sont suffisants dans ce cas, puisqu'ils permettent des objets jusqu'à une taille de 2^{32} octets.

Le contrôle des limites des zones de mémoire est fait lors des opérations d'arithmétique de pointeur, au lieu d'exercer le contrôle à chaque accès mémoire. Pour une opération arithmétique sur un pointeur :

$$p' = p + i$$

Où p représente un pointeur, et p' représente le pointeur modifié, l'outil récupère le logarithme de la taille associée au pointeur p , calcule la taille de l'objet alloué, puis vérifie que le pointeur

p' se trouve toujours dans les bornes. Puisque chaque allocation est alignée, l'outil n'a qu'à s'assurer que le pointeur ne dépasse pas l'adresse alignée inférieure de s bits, où s est sa taille d'allocation.

La structure de métadonnées peut être omise pour inclure cette information dans le pointeur lui-même. En effet, puisqu'un seul octet est stocké par pointeur, il peut être mis dans les bits inutilisés du pointeur, sans pour autant changer sa représentation comme pour les pointeurs gras. Le matériel de mise en page doit être modifié, toutefois, pour éviter une exception lors d'accès mémoire avec les pointeurs étiquetés.

Utilisant les tests de performance de *Olden* [15] et de *SPECINT2000* [21], le surcoût de temps est d'un facteur d'environ 1,6 en moyenne, alors que le surcoût de mémoire est d'un facteur d'environ 1,15 .

2.4 Instrumentation

Toutes les techniques mentionnées plus haut permettent de détecter des erreurs de mémoire. Pourtant, elles sont inutiles sans une méthode pour vérifier chaque accès mémoire. Dans la plupart des cas, l'instrumentation est cette méthode clé qui permet la vérification de chaque accès mémoire. Comme mentionné précédemment, l'instrumentation consiste à insérer des instructions dans le programme. En interrompant chaque accès mémoire, l'instrumentation peut insérer des instructions afin de vérifier l'adresse accédée. Toutefois, il y a plusieurs méthodes pour implémenter l'instrumentation nécessaire à la vérification des accès mémoires, chacune avec des avantages et inconvénients. Comme précédemment mentionnée, l'instrumentation statique insère les instructions dans le code source directement, alors que l'instrumentation dynamique l'insère pendant l'exécution.

Il faut aussi ajouter la distinction entre l'instrumentation au temps de compilation et l'instrumentation au temps d'exécution. L'instrumentation lors de la compilation est plus performante, mais elle est certainement plus invasive que l'instrumentation lors de l'exécution même du programme.

Les outils d'instrumentation statique et dynamique peuvent être utilisés tout en assurant une assez bonne performance. L'utilisation principale de l'instrumentation dans le contexte de la détection d'erreurs de mémoire consiste à vérifier l'état de la mémoire avant chaque accès. Cependant, l'application de l'instrumentation peut être omise si l'outil de détection d'erreurs de mémoire peut prendre le contrôle du programme avant l'instruction de l'accès mémoire, par exemple par une interruption causée par la mémoire protégée (`PROT_NONE`) ou l'accès via un pointeur tainté.

Valgrind

Valgrind [22] est un cadre d'instrumentation binaire dynamique qui est utilisée par *Memcheck*, entre autres, pour fournir l'instrumentation nécessaire des accès mémoire. Il prévoit une architecture où un plugiciel peut être combiné avec le cadre *Valgrind* pour donner un outil de la suite *Valgrind* (comme *Memcheck*).

Valgrind offre du support spécialisé pour les valeurs fantômes. Pour ce faire, il répond à 9 exigences pour rendre les registres fantômes faciles à utiliser, ce qui constitue la première exigence :

1. Fournir des registres fantômes
2. Fournir la mémoire fantôme
3. Instrumenter les instructions de lecture/écriture
4. Instrumenter les appels système de lecture/écriture
5. Instrumenter les allocations de démarrage
6. Instrumenter les appels système qui allouent ou désallouent de la mémoire (par exemple `brk`, `mmap`, `munmap`)
7. Instrumenter les allocations/désallocations de mémoire de pile (stack)
8. Instrumenter les allocations/désallocations de mémoire de tas (heap)
9. Utiliser un canal latéral pour la sortie de l'outil

Puisque chaque opération `load/store` devient non atomique, du fait que l'on doit y rajouter une autre opération `load/store` pour sa mémoire fantôme équivalente, un mécanisme de verrou de fil d'exécution est implémenté pour pallier ce problème.

La sérialisation des fils d'exécution et la livraison de signaux asynchrones entre blocs de code fournissent une garantie que les outils utilisant des valeurs fantômes pourront exécuter des programmes multifs sans problème, ce qui répond à la seconde exigence.

Valgrind fournit un *système d'évènements* pour aider la communication des allocations/désallocations mémoire aux outils de vérification. Ainsi, l'outil peut recueillir l'information de chaque allocation/désallocation à l'aide de l'évènement envoyé par *Valgrind*. Pour s'assurer de pouvoir traquer les accès mémoire faits à l'aide d'appels système, un emballage est utilisé pour chaque appel système, bien que cela implique plus de 15,000 lignes de code. Ce système d'évènements donne donc la possibilité aux outils d'instrumenter les appels système, les allocations de démarrage, les allocations/désallocations d'appels système, et les allocations/désallocations de mémoire de pile (stack), ce qui répond aux exigences 4 à 7.

L'instrumentation des allocations/désallocations de la mémoire de tas n'est pas requise par le système d'évènement, les outils fournissant des emballages sur les méthodes d'allocations de la librairie C, telles que `malloc`, `realloc`, `calloc`, `free`, etc.

DynamoRIO

DynamoRIO [23] est un cadre pour l'implémentation d'analyses dynamiques et d'optimisations. Il est utilisé entre autres par *Dr. Memory* pour l'instrumentation du programme.

Lors de la représentation des instructions du programme, *DynamoRIO* utilise un système de détail par niveaux. Ainsi, l'utilisateur peut choisir le niveau de détail approprié et donc économiser sur le surcoût requis pour le décodage d'instructions. Les niveaux se définissent comme suit :

- Niveau 0 : Contiens les octets d'instructions d'une série d'instructions.
- Niveau 1 : Les informations du niveau précédent sont divisées par chaque instruction.
- Niveau 2 : Les instructions sont partiellement décodées pour obtenir le code d'opération (opcode) et son effet sur les drapeaux *eflags*.
- Niveau 3 : Les instructions sont complètement décodées pour y ajouter les informations sur les opérandes.
- Niveau 4 : Les instructions complètement décodées qui ont été modifiées, invalidant leurs octets d'instructions. Pour obtenir les octets d'instructions à nouveau, il faudra ré encoder les informations d'instructions.

Les niveaux de représentation d'instructions sont mis à jour automatiquement. Ces différents niveaux de détails permettent donc de minimiser le surcoût de performance utilisé par des outils qui veulent récupérer et/ou modifier les instructions-machine.

L'interface client de *DynamoRIO* permet diverses fonctions qui aident l'instrumentation des deux types de séquences de codes supportés, étant les traces et les blocs de code. Deux méthodes additionnelles sont ajoutées pour modifier les fragments de code lors de la compilation, afin de supporter l'optimisation adaptative.

DynamoRIO présente 4 optimisations additionnelles sur l'interface client, présentant qu'un faible surcoût en performance comparé à l'approche de base de *DynamoRIO* :

- Suppression d'opérations de chargement redondantes
- Remplacer l'instruction `inc` par `add 1` sur l'architecture Pentium 4
- Profilage de cibles de branches indirectes afin d'éviter des recherches de tables de hachage coûteux
- Traces personnalisées qui limitent les recherches de tables de hachage

LLVM

LLVM [24] se présente comme un cadre pour l’analyse de programmes à la compilation. Il met à profit la représentation du code afin de garantir 5 capacités :

1. Information persistante du programme, permettant l’optimisation du code en tout temps
2. Génération de code hors-ligne
3. Profilage et optimisation basée sur l’utilisateur
4. Modèle d’exécution transparent, permettant son utilisation par n’importe quel langage.
5. Compilation complète du programme de manière uniforme.

Ces 5 capacités ne sont pas garanties ensemble par les autres compilateurs du genre, ce qui avantage *LLVM*.

La représentation du programme offerte par *LLVM* se distingue des autres systèmes. Il utilise comme forme de représentation principale le *Static Single Assignment*. Ceci implique que chaque registre est écrit en exactement une instruction, et que chaque utilisation d’un registre est dominée par sa définition. Un contrôle de flux est généré pour chaque fonction dans la représentation de code.

L’architecture du compilateur *LLVM* est faite de manière à ce que les 5 capacités précédentes soient satisfaites. Premièrement, le compilateur statique traduit le code dans sa représentation *LLVM*, qui est fusionné avec le lieur *LLVM*. Le lieur *LLVM* effectue des optimisations de temps de liaison, comme des optimisations interprocédurales. Le code *LLVM* généré est ensuite converti en code natif, lors de la liaison ou de l’installation. Le générateur de code natif insère de l’instrumentation simple pour détecter les régions d’instructions répétées un grand nombre de fois, qui peuvent être optimisées lors de l’exécution.

Il représente un temps de compilation de programme bien plus petit que GCC, bien que GCC n’ait que très peu d’optimisations disponibles, comparé à *LLVM*.

2.5 Conclusion de la revue de littérature

Les outils de débogage de mémoire en temps réel utilisent tous des techniques similaires. La mémoire fantôme permet à certains des outils de stocker l’information d’adressabilité de la mémoire dans un sous-ensemble de mémoire. Les zones rouges sont disposées avant et après chaque allocation protégée afin d’assurer la détection des accès hors mémoire. Certains outils

profitent de la représentation des pointeurs afin d'y enregistrer une étiquette qui servira à identifier l'objet, ou disposent la mémoire d'une certaine façon afin de pouvoir extraire l'information requise par le pointeur de l'objet. Par contre, la grande totalité des outils (avec l'exception d'*Electric Fence*) doit tout de même instrumenter chaque accès mémoire, ou chaque opération arithmétique de pointeurs, même si l'objet concerné n'est pas protégé par l'outil.

CHAPITRE 3 MÉTHODOLOGIE

Notre processus de conception se résume à l’exploration de techniques visant à réduire les surcoûts associés aux outils de vérification des accès en mémoire. En effet, dans le contexte des systèmes embarqués, la tolérance des surcoûts associés aux outils de vérification de mémoire en temps réel est bien plus faible que pour d’autres systèmes. Il est aussi important de minimiser l’invasivité de l’approche. Par exemple, l’utilisation de l’instrumentation statique, lors de la compilation, comme pour *AddressSanitizer*, bien qu’elle réduise considérablement le surcoût en temps, est tout de même une approche invasive. En effet, cela requiert de recompiler le programme afin d’insérer l’instrumentation requise pour la vérification de chaque accès mémoire. Pour cette raison, notre meilleure option reste l’instrumentation lors du temps d’exécution du programme.

Notre approche prend inspiration dans le travail passé de Ciena sur le sujet avec *Data Watch* [25]. Il utilise une approche avec étiquettes de pointeurs semblable à MESH, mais il doit l’adapter à des systèmes 32 bits. L’espace adressable virtuel est divisé en plusieurs régions, qui sont elles-mêmes divisées en plusieurs blocs de même taille, afin de prendre en compte toutes les tailles d’allocations possibles. Ayant peu de mémoire allouée, dans leurs cas d’utilisation embarqués, une telle approche peut être appliquée sans manquer d’espace. Une table de métadonnées est créée afin de traduire l’adresse virtuelle en adresse physique et enregistrer la taille d’allocation. Chaque accès à un pointeur étiqueté produit un *Data Access Exception*, qui est traité pour vérifier l’accès mémoire en question.

Dans le but de pleinement explorer les différentes techniques possibles, afin de satisfaire nos objectifs mentionnés plus haut, notre approche ne traite initialement pas l’exécution multi fils et se base sur l’architecture `x86_64`.

Contrairement aux autres outils discutés dans le chapitre précédent, qui évitent de déclencher des signaux SIGSEGV, cette fonctionnalité sert à notre approche afin de prendre contrôle du programme. Une fonctionnalité comme *Top Byte Ignore*, avec les pointeurs taintés, pour le cas de *Hardware-Assisted AddressSanitizer*, permet d’éviter les coûteuses interruptions de signaux SIGSEGV. Cependant, ceci exige d’instrumenter tous les accès en mémoire.

En utilisant des pointeurs taintés pour générer des interruptions, ceci a pour effet que la vérification d’accès mémoire ne sera effectuée que pour les objets protégés. Bien que, par défaut, tous les objets alloués sur la mémoire de tas sont protégés, cela laisse la possibilité de protéger un certain sous-ensemble des objets alloués, afin de réduire considérablement le surcoût en temps et en mémoire. En effet, dans plusieurs cas, les développeurs peuvent

avoir certaines informations partielles sur l'objet mémoire posant problème. Par exemple, s'ils savent que l'objet qui présente un problème d'accès en mémoire a une taille probable de 8 octets, ils pourront alors protéger tous les objets de taille de 8 octets et assurer un surcoût adéquat pour le contexte des systèmes embarqués.

Le développement de cette approche réduit l'invasivité de l'outil proposé, ou la perturbation de l'implémentation sur le système. Ainsi, il n'y a aucun besoin de recompiler le programme, puisque l'insertion de la librairie se fait lors de l'exécution.

Pour efficacement inspecter les accès mémoire faits lors de l'exécution d'un programme, il faut suivre ces 6 étapes, lors de chaque accès mémoire d'intérêt :

1. Prendre contrôle du programme avant l'accès
2. Vérifier que l'accès est valide
3. Déprotéger l'objet impliqué dans l'accès
4. Effectuer l'accès
5. Protéger l'objet à nouveau pour les accès futurs
6. Continuer l'exécution du programme

L'implémentation des différentes techniques utilisées pour répondre à ces 5 étapes est détaillée dans le chapitre suivant. Ce chapitre constitue en même temps un article scientifique, intitulé "Targeted Memory Runtime Analysis Techniques for Embedded Systems" et soumis à la revue "Software : Practice and Experience".

Les travaux des autres étudiants du laboratoire DORSAL seront d'une aide précieuse. En effet, l'utilisation de la librairie *Libpatch* [26], développée par Olivier Dion dans le cadre de sa maîtrise, nous sera indispensable à la conception d'une solution basée sur l'émulation d'instruction. *Libpatch* est un outil qui se spécialise dans l'insertion de *probes* lors de l'exécution d'un programme. Elle remplace l'instruction ciblée par une instruction de branchement vers un trampoline. Le probe est ensuite exécuté, puis l'instruction ciblée est émulée avec le tampon OLX (*Out-of-line*), puis une instruction de branchement retourne à l'instruction suivante, en ayant rétabli le contexte d'exécution. La librairie *Libolx* [27] est une version minimaliste de *Libpatch* qui n'implémente que le tampon OLX. Cette librairie sera aussi utilisée dans la conception de notre solution.

CHAPITRE 4 ARTICLE 1 : TARGETED MEMORY RUNTIME ANALYSIS TECHNIQUES FOR EMBEDDED SYSTEMS

David Piché

Michel Dagenais

Mohammad Nassiri

As a master’s student, although my coauthors have helped me greatly in the work presented by this paper, I contributed to every aspect of this article : conception, experimentation, analysis, writing, and related work research. This article was submitted to Software : Practice and Experience in August 30th 2023.

4.1 Abstract

Memory errors still plague C/C++ systems, because memory allocation and deallocation are left to the responsibility of the developer. Notably, out-of-bound and use-after-free errors each represents problems that may result in crashes or security vulnerabilities, and thus need to be addressed. Existing memory verification tools like *ASan* and *Memcheck* are fairly effective at finding memory access errors, and are used extensively for debugging. However, they exhibit a large overhead in terms of memory and runtime. *ASan* makes an interesting subset of the verifications performed by *Memcheck*, at a much smaller but still sizeable overhead. Moreover, it requires recompiling the program. The overhead of these tools is not acceptable when the program is already at the limit of the available hardware, especially in constrained environments like real time and embedded systems.

To decrease the overhead, and avoid recompiling the program, various approaches were investigated in this work : pointer tainting, changing memory region access rights, software interrupts, dynamic instrumentation, out-of-line instruction execution, and selective verification of heap allocated objects. This paper explores these multiple techniques in an effort to reduce the verification overhead, while being minimally invasive, and ensuring efficient bounds checking. The allocation and memory access patterns of several programs from the SPEC benchmark were characterized, and new efficient strategies were proposed to debug memory access problems in embedded software. Our experimental results show that a hybrid configuration based on pointer tainting and out-of-line execution/patching achieve extremely promising results in terms of flexibility, memory consumption and run-time overhead. These

efficient strategies, together, form a flexible toolbox that allows the developer to select the objects to verify, and the most suitable verification approach to apply for each use case.

4.2 Introduction

C/C++ systems leave allocation and deallocation to the developer, allowing full control and maximum flexibility for memory management. By contrast, other programming languages such as Java use a garbage collector. Automatic memory management frees the developer from memory allocation and deallocation concerns, but may result in larger memory usage. More importantly, it may cause regular pauses for the garbage collection. However, the flexibility and control brought by the explicit memory allocation and deallocation opens a Pandora box of memory access errors, resulting in corruption and vulnerabilities that still plague systems based on C/C++ to this day. Those errors can easily pass unnoticed to the eye of the developer, if they are not triggered during the testing phase. As a means of detecting these errors, static or runtime verification may be used.

Static verification is a different area and is out of the scope of this work. It is extremely difficult to reason automatically in terms of memory accesses through pointers. Static analysis tools have the potential of detecting all errors, not only those triggered during a specific execution. However, in practice, tools based on static verification either i) only report simple, easy to prove, errors, or ii) generate a significant number of false positives and are almost unusable. Thus, many compilers will report errors of type i) but developers will resort to runtime verification tools for detecting the remaining memory access errors.

Many runtime memory access debugging tools use a mix of different techniques, the most common being redzones, shadow memory, and pointer tagging. In addition, most tools must use instrumentation to verify each memory access or pointer arithmetic operation. The instrumentation can be inserted either statically, at compilation time, or dynamically, during the execution. Static instrumentation is easier to implement and generally more optimized. However, dynamic instrumentation obviates the need for recompilation and is more convenient for the developer.

There have been several advances in reducing memory and runtime overhead, but memory accesses runtime verification tools remain difficult to use in the context of embedded systems. Furthermore, many tools use approaches that are far too invasive for use in embedded systems. *ASan* [2] (or *Address Sanitizer*) and *Memcheck* [3,4] (from the *Valgrind* framework) are both popular memory runtime verification tools, but they each suffer from overhead that is too large for most embedded systems. The work presented is based on the approach put forward

by *Data Watch* [28, 29], and extends its applicability to user-mode applications and complex instruction sets.

This paper proposes different techniques in order to reduce the verification overhead, while keeping the approach flexible and as unintrusive as possible. Section 2 describes relevant related work, then Section 3 details different possible approaches for instrumentation and memory accesses verification. Section 4 details the implementation of the proposed new approaches, and Section 5 presents the validation results, before the conclusion in Section 6.

4.3 Related Work

The different techniques described in the literature for memory accesses verification are presented here, along with how they are used in the different runtime verification tools.

4.3.1 Redzones

In order to detect out-of-bound errors, memory regions marked as non-addressable, called redzones, are placed after and/or before every allocated object. Any access to a location in a redzone is a memory access error. This can only detect out-of-bound errors up to the size of the redzone. If an out-of-bound access falls within a subsequent object, the error will not be detected. As such, there is a trade-off between out-of-bound detection and memory overhead, in selecting the size of the redzones. This technique is often complemented with a quarantine for newly freed objects, to detect use-after-free errors. Newly freed objects are kept aside, unallocated, for a certain time, acting like redzones. To detect accesses to non-addressable regions, different techniques may be used. If every memory access is instrumented, a table of *shadow memory* may be used to verify if the address accessed is not in a redzone. This is one of the techniques used by *ASan* [2]. Otherwise, memory protection hardware (mprotect with `PROT_NONE`), may be used to detect accesses to redzones. The difficulty, though, is that memory protection hardware has a granularity of one page (4KB) on most systems, possibly requiring a full protected *guard* page as redzone after each allocated object. Tools like *ElectricFence* [1] only use such redzones and memory protection hardware, in order to detect out-of-bound errors and use-after-free errors. The runtime overhead is small, since memory accesses are not instrumented and memory protection software interrupts are only generated in case of an error. However, its memory overhead is enormous, since it rounds the allocation to the nearest page size multiple, and adds another page for the redzone.

4.3.2 Shadow Memory

Many memory debugging tools use a technique known as *shadow memory* to store metadata about the state of the different memory locations or allocated objects. The entire allocation space of the program is represented by a subset called the shadow memory. It stores their information about the availability status of memory regions, namely if the memory is undefined, uninitialized or addressable. Thus, when instrumenting each memory access, the shadow memory entry for the memory access address is checked, in order to verify the validity of the access. On its own, it remains very prone to out-of-bound errors. Indeed, an out-of-bound access to an object, that falls within a subsequent object, will not be detected by the use of shadow memory alone. Thus, redzones, separating the allocated objects, are often used in conjunction. Several major memory debugging tools use shadow memory.

Memcheck [3, 4] is a popular tool of the *Valgrind* suite. It uses a combination of shadow memory, redzones and runtime instrumentation (with the *Valgrind* framework [22]) in order to verify each memory access at runtime. It can detect a large amount of memory error types [12], such as reads or writes to non-addressable memory, read accesses to uninitialised values, incorrect (uninitialised or non-addressable) pointer arguments to system calls, etc. Running a program under *Memcheck* incurs a runtime overhead of 22.2x and a memory overhead of 1.26.

ASan [2] is a popular memory error detector that, much like the *MemCheck* tool of *Valgrind*, uses shadow memory to store information about the state of the addressable memory. An eighth of the allocated memory is thus added as shadow memory to track the addressability of every byte. It then instruments, at compilation time, every memory access, to check for its addressability, using the shadow memory for bounds checking. It also creates redzones at compile time around global and stack objects, and at runtime around heap objects, to detect underflows and overflows. It achieves an average running time increase of 73% and uses 3.4x more memory. This approach carries a lower runtime and memory overhead than most other tools, making it very popular among developers. However, it still presents a significant overhead, unsuitable for most real-time and embedded systems. It is also very invasive, since the developer needs to recompile the program. Finally, it detects fewer errors than *Memcheck* since, unlike *Memcheck*, it does not differentiate between a non-addressable byte and an uninitialized byte.

4.3.3 Pointer Tainting

Pointer tainting uses the representation of the pointer itself to store useful information for bounds checking. On many architectures, including the `x86_64` architecture, the top 16 bits of the 64-bit representation of the pointer are unused. However, canonical addresses require the 16 most significant bits to be of the same value (either all 0s or all 1s), triggering a SIGSEGV/SIGBUS signal otherwise. However, architecture specific techniques can be used to store information in some of those 16 bits. For example, *Hardware-Assisted ASan* [6], a variant of *ASan*, takes advantage of the fact that the AArch64 architecture has a Top Byte Ignore feature that allows it to store information in the top 8 bits.

One technique is to store a color tag in the top 8 bits of the address, whenever a new object is allocated. This color tag, one among 256 (2^8) possible values, is also stored in the shadow memory corresponding to that allocated memory region. When there is a memory access, the color tag of the region shadow memory is compared to the pointer color tag, and a mismatch indicates an illegal memory access. Since the tag cannot be unique for each allocated object, the detection of out-of-bound accesses is probabilistic, there is a $1/256$ chance (.0039) of a collision, creating a false negative. Different techniques may be used for the color tag selection, to reduce the chance of false negatives. For instance, adjacent objects in memory should use different color tags to better detect slightly out-of-bound accesses. Similarly, when an object is freed and its memory is reallocated, a different color tag should be used to better detect use-after-free accesses.

An alternative technique is to use an object id (oid) to store in the unused bits of the object address. If only 8 bits are available, this strongly limits the number of objects for which runtime memory accesses verification is possible. When 16 bits are available, this is sufficient for most programs, especially on embedded systems. For each object identifier, a corresponding entry in a table in the shadow memory stores the bounds of the allocated object. When there is a memory access, the oid is used to check the access against the bounds stored in the shadow memory table entry, and detect any error. It is important to note that with either method, color tag or oid, it is checked that the proper object is accessed, not only that the accessed bytes are *addressable*. For this reason, redzones, and quarantines for freed objects, are not required. This considerably reduces the memory overhead of these techniques as compared to *Memcheck* or *Asan*.

Other approaches use similar techniques, storing object bounds or color, without tainting pointers with extra explicit information added to the pointer itself. Fat pointers use a small data structure, instead of a single word, to represent pointers. They store not only the address but also a color, oid or even the object bounds within that data structure. The

program needs to be recompiled to account for the increased pointer size. Moreover, this often creates problems for accessing outside libraries that do not expect such fat pointers. Some low-level programming techniques even make assumptions about pointer sizes and may not be compatible with fat pointers. *Cyclone* [7] uses fat pointers to provide a memory-safe pointer representation.

Interestingly, other approaches have succeeded in adding accurate bounds checking information, without tainting the pointers or increasing their size. Low-fat pointers [8, 9] and *BaggyBounds* [20] achieve this by storing size information implicitly in pointers. They restrict allocation sizes to powers of two, thus limiting the number of different allocation sizes. Then, the memory is separated in different regions, each storing objects of a specific size, aligned on their size. Thus, when memory is accessed through a pointer, it is easy to check in which region it resides, and consequently determine its size and start address. An object header, at the start address, can provide information about the object real size, before it was rounded to the next power of two. The advantage of this approach is that pointers are not changed and can easily be used with existing libraries, or as arguments to system calls. The problem with this approach is that objects are implicitly identified. Thus, an out-of-bounds access may implicitly lead to another object, not detecting the out-of-bounds error. To counter this, in [9] they instrument or analyze statically all pointer arithmetic, to detect any operation that would change a pointer beyond its targeted object. This requires static analysis and instrumentation, making this approach more invasive.

4.3.4 Instrumentation

All these techniques still need a way to check memory accesses. This is where instrumentation comes in, providing a way to insert instructions to validate accesses, before they are performed. Compilation time instrumentation is simpler and incurs less overhead than runtime instrumentation, but it is fairly invasive, as it requires to recompile the program. *LLVM* [24], *DynamoRIO* [23], and *Valgrind* [22] provide instrumentation frameworks that are used with current memory debugging tools. *LLVM* static instrumentation is used by *ASan*, while the *Valgrind* dynamic instrumentation framework is used by *Memcheck*. In addition, some memory debugging tools instrument pointer arithmetic operations. Low-Fat Pointers [8, 9] instruments each pointer arithmetic operation and memory access, and *BaggyBounds* [20] instruments each pointer arithmetic operation, but it does not instrument memory accesses.

With tainted pointers or hardware memory protection, explicit instrumentation may not be required. Indeed, accessing a tainted pointer on x86-64, because of the information stored in

the 16 most significant bits, will generate a software interrupt for the SIGSEGV/SIGBUS signal. Similarly, accessing a hardware protected memory region, for instance a redzone with mprotect property PROT_NONE, will also generate a software interrupt. An interrupt is more costly than a statically inserted check. However, it is less invasive and it allows to only check a subset of allocated objects, for which tainted pointers or memory protection are used.

4.3.5 DataWatch for Embedded Systems

Because the available memory debugging tools caused an unacceptable overhead for their embedded applications, Puncher and his group developed a low overhead tool for memory debugging [28, 29]. Their networking applications run on embedded systems with limited available memory, and came with a number of realtime constraints related to the timing of networking protocols. When a memory corruption problem is detected, a core image of the application is taken and the heap is analyzed through a debugger with analysis scripts. From the offending access (e.g., corrupted data or pointer), they can determine the corrupted object, and assume that the culprit is an out-of-bounds error while accessing the preceding object in memory. They can easily determine the size of the preceding object and thereafter focus their memory access verifications on objects of this size. They can then rerun the application while tainting, with an oid, the pointers for allocated objects of the suspected size. Each access generates an interrupt that allows to check the access. You then need to untaint the pointer, make the access by emulating the memory access out-of-line, retaint the pointer, and continue with the instruction after the memory access. This technique works well if the number of targeted objects, and the number of accesses, are not too large, keeping the overhead small.

While this tool demonstrated the interest of the approach for embedded systems, the specific context in which it was used simplified its implementation. Indeed, *DataWatch* was used in a single process context, where everything runs in the operating system kernel. This considerably reduces the cost of interrupts, as compared to a user application running under an operating system like Linux. Moreover, the tool ran on the PowerPC architecture, where emulating memory accesses out-of-line is much simpler than on a complex architecture like the x86-64.

The motivation for the work presented here was therefore to extend the approach, put forward by *DataWatch*, for its application to user-mode applications and complex instruction sets. Moreover, the memory allocation and access patterns of several applications were studied, and several major refinements and optimizations have been proposed to the techniques used in *DataWatch*.

4.4 Concepts

The basic approach is to verify accesses to dynamically allocated objects, with the possibility of verifying only a fraction of the objects, by selectively protecting only some of the objects at allocation time. The criteria to select which objects to verify may include the allocation location, the allocation size, or the rank among allocations (e.g., every 10th allocation). Often, a problem has been identified in a program and the specific object size involved in the problem is known. In that case, verifying only the suspected objects, at a much lower overhead, is advantageous.

To verify accesses in this way, several steps are required : i) get control before the access, ii) verify that the access is valid, iii) unprotect the object to perform the access, iv) perform the access, v) reprotect the object for verifying the subsequent accesses. There are many different ways to achieve each of these steps, from the least invasive and higher overhead, to the more complex and invasive but lower overhead.

To get control before the access, *ASan* recompiles the program, instrumenting all the memory accesses to execute a function that checks if the access is valid. *Valgrind* similarly instruments all the memory accesses, but using binary instrumentation. Both of these approaches are fairly invasive. Moreover, the access location is not a priori easily correlated with the object accessed. It then becomes difficult to only verify a selected subset of the objects.

Another technique, to get control before the access, is to get an interrupt upon attempting to access the object. A first method to achieve this is to protect the associated memory region. This is fairly non-invasive, as pointers are unmodified, but incurs a notable runtime overhead, unprotecting and reprotecting the memory region upon every access. However, if very few objects are protected in this way, and not accessed very often, this can be an excellent compromise. To control the protection of each object individually, they need to be put in separate memory regions. The minimal size for a memory region is 4 KB page on Linux x86. For objects much smaller than 4 KB, this represents a very large memory overhead. It is also possible to put several small objects in the same page, but in that case you lose control over protecting or not each object individually. Moreover, since the pointer does not contain an oid or color tag, redzones are required between the objects to detect out-of-bounds accesses.

Finally, pointer tainting is another method used to generate interrupts before the access. When an object is allocated, if it matches the criterias for getting its accesses verified, the pointer is tainted. An oid or color tag is added to its unused address bits, such that whenever the pointer is dereferenced, the tainted address is invalid and an interrupt is generated. At that point, the register containing the tainted address can be untainted, the access verified

and performed, and the register retained. Untainting and retainting only require a few instructions. The overhead is thus much smaller than with the previous technique that requires unprotecting and reprotecting a memory region. It is, however, more complex to implement and more invasive. Indeed, you need to disassemble the instruction in order to identify the register containing the tainted pointer, and reapply the taint after the access. Different special cases may arise, like if the register containing the tainted pointer is modified by the instruction doing the access. Indeed, if the register is overwritten, no retainting is necessary. Nonetheless, if the register is autoincremented, retainting should be applied over the modified value.

Verifying the access is actually relatively simple. Here again, 3 different strategies are possible : object identifiers, status bits, or color tags. Bounds (start address and size) can be associated with each object, either in the object header, or in a separate table with a unique object identifier. Another technique is to have status bits associated with memory locations. These bits tell if the corresponding memory location is allocated and initialized. This will detect if an unallocated location is accessed or an uninitialised location is read. However, an out-of-bound access to an allocated location in another object will not be detected. Memory tagging is an alternative that can detect such problems. Each allocated object gets a color tag (e.g. one among 256 possible) and the corresponding memory region is tagged with that color code. The returned pointer uses that tag as taint. Each access is then verified by checking that a tainted pointer accesses a memory location of the corresponding color. When an object is freed, its tag is set to a special color (e.g., 0). When an object is freed and its memory reallocated for another object, care is taken to ensure that it is tagged with a different color.

Surprisingly, performing the access is the more difficult step. Indeed, the difficulty lies in the fact that you need to get the control back, right after the access, to reprotect the object for subsequent accesses. One way to achieve this is to single-step the access instruction. Single-stepping an instruction is available through the *Ptrace* system call. It typically relies on special hardware facilities where a status bit is set before executing the instruction, causing an interrupt right after the next instruction, in order to regain control. This incurs a large overhead and must be performed from a separate process that controls the program through *Ptrace*.

A second method is to execute the access instruction out-of-line. This is commonly used in debuggers for multithreaded applications, like GDB in nonstop mode. For single-threaded programs, to resume execution after a breakpoint is hit, the breakpoint is removed and the program is single-stepped over the instruction. The breakpoint is then restored and the execution resumed. However, in a multithreaded program, when a thread hits the breakpoint,

you cannot remove the breakpoint because it must remain there for other threads that are still executing. The solution is to execute the instruction out-of-line and resume the execution after that instruction. Out-of-line execution is much faster than single-stepping with *Ptrace*, but more complex to implement. Indeed, you need to compensate for the different location of the out-of-line copy of the instruction, for operations that depend on the instruction pointer. The final step, reprotecting the object, is the simplest. You simply reprotect the memory region, or retaint the pointer in the appropriate register, before resuming the execution.

A special case needs to be handled differently for accesses : system calls. When a tainted pointer, or a pointer to a protected object, is passed as an argument to a system call, an invalid memory access will be detected upon checking the arguments in the operating system kernel. The system call will then fail, instead of generating an interrupt. Here again, several methods are possible to handle this, representing different compromises in terms of overhead versus invasiveness. A first method is to modify the Linux kernel, to remove the taint from pointers received as arguments. A second method is to wrap all calls to system libraries, to check accesses and remove taints from pointer arguments. This is difficult to achieve, because not all library calls are easy to intercept. The GNU C library is known to use complicated macros and other intricate methods to satisfy its needs of adapting to a large array of different architectures. This makes intercepting all system calls very difficult. A last method, easier to implement but with a much higher overhead, is to use *Ptrace* to intercept the execution before and after each system call. It then becomes possible to unprotect memory objects or untaint pointers. Nonetheless, you need to know which arguments are pointers, to untaint them. If using memory protection, you can always unprotect all memory regions containing protected objects, before each system call, which is simpler but still higher overhead.

4.5 Implementation

This section describes the different methods used in our prototype, MallocWatch, and their specific implementation details. For each of the alternative methods discussed in the previous section, we have implemented either a working prototype, or a micro-benchmark to measure its associated overhead. This allows us, in the evaluation section, to compare all the approaches in terms of overhead.

For all approaches, we use the `LD_PRELOAD` technique to load our shared library before the program. This ensures proper initialization, before the underlying program starts, and allows intercepting all the memory allocation functions.

4.5.1 Memory Protection

In order to ensure that allocations are protected, as for many of the aforementioned memory debugging tools, we have implemented a custom wrapper of the different allocation methods in the standard C library, such as `malloc`, `realloc`, `memalign` and `calloc`. This custom allocator gives us a way to protect objects at allocation. We have also implemented a wrapper of the `free` function to remove protection as needed.

In order to protect the allocated objects, as mentioned earlier, two main approaches were implemented. The first is by using the `mprotect` system call with the `PROT_NONE` flag. This ensures that any access to that region triggers a segmentation fault, which we can use to check its bounds. It does require us to round the size of the protected area to the nearest page. It also only protects against out-of-bound accesses that fall inside those previously protected allocation pages. This raises the possibility of false negatives.

The second way to protect the allocated objects is by using the 16 most significant bits of the 64-bit pointers, as a way to store a unique object identifier. This means that there can only be 65534 unique allocations at the same time, which may or not be an issue for large programs. Since the canonical address format requires all bits of this field to either be all 0s or all 1s on x86-64, the resulting segmentation fault can then be used as a way to interrupt the program and verify the memory access. Having a unique object identifier means that our runtime library needs to track the availability of each unique identifier. A color tag could be used instead of an oid for access checking, but the implementation and associated overhead would be almost identical.

In order to be able to do bounds checking, both methods use a metadata table with information about each allocated object size and base address. An additional field serves as a flag to indicate if the oid is currently used or not. The custom `free` wrapper takes care of removing the de-allocated object information from the metadata table.

As mentioned in the *Hardware-assisted ASan* design documentation, tainted pointers being passed to system calls do not trigger segmentation faults, and thus cannot be easily unprotected in the interrupt handler. In order to solve this, a kernel patch was made to remove the protection before the core of the system call. Requiring a modified operating system kernel may be considered quite invasive, especially for embedded systems. As an alternative, system call wrappers can be implemented using a preloaded library. The wrappers intercept the calls to the functions that issue the system calls in the standard C library. They check the validity of the pointer arguments before sending the untainted pointers to the real function. However, wrapping all the low-level library functions is a significant undertaking. For instance, some

functions like `ioctl` or `fprintf` receive a variable list of arguments that depends on a request type or a format string. In *Memcheck*, around 15000 lines of code are dedicated to verifying arguments to C library calls. In our initial implementation, a subset of common C library calls has been wrapped, which is sufficient for several applications.

A simpler but less efficient alternative to intercept all system calls is using *Ptrace*. The registers containing pointers, known based on the system call number, can then be unprotected or untainted. For objects protected with `mprotect PROT_NONE`, it is even possible to unprotect all the objects that the registers may be pointing to, without complex logic to determine which register actually contains a pointer (based on the system call number and parameters). Indeed, if a register contains an integer that could be mistaken for a pointer to a protected object, that object will be unprotected needlessly, but this will not interfere with the execution. However, in that case, if possible pointers are simply guessed, the corresponding memory accesses cannot be verified to flag access errors with certainty.

In the current implementation, the user can specify which objects are protected based on their size (minimum and maximum size) and their allocation rank (first protected object and maximum number of protected objects). Furthermore, the user can specify the protection strategy, `mprotect` or pointer tainting, and three different methods are available to handle tainted pointers, *Ptrace*, *Libolx* and *Libpatch* (explained in the *Single-Step* section below). The interest of the approach is that the memory overhead is smaller for tainted pointers. For both protection strategies, the runtime overhead is directly proportional to the number of accesses. If fewer objects are protected, the number of accesses to those objects will also be smaller. As mentioned earlier, the object size is often the selection criteria to protect a subset of objects.

4.5.2 Disassembly

In order to know which registers contain pointers, and thus which objects to unprotect and check for a valid access, the instruction causing the software interrupt must be disassembled and analyzed. This is achieved using the Capstone disassembly library [30]. This library provides the list of arguments (explicit and implicit) to the instruction and their characteristics (memory or register, size, read or write). On the x86-64 architecture, the general form of memory operands follows this formula :

$$Base + (Index \times Scale) + Displacement$$

The **Base** and **Index** registers can be omitted. If neither is present, it becomes a static absolute

address, the displacement. This is never used for pointers to a heap object. If the scale is not 1, then the index should be an offset and not a heap pointer, since multiplications do not make sense for absolute addresses. The base register content would then be the *base* heap pointer tainted to generate an interrupt. However, if the scale is 1, then the base and index registers have the exact same effect, and either could contain the heap pointer generating the interrupt. Their content is examined to determine which one contains a tainted pointer.

The disassembly takes place only upon the first encounter of an interrupt for a given instruction address. At that point, the scale, base and index values are used to determine which register should be untainted. The underlying assumption is that the register containing the protected pointer will not change for the same instruction. This seems to be the case with our benchmarks so far, and we have put an assertion in our tool to validate this assumption.

While most of the x86-64 instructions can only have a single memory operand, a few instructions, notably string comparison operations, may have two memory operands. Therefore, there may be two heap pointers to unprotect and verify their access. It is also important to verify the register operands being modified by the instructions. If the heap pointer in the base or index register is overwritten by the instruction, it should not be retained after the instruction execution. Here again, an assertion was added to our tool to verify that when the tainted register is not an output register operand, it remains unmodified after the instruction.

With the VEX extensions, the x86-64 architecture has some instructions that accept a vector register as an array of indexes, in order to access an array of values to transfer to a vector register. The case where the base register is not used, and there is a vector of tainted pointers, is not currently handled but would be a simple extension to the current implementation.

4.5.3 Single-step

To retaint a register, or reprotect a memory region, after executing the instruction that generated an interrupt, the control should be regained. This can be achieved by single-stepping the instruction. A first approach is to use *Ptrace*. The *Ptrace* system call allows one process to trace and debug another process. The traced process will execute the program with special allocators that protect some heap allocated objects. The second process, known as the *tracer*, will trace the first process, and handle verifying protected memory accesses with proper bounds checking.

This approach, using *Ptrace*, does not need to use a custom signal handler for handling the SIGSEGV/SIGBUS signals. Instead, it uses the communication between the traced process and the tracer to intercept signals and pass the necessary information such as register values.

However, the two processes should ideally share the same virtual space, as pointer values are only valid if used within the same virtual space. Using `clone()` with the `CLONE_VM` flag ensures that the tracer process shares its virtual space with the traced process (that created it from within the preloaded library, for runtime verification). The disadvantage of this relatively simple approach to single-stepping is the overhead of using the *Ptrace* system call.

The second approach to single-step instructions is to use Out-of-Line eXecution (OLX). This is particularly difficult to achieve for a complex instruction set such as the x86-64. Fortunately, a new library for out-of-line execution, *Libolx* [27], was developed by Olivier Dion. It generates a sequence of binary instructions, *olx* sequence, that have the exact same effect as the target instruction. While in many cases, the *olx* sequence is nothing more than a mere copy of the original instruction, in other cases it contains a more elaborate sequence of instructions that compensate for the fact that the instruction is executed at a different address. It is the case, notably, for all instructions that are relative to the instruction pointer. The programmer can then specify a sequence of binary instructions to execute before (prologue) and to execute after (epilogue) the *olx* sequence, as well as a continuation address to resume execution. The epilogue simply needs to retain the register containing the heap pointer. However, the instruction sequences in the prologue or epilogue must not modify the execution environment (registers or flags) except for its intended purpose (retain the heap pointer in its register). Registers may thus need to be saved and restored from the stack, depending on the instructions in the prologue. The target address, to resume execution, is the address of the instruction that follows the offending instruction.

The execution then proceeds as follows. When a tainted pointer is accessed, an interrupt is generated. The signal handler looks at the target instruction address. Upon the first encounter of this address, the instruction is disassembled, its tainted register identified, and an *olx* buffer created for the instruction (*olx* sequence and retainting epilogue). Thereafter, the register is untainted, the memory access checked, and the saved instruction pointer is replaced by the address of the *olx* buffer. Upon returning from the handler, the content of the *olx* buffer is executed, with the *olx* sequence to emulate the offending instruction and the epilogue to retain the register. Then, the *olx* buffer contains a jump to the continuation address, the next instruction after the offending instruction. The execution of the few machine instructions in the *olx* buffer only takes a few cycles. The larger part of the overhead is associated with the software interrupt and executing the SIGSEGV/SIGBUS signal handler, which can take thousands of cycles.

As an optimization, when an interrupt is encountered for an instruction, a dynamic instru-

mentation point may be placed on the instruction. This is achieved with a new library for dynamic instrumentation, *Libpatch* [26], developed by Olivier Dion. In the simple case, it replaces the target instruction with a jump to an olx buffer. The prologue of the olx buffer saves the registers, calls a user-defined pre handler, restores the registers, executes the olx sequence, saves the registers, calls a user-defined post handler, restores the registers, and jumps to the continuation address, the instruction that follows the target instruction. The situation is more involved if the target instruction is shorter than a jump instruction, but the result is the same with the pre and post handlers being called around the target instruction. For a few short instructions, the dynamic instrumentation may fail. In that case, the optimization will not be possible and interrupts and an olx buffer will be used.

The execution can then proceed as follows. When a tainted pointer is encountered for the first time by an instruction, the interrupt handler is called. *Libpatch* is then used to dynamically instrument that instruction. The handler then returns, and the target instruction is retried. However, this time it has been instrumented. The pre handler will check if the registers identified during the disassembly contain a tainted pointer. If so, the pointer will be untainted and the access checked. After the instruction is executed out-of-line, the post handler will be called to retaint as needed the registers containing the pointers. The execution then resumes at the continuation address. When the same instruction is executed later, because it is instrumented, no interrupt will be generated. The pre handler will check for tainted pointers before the instruction is executed out-of-line.

By calling user-defined handlers, *Libpatch* greatly simplifies the programming of pointer tainting and memory access checking techniques, using high-level languages rather than machine instructions. However, the execution of a pre and post handler takes more time than executing a handcrafted prologue and epilogue. Indeed, all the registers are saved and restored around these handlers, just in case. Another downside of this optimization is that the pre and post handlers are called every time the target instruction is executed, even if it is not processing a tainted pointer. Nonetheless, the big gain from this optimization is avoiding the interrupt, because of the dynamic instrumentation. This optimization is thus very beneficial when the target instruction is executed many times and always with tainted pointers.

The pseudo-code for the core of the prototype using *Libolx* and *Libpatch* is provided in Algorithm 1.

Algorithm 1 Core of *Libolx* and *Libpatch* prototype

```

1: ▷ Called when a tainted pointer is encountered, except when the access instruction was
   patched with a pre handler that untaints the pointer
2: function SIGSEGV_HANDLER(context)
3:   fault_insn = context->regs[REG_RIP]
4:   entry = get_table_entry(fault_insn)

5:   if entry not in table then                                     ▷ First encounter of that instruction

6:     continuation = fault_insn + fault_insn_size
7:     olx_buffer = create_olx_buffer(fault_insn, prologue, continuation)
8:     entry = add_table_entry(fault_insn, olx_buffer)

9:     ▷ The instruction is disassembled to determine the tainted register(s) and other
       characteristics, and an entry added for this instruction in the table

10:    patch(entry, fault_insn, pre_handler, post_handler)

11:    ▷ We try the optimization of dynamic instrumentation with Libpatch

12:    if patch succeeded then

13:      ▷ The patched instruction can be reexecuted with its handlers
14:      return

15:    end if
16:  end if

17:  entry->taint = UNTAINT(context->regs[entry->tainted_register])
18:  check_access(entry, context)
19:  context->regs[REG_RIP] = entry->olx_buffer

20: end function

21: ▷ Called before the out-of-line execution of a patched instruction
22: function PRE_HANDLER(entry)
23:   entry->taint = UNTAINT(context->regs[entry->tainted_register])
24:   check_access(entry, context)

25: end function

26: ▷ Called after the out-of-line execution of a patched instruction. The prologue in the olx
   buffer is similar to the post_handler
27: function POST_HANDLER(entry)
28:   RETAINT(context->regs[entry->tainted_register], entry->taint)

29: end function

```

4.6 Experimental study and Results

The primary target, for the proposed memory debugging runtime tools, is applications running in constrained environments like embedded systems. A common use case is to selectively protect a subset of the allocated objects, often based on their size. In order to determine the global memory and runtime overhead encountered in such a scenario, the number and proportion of allocated objects of each size are first measured. This determines the savings of selectively protecting objects, and the required size for the table of oid. Thereafter, the number of accesses to each object, and the number of accesses from each instruction involved, are measured. This information is used to determine how many olx buffers must be created with *Libolx*, or instructions should be instrumented with *Libpatch*, and how often they are used afterwards. Finally, micro-benchmarks are used to measure the cost of the different techniques. The relative cost of these operations helps determining in which situation each technique is more advantageous.

4.6.1 Experimental setup

In order to characterize the allocation and access patterns in typical applications, and measure the performance of the different runtime memory verification techniques, the SPEC CPU 2017 [31] benchmarks were used. These benchmarks were run on a AMD® Ryzen 7 5700g processor with 32 Gb of RAM, running Ubuntu 22.04.2 LTS with Linux kernel version 5.19.0-50-generic. The results will help to select the best configuration for detecting memory errors in different types of applications. As detailed in the previous section, 2 different methods are used to protect heap memory objects :

- Protecting memory pages with mprotect and the PROT_NONE flag
- Protecting pointers with an oid or color tag in the top 16 bits

Three different techniques were proposed to gain control before memory accesses, unprotect heap objects, single-step the execution and reprotect the heap objects after the memory access instructions :

- *Ptrace* with a tracer and a traced process, sharing their virtual space, to intercept signals and single-step target instructions
- signal handler for interrupts and *Libolx* for out-of-line execution of the target instructions
- dynamic instrumentation with *Libpatch*, for out-of-line execution and pre and post handlers

Finally, to address the issue of protected arguments in system calls, 3 different methods were

prototyped :

- Kernel module
- System call wrappers
- Handling each system call entry/exit with *Ptrace* traps to unprotect and reprotect heap objects

We have tested a subset of 4 possible combinations, from the safest and least invasive, but with higher overhead, to the more aggressive but lowest overhead. We have also measured the cost of each of the techniques involved, with micro-benchmarks. This allows us to better understand the performance of each of the combinations tested, and to extrapolate the performance of other possible combinations. The 4 combinations prototyped are as follows :

- Using *Ptrace* and *mprotect* (Ptrace-mprotect) : use *Ptrace* to intercept signals, single-step the target instruction, and intercept each system call
- Using *Ptrace*, tainted pointers and system call wrappers (Ptrace-taint) : use *Ptrace* to intercept signals and single-step the target instruction
- Using tainted pointers, a signal handler, *Libolx* and system call wrappers (OLX-taint) : use a signal handler to catch accesses to tainted pointers, and single-stepping the target instruction with *Libolx*
- Using tainted pointers, *Libpatch* and system call wrappers (Patch-taint) : dynamically instrument the target instructions and use the pre and post handlers to get control before and after the instructions, instead of single-stepping

4.6.2 Allocation Distribution

The number and distribution of allocation sizes are an important consideration to select an efficient runtime memory verification strategy. If all allocations are of the same size, selective protection of the objects based on that criteria becomes difficult. If more than 65534 objects are simultaneously allocated, the 16 unused bits on x86-64 will not be sufficient to assign a unique oid for each allocated object. Moreover, very small objects currently cause a very high memory overhead when protecting objects with *mprotect*, since each object occupies a whole page. For 15 of the SPEC CPU 2017 benchmarks, we implemented a small wrapper that tracks all allocations and deallocations within the program. From those logs, we generate 3 types of graphs :

- Distribution of allocated objects sizes
- Number of simultaneously allocated objects as a function of time
- Number of simultaneously allocated objects bytes as a function of time

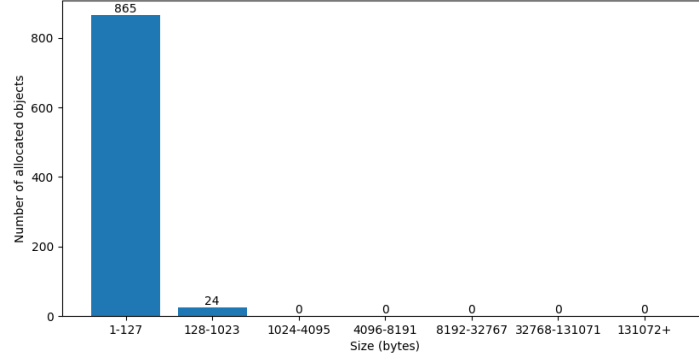


FIGURE 4.1 Distribution of allocated objects sizes (bytes) for xalancmbk

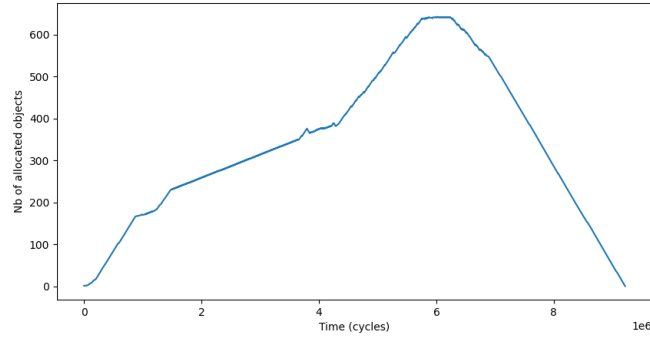


FIGURE 4.2 Number of allocated objects in time for xalancmbk

Figures 4.1, 4.2 and 4.3 show those 3 graphs for one sample benchmark, `523.xalancmbk_r`. Condensed information is found for the allocated objects size distribution in Table 4.1. Except for a few benchmarks, the vast majority of the applications have a very large percentage of allocations that are very small (smaller than 127 bytes). This observation makes our first approach (using `mprotect` to protect entire pages) very susceptible to large memory overheads.

4.6.3 Instruction Distribution

The first time an instruction accessing a tainted pointer is encountered, more work is required. The instruction must be disassembled to determine the tainted register. Furthermore, for OLX-taint the olx buffer must be created, and for Patch-taint the target instruction should be instrumented. More importantly, for Patch-taint, no software interrupt is needed on subsequent accesses. The impact of this optimization depends on how many instructions are repeated several times. We used the `Ptrace-mprotect` approach to measure the instruction distribution of accesses to protected objects. Even though it carries a larger overhead, it is

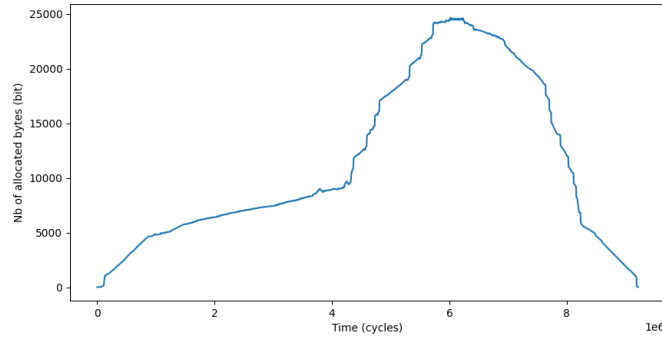


FIGURE 4.3 Number of allocated bytes in time for xalancmbk

less invasive and does not require complete coverage from the system call wrappers.

As seen in Table 4.2, every benchmark has multiple instructions accessing protected objects that are executed multiple times. In the case of `531.deepsjeng_r`, one specific instruction is run a very large number of times (1M+). These results show the interest of techniques like Patch-taint or even OLX-taint to reduce the runtime overhead. This conclusion is illustrated even more clearly with Table 4.3, showing the difference between the number of unique instructions and the total number of instruction executions accessing protected objects in the benchmark.

4.6.4 Overhead

Ptrace-mprotect

The first approach, Ptrace-mprotect, was tested against the 7 benchmarks showcased in Tables 4.2 and 4.3: `505.mcf_r`, `519.lbm_r`, `523.xalancmbk_r`, `526.blender_r`, `531.deepsjeng_r`, `541.leela_r` and `544.nab_r`. For a benchmark such as `519.lbm_r`, the time taken for handling a protected object access in the tracer process stays mostly constant at an average of 6.8 μ s. This includes getting a copy of the registers, getting the SIGINFO context with the address of the protected object, checking that the access is valid, unprotecting the object, single-stepping the target instruction, reprotecting the object, and sending back the control to the traced process. The total overhead includes, outside of this 6.8 μ s, the delay to generate an interrupt and wake up the tracer thread, because a protected object is accessed, and the time to wake up the traced process, once the tracer sends the `PTRACE_CONT` continuation command.

The overhead related to memory handling for other benchmarks stays similar at first, but

Benchmark name	Number of allocated objects of size						
	1 - 127	128 - 1023	1024 - 4k	4k-8k	8k-32k	32k-128k	128k+
500.perlbench_r	4224	271	58	3	4	0	0
502.gcc_r	776	5	210	4	7	19	0
505.mcf_r	60	2	1	2	0	0	0
508.namd_r	19	0	1	0	0	0	0
510.parest_r	210	110	5	2	1	0	0
511.povray_r	260	7	0	1	10	136	0
519.lbm_r	0	2	0	3	0	0	2
520.omnetpp_r	2649	261	5	0	0	0	0
523.xalancbmk_r	865	24	0	0	0	0	0
525.x264_r	1	3	0	0	0	0	0
526.blender_r	4657	3496	1162	15	2873	25	155
531.deepsjeng_r	0	1	0	1	0	0	1
538.imagick_r	1073	43	4	7	48	1	1
541.leela_r	2960618	64372	245642	369	475	2	268
544.nab_r	1600	4758	215	7	0	0	1

TABLE 4.1 Distribution of allocated object sizes for each benchmark

Benchmark name	Number of instructions run				
	1 time	2-10 times	11-127 times	128-1023 times	1024 times and more
505.mcf_r	129	30	11	43	317
519.lbm_r	109	13	4	37	234
523.xalancbmk_r	6209	3424	3183	1631	395
526.blender_r	743	836	451	249	290
531.deepsjeng_r	128	58	0	0	2
541.leela_r	184	133	498	166	261
544.nab_r	193	151	165	66	248

TABLE 4.2 Instruction Distribution

Benchmark	Unique instruction count	Total number of instructions run	Base execution time (seconds)
505.mcf_r	616	1515457704	5.473
519.lbm_r	397	1401898836	0.951
523.xalancbmk_r	14842	5555222	0.036
526.blender_r	2575	45584324	0.136
531.deepsjeng_r	232	1637329063	4.056
541.leela_r	1275	217922201	2.920
544.nab_r	882	469570759	0.700

TABLE 4.3 Unique instructions accessing protected objects, total number of instruction executions accessing protected memory objects, and base execution time (without any runtime verification).

steadily increases over time. This is due to the number of objects protected by this approach. Indeed, `519.lbm_r` issues very few allocations, with mostly one big allocation of 204 MB. The other benchmarks present a much higher number of allocations. This causes problems for the current method to search for the object protected by `mprotect`, based on the pointer address. With the tainted pointers, the oid is used as an index in a table, and the information about the object bounds is accessed efficiently. With `mprotect`, a linear search is currently performed in the table containing the protected objects bounds. This naive implementation was selected for `Ptrace-mprotect` because it was not seen as a serious contender, due to its large memory overhead, but rather as a safer and least invasive implementation for comparison purposes. It would be fairly simple to implement an efficient data structure for the search, if needed. We should therefore use the numbers obtained when few objects are protected, and disregard the effect of the inefficient linear search.

Ptrace-taint

Using a micro-benchmark, the overhead associated with the `Ptrace-taint` approach was evaluated. A simple program, repeatedly accessing a protected object, was implemented as a micro-benchmark to measure the cost of the approach. Figure 4.4 shows a code snippet of the micro-benchmark containing two memory accesses, on lines 5 and 7. These two instructions result in the following assembly code in Figure 4.5. With a `size` of 10 and a `loop_size` of 1 000 000, these two instructions are executed a total of 10,000 000 times each. The overhead per protected memory access is computed as the difference between the base execution time, without protection, and the execution time with protection, divided by the number of protected memory accesses. The overhead encountered for this micro-benchmark is obviously

maximized and not representative of typical applications. Indeed, it only repeatedly accesses protected objects, doing nothing else. A real application would do some other computations, and typically the user would only ask for a subset of the allocations to be protected.

There are a few differences with the *Ptrace-mprotect* approach. The memory protection with *mprotect* is avoided, a cost of about 0.8 μ s in total for unprotecting and reprotecting the object with the two corresponding calls to *mprotect*. In addition, the bounds checking only takes a few cycles, since the table is indexed directly with the oid used as taint for the pointer. On the other hand, a `PTRACE_SETREGS` command must be made to modify the register values in the traced thread, and another `PTRACE_GETREGS` command must be sent to verify that the register values stayed the same after the instruction is run using the `PTRACE_SINGLESTEP` command. The cost associated with those two *Ptrace* commands is about 0.97 μ s. The overhead measured per protected memory access is 14.7 μ s. By comparison, running the same micro-benchmark with the *Ptrace-mprotect* approach gives an overhead of 8.6 μ s per protected memory access. This measure is somewhat different from that reported in the previous subsection for the SPEC benchmark. The difference may be explained by the cost of the additional *PTrace* commands, and the fact that the earlier number excluded the time for waking up the tracer and the tracee during the communication.

```

1 1  crc = 0;
2 2  for (i = 0; i < loop_size; i++) {
3 3      for (j = 0; j < size; j++)
4 4          {
5 5              line = table[j];
6 6              crc += 1;
7 7              crc += line[0];
8 8          }
9 9  }
10

```

FIGURE 4.4 Code snippet of the micro-benchmark

```

0x55e982acb34c:  mov rax, qword ptr [rax]
0x55e982acb3b5:  movzx eax, byte ptr [rax]

```

FIGURE 4.5 Assembly instructions of repeated memory access instructions in micro-benchmark

OLX-taint

For OLX-taint, the same micro-benchmark was used in order to evaluate the overhead for each first protected object access instruction use, and for subsequent uses of that same instruction. For the Ptrace-mprotect and Ptrace-taint approaches, fewer optimizations were made and the same processing is performed each time. In the case of OLX-taint, the disassembly of the target instruction, and the creation of the olx buffer, happen upon the first encounter, causing additional work. Thereafter, the olx buffer already exists and is only executed.

The first access to a protected object takes an average of 4.55 μ s. This time includes taking the interrupt with the signal handler, disassembling the target instruction with *Capstone*, and creating the olx buffer. Thereafter, each subsequent access to a protected object incurs an average overhead of 1.13 μ s. This includes mostly taking the interrupt with the handler. Executing the olx buffer when exiting the interrupt handler only takes a few cycles. The time to save the protected registers, untaint them, execute the instruction out-of-line and reprotect the registers was measured to be 7.9 ns.

Patch-taint

For Patch-taint, the same micro-benchmark was used again in order to evaluate the overhead for each first protected object access instruction use, and for subsequent uses of that same instruction. The average time required for the first encounter of each instruction is 665 μ s. This large overhead is explained by the fact that great care must be taken when dynamically patching a program during its execution. *Libpatch* was designed for the live patching of multithreaded applications. It reads the debugging information, disassembles the whole function containing the target instruction to detect possible jumps to the patched bytes, patches the instruction in several steps, and synchronizes all the threads to issue a serializing instruction. The serializing instructions are required to ensure that the code modifications are coherently propagated to the instruction caches and internal structures of the x64-64 processors. The subsequent executions of the same target instruction require an average of 0.065 μ s (65 ns). A large portion of this time is devoted to saving and restoring registers around the execution of the pre and post handlers.

The large overhead, associated with the dynamic instrumentation upon the first encounter, is a concern. *Libpatch* is relatively new and was developed in the context of dynamically instrumenting programs for tracing. Further optimizations are possible, for instance if the target program is single-threaded. In the context of the runtime verification of memory accesses, important optimizations would be possible. For example, a sample run could be used

to detect the location of most of the instructions accessing protected objects. Then, the program could be run with an initial list of instructions to patch together, from the start. When the patching of several instructions are batched together, the threads synchronizations and serialization instructions steps can be shared, resulting in significant savings.

Another interesting opportunity is to eliminate the need for disassembling whole functions at runtime, when patching individual instructions. This is currently required to determine if other instructions may jump in the middle of the patched bytes. Indeed, the dwarf format [32] has provisions to store information about basic blocks. The basic block heads are the only jump targets in a function. Unfortunately, most compilers do not yet provide this information, but could easily. It would also be possible to compute this information, about jump targets, offline to avoid having to do it at runtime.

While the overhead for subsequent instructions with Patch-taint, 65 ns, is much lower than for OLX-taint, 1.13 μ s, further optimizations are possible. Currently, the code surrounding the pre and post handlers is very conservative and saves all the registers. This accounts for a large part of the overhead. The author of *Libpatch* plans to offer a choice of configurations, allowing to choose which registers are saved, depending on the content of the pre and post handlers. Ultimately, the cost could approach that of the untainting and execution of the olx buffer, in the OLX-taint method, at 7.9 ns.

Comparing the Approaches

The overhead was measured using the micro-benchmark and the time command. The micro-benchmark was executed once without memory protection, and once with each of the four approaches discussed. Since the two memory access instructions in the micro-benchmark are executed a very large number of times, we can accurately evaluate the overhead per instruction accessing a protected object. Table 4.4 shows, for each approach, the associated execution time, and overhead per instruction.

Approach Used	Time (s)	Overhead per Instruction
No protection	0.017	-
Ptrace-mprotect	173.120	8.6 μ s
Ptrace-taint	293.619	14.7 μ s
OLX-taint	22.649	1.13 μ s
Patch-taint	1.320	65 ns

TABLE 4.4 Micro-benchmark results

From the SPEC benchmarks execution times, and the number of protected memory accesses, provided in Table 4.3, and the overhead per protected memory access of each approach, provided in Table 4.4, we can extrapolate the overhead that would be obtained for each benchmark and each protection approach. This is presented in Table 4.5, with a comparison to *AddressSanitizer*'s time overhead for those same benchmarks. It is interesting to note that the OLX-taint approach is significantly more efficient than the Ptrace based approaches. Even more spectacular is the further reduction in overhead brought by the dynamic instrumentation, with the new Patch-taint approach.

Unsurprisingly, the runtime overhead remains significantly larger than with *Asan*. Indeed, *Asan* uses static instrumentation, which allows for the highest level of efficiency. Moreover, it was refined over several years. Nonetheless, *Asan* is not applicable in most embedded systems, not only because of its runtime overhead of 1.73x, but also because of its 3.4x memory overhead and the requirement to recompile the application. As can be seen with the time of 7.9 ns measured in OLX-taint to untaint, execute out-of-line and retaint, we can expect further optimizations in *Libpatch* and in our code to reduce the overhead by another factor of approximately 5. The overhead then becomes closer to what static instrumentation has to offer. However, the huge advantage of the new approach proposed, with OLX-taint and Patch-taint, is that the developer can selectively protect objects, and in this way arbitrarily diminish the overhead.

4.6.5 Selective Protection Based on the Object Size

As previously mentioned, the Patch-taint approach could suffer additional overhead, as compared to OLX-taint, when a subset of all object allocations are protected. Indeed, since it patches the instructions accessing protected memory objects, a jump is installed over the

Benchmark	Ptrace-mprotect	Ptrace-taint	OLX-taint	Patch-taint	Address Sanitizer
505.mcf_r	2382x	4071x	313x	19.0x	1.30x
519.lbm_r	12678x	21670x	1666x	96.8x	1.30x
523.xalancbmk_r	1328x	2269x	175x	11.0x	3.19x
526.blender_r	2883x	4928x	379x	22.8x	1.90x
531.deepsjeng_r	3472x	5935x	457x	27.2x	1.82x
541.leela_r	642x	1098x	85x	5.85x	2.18x
544.nab_r	5770x	9861x	759x	44.6x	1.38x

TABLE 4.5 For each approach, the time relative to the base execution time is provided

target instruction. It jumps to an olx buffer that executes the pre handler, the olx sequence and the post handler. These handlers will thus get called every time the instruction is encountered, even when the pointer accessed is not tainted, adding overhead. The Patch-taint approach could thus be less advantageous than OLX-taint, if a large proportion of the executions are to access unprotected objects. We have analyzed two benchmarks, `519.lbm_r` and `523.xalancmbk_r`, to look in greater details at the frequently executed instructions and their access patterns.

For each frequently executed instruction, to access protected objects, we traced the size of the object accessed. In `519.lbm_r`, every frequently executed access instruction handles a single object size. This comes as no surprise for this benchmark, since a large number of instructions operate on a single large allocation, meant to represent a tri-dimensional space.

In `523.xalancmbk_r`, however, we have more interesting results. For the different target instructions, accessing protected objects, the instructions that are executed a very large number of times all (or almost all at 99,9 %) cater to a single object size. Nonetheless, some instructions that are executed a moderate number of times have a more diverse set of object sizes associated with their memory accesses.

Consequently, it appears that if a subset of the allocated objects are selected, based on the allocation size, the Patch-taint approach will remain extremely interesting. Indeed, instructions that are executed a very large number of times have nearly all the same related object size. If a different criteria is used to selectively protect the allocated objects, for instance the rank (in order to protect a single object) then the OLX-taint approach may become a more interesting choice.

4.7 Discussion

The problem of untainting pointers before they are used in system calls has been mentioned, but its performance was not discussed. The Linux kernel patch, to untaint pointers received as arguments, was proposed for *Hardware ASan* but never merged in the mainline. It works well and carries a negligible overhead. Nonetheless, the requirement of a modified kernel is a strong deterrent for many. The ptrace approach, to intercept system calls and untaint pointers, is a possible alternative with moderate but significant implementation complexity and a large runtime overhead. Wrappers are the least invasive approach, but come at a very significant implementation cost. We have implemented a partial set of wrappers for the C library calls, including several system calls. It works well and the runtime overhead is mostly negligible. However, completing the wrappers to cover a very large proportion of the C library

would be a major undertaking, as illustrated by the wrappers in *Memcheck* or *ASan* that contain more than 15000 lines of code. Fortunately, it is possible to extend our set of wrappers gradually, as the need arises.

Most aspects of the proposed approaches may be applied to several different architectures. Our work initially focused on the x86-64, because of its widespread use. Yet, because of its extreme complexity, there are still some corner cases that we do not handle, such as memory accesses with vectored indexes and no base register. We have explored the ARM and PowerPC instruction sets and they should be significantly simpler to support, especially for the dynamic instrumentation part for the Patch-taint approach. The *Capstone* library for disassembly already supports most major architectures. Both *Libolx* and *Libpatch* plan to support other architectures in the future, starting with ARM. It will then become relatively easy to adapt our tools to the different architectures. The architecture-dependent part, interfacing with memory addressing modes and registers, remains fairly small.

The current prototypes initially targeted single-threaded applications. Our focus was to compare the different mechanisms available and measure their associated overhead. Nothing in these approaches precludes analyzing multithreaded applications. Several adaptations will be required, however. First, all the shared data structures will require the use of a synchronisation mechanism, such as mutexes or read-copy-update. This concerns the table of protected objects, modified upon allocations and deallocations, and accessed every time a memory access is checked. It also concerns the table of targeted instructions, modified each time a new memory access instruction is encountered for the first time, and accessed each time one such instruction is executed. A second adaptation is to change a few global variables that store the execution status. They should be converted into thread-level storage variables. For example, a status variable denotes when we are within C library calls wrappers, and further calls should not be wrapped.

A more problematic concern, for multithreading adaptation, is that the *Libpatch* library can dynamically instrument multithreaded applications, but it is assumed that a single thread makes the calls to *Libpatch*. It is the normal case for tracing tools, the main area targeted by *Libpatch*. This suggests an interesting workaround that can be used as well to further optimize the Patch-taint approach. When a new memory access instruction is encountered, the OLX-taint method could initially be used. After a given instruction is encountered more than a certain number of times, e.g. 10 times, it could be queued to become handled with the Patch-taint method. A multithreaded queue would accumulate the list of instructions to patch from all the different threads. Then, a single thread would batch the instructions in the queue, to instrument them with *Libpatch*. This scheme would take care of ensuring

that only frequently executed instructions use the Patch-taint method, would batch together several calls to *Libpatch* to save on the dynamic instrumentation cost, and would ensure that a single thread interacts with *Libpatch*.

The link to the prototype tool, MallocWatch, is available at the following link : <https://github.com/damadux/MallocWatch>

4.8 Conclusion

This paper presents different techniques to reduce the memory and time overhead for the runtime checking of memory accesses, in order to make it applicable to embedded and real-time systems. The first technique, Ptrace-mprotect, results in the highest overhead. It boasts a large time overhead and a very large memory overhead. Furthermore, memory accesses are only checked to see if the target address falls within an allocated object (not in a redzone). This can create a large proportion of false negatives. Ptrace-mprotect is, however, the least invasive, since it does not modify the value of allocated objects pointers, avoiding the need to untaint and retaint pointers. The second technique, Ptrace-taint, presents a similar time overhead and almost no memory overhead. Moreover, each access is verified against the bounds of the targeted object, resulting in a more precise check. Nonetheless, it is more invasive than Ptrace-mprotect, and its time overhead remains large. The OLX-taint approach has similar characteristics to Ptrace-taint in terms of low memory overhead and checking precision. Yet, it boasts a much lower runtime overhead. It is the most interesting approach, when a very small subset of the allocated objects are protected.

The fourth and final approach, Patch-taint, presents the lowest runtime overhead, while sharing the low memory overhead and high-checking precision of Ptrace-taint and OLX-taint. It represents a major contribution of this work. It currently comes with a few limitations that will be addressed in future work. The first limitation is the large delay associated with the dynamic instrumentation of the target instructions. An offline analysis of the targeted application could be used to determine the possible jump locations, reducing the work required at instrumentation time. In addition, a coverage analysis of the application could be used to identify the frequently and unfrequently executed access instructions. Thereafter, at runtime, only the frequently executed access instructions would use the Patch-taint approach, while the OLX-taint approach would be used for infrequently executed instructions. As mentioned earlier, batching the dynamic instrumentation of several target instructions, when the application is started, is another way to reduce the instrumentation time per instruction.

The second limitation is the fact that the pre and post handlers are called every time a target

instruction is executed, even if the pointer accessed is not tainted. It adds an unneeded runtime overhead. This limitation will become much less problematic if the overhead of calling the pre and post handlers is greatly diminished. This will happen when more choices are added in terms of the registers saved and restored.

Together, these techniques constitute an invaluable toolbox that will provide embedded systems developers with great control over how they perform runtime memory access checking in highly constrained environments.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Limitations et améliorations futures de Libpatch

Tel qu'expliqué plus tôt, pour insérer les « probes » avant et après une instruction, cette instruction est remplacée par une instruction de saut, *jump*, qui mène vers un trampoline, où le(s) probe(s) sont insérés et l'instruction est émulée « out-of-line ». Cependant, lorsque l'instruction à remplacer est d'une taille de moins de 5 octets, *Libpatch* est contraint de remplacer une ou plusieurs instructions additionnelles afin d'atteindre les 5 octets de taille. Puisque ce groupe d'instruction est présentement émulé avant le « post-probe », ce dernier ayant comme but de protéger à nouveau le pointeur, notre contrainte de reprotéger le pointeur directement après l'accès mémoire n'est pas respectée. Si une autre instruction dans ce groupe d'instructions émulée manipule le même registre utilisé pour l'accès mémoire, le comportement original pourrait ne pas être préservé, corrompant l'exécution. Une solution à ce problème est toutefois en développement actif par son développeur, Olivier Dion.

La performance pour l'exécution des « probes » devrait aussi s'améliorer dans une version ultérieure de *Libpatch*. En effet, présentement tous les registres sont sauvés, ce qui donne toute liberté aux « probes » d'utiliser n'importe quel registre. En pratique, la plupart des actions pour la vérification de mémoire n'ont pas de tous ces registres. Un temps significatif pourra être gagné en optimisant le code qui appelle les « probes ».

5.2 Groupes de distribution d'allocations

Dans la section résultats de l'article, une analyse des allocations est effectuée dans le cadre de la suite de test *SPEC CPU 2017*. Elle se compose de 3 graphiques pour 15 tests *SPECrate* en C/C++ : la distribution des allocations selon la taille, le nombre d'objets alloué par rapport au temps. L'article montre ces 3 graphiques pour un seul test, *523.xalancbmk_r*. Cette section examine ces graphiques pour 6 autres tests de performances : *505.mcf_r*, *519.lbm_r*, *526.blender_r*, *531.deepsjeng_r*, *541.leela_r* et *544.nab_r*, qui sont affichés en annexe. Cette analyse de tailles est faite pour la suite de tests « Test ».

5.2.1 505.mcf_r

505.mcf_r est un programme de calcul d'optimisation par combinaisons [33]. Il est observé avec la figure 5.1 un nombre relativement faible d'allocations. De plus, aucune allocation

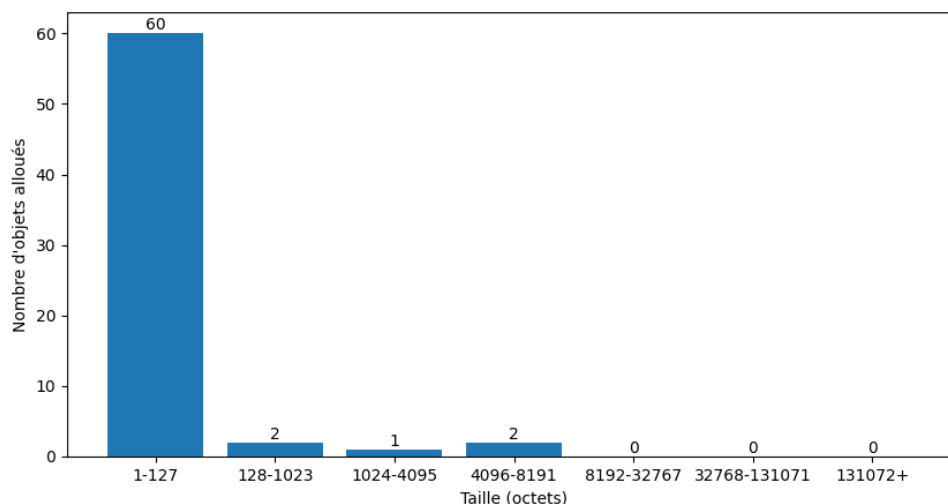


FIGURE 5.1 Distribution des tailles d'objets alloués pour mcf

n'est de taille supérieure à 2 pages de mémoire. L'existence de cycles d'allocations et de désallocations d'objets est confirmée avec la figure 5.2, bien que la figure 5.3 confirme qu'il s'agit un cycle d'allocations et de désallocations d'objets de petite taille.

5.2.2 519.lbm_r

519.lbm_r est un test qui fait un grand nombre d'instructions sur un tableau représentant un espace tridimensionnel pour une simulation de liquide [34]. De ce fait, il n'a que très peu d'allocations, mais une de ces allocations est d'un espace très grand, pour représenter ce tableau. Ainsi, on remarque sur la figure 5.4 un très petit nombre d'allocations. Toutefois, 2 de ces allocations sont d'une taille conséquente, représentant l'espace utilisé pour la simulation de liquide. Les figures 5.5 et 5.6 illustrent les graphiques d'allocations de ce test selon le temps, bien qu'ils demeurent peu informatifs dû au faible nombre d'allocations.

5.2.3 526.blender_r

526.blender_r est une suite de test concernant blender, un programme de rendu 3D [35]. Pour les figures 5.8 et 5.9, plusieurs cycles d'allocations et de désallocations d'objets y sont observés. Dû au nombre élevé d'allocations d'objets, la figure 5.9 illustre mieux cette présence de cycles. La figure 5.7 illustre un nombre élevé d'allocations de plus de deux pages de mémoire. Il y a tout de même une majorité des objets alloués qui sont de taille inférieure à une page.

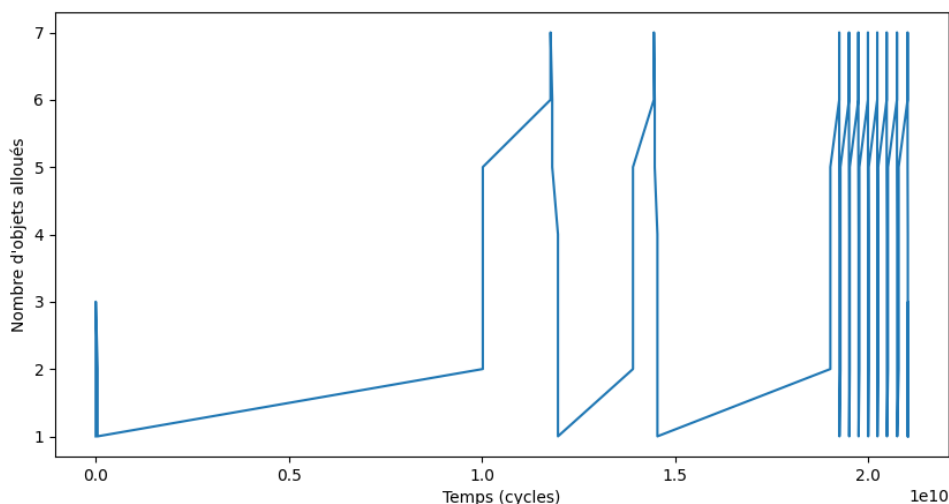


FIGURE 5.2 Nombre d’objets alloués dans le temps pour mcf

5.2.4 531.deepsjeng_r

531.deepsjeng_r est un programme d’intelligence artificielle par la recherche dans un arbre alpha bêta et la reconnaissance de motifs [36]. Similairement à 519.lbm_r, il est observé avec la figure 5.10 un faible nombre d’allocations d’objets. Toutefois, on remarque une allocation en particulier prenant une grande région de mémoire. Dans ce cas, les figures 5.11 et 5.12 ne sont que de peu d’intérêt, étant donné le faible nombre d’allocations au sein du programme de test.

5.2.5 541.leela_r

541.leela_r est aussi un programme d’intelligence artificielle, cette fois-ci par une simulation avec la méthode de Monte-Carlo, une recherche dans un arbre de jeu, et la reconnaissance de motifs [37]. Similairement à 526.blender_r et 505.mcf_r, on observe dans les figures 5.14 et 5.15 un cycle d’allocations et de désallocations d’objets. La figure 5.13 démontre aussi une grande proportion des allocations faites de taille inférieure à une page, bien qu’il y ait tout de même la présence d’allocations de très grandes tailles.

5.2.6 544.nab_r

Finalement, 544.nab_r est un programme de test de modélisation de molécules [38]. Comme indiqué à la figure 5.16, la très grande majorité des allocations prennent moins d’une page.

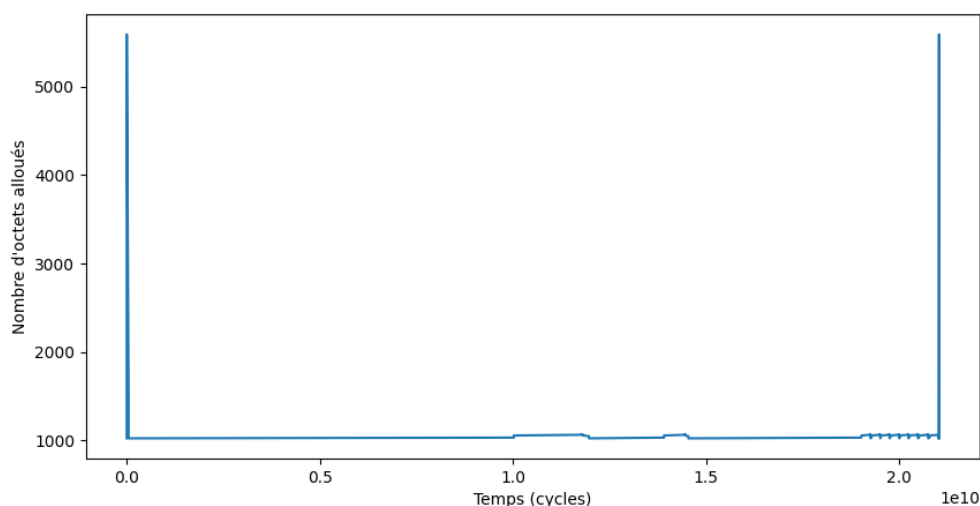


FIGURE 5.3 Nombre d’octets alloués dans le temps pour mcf

On aperçoit, quoique difficilement, dans les figures 5.17 et 5.18, un certain cycle d’allocations et de désallocations d’objets.

5.3 Considérations pour le respect des objectifs

Cette section résume les différentes considérations prises pour respecter les objectifs établis dans l’introduction, et ses impacts dans l’implémentation de la solution.

5.3.1 Minimiser le surcoût en temps

Le surcoût en temps de l’approche est un facteur important tout au long de ce travail de maîtrise. En effet, les techniques développées augmentent progressivement la performance. Les deux techniques conçues en premier, utilisant *PTrace* avec des méthodes différentes de protection, souffrent d’un surcoût de temps important. La conception des deux prochaines techniques, avec *Libolx* et *Libpatch*, sert à augmenter la performance en enlevant les appels redondants au désassembleur et en diminuant les signaux SIGSEGV levés.

De plus, notre approche apporte un avantage qui fait défaut aux autres outils de débogage : la particularité de pouvoir choisir un sous-ensemble des objets à protéger, selon un paramètre, par exemple la taille de l’objet alloué. En effet, bien qu’il soit évidemment possible de ne protéger qu’un sous-ensemble des allocations avec les autres outils de débogage mémoire, il est tout de même obligatoire d’instrumenter tous les accès mémoire (ou toutes les opéra-

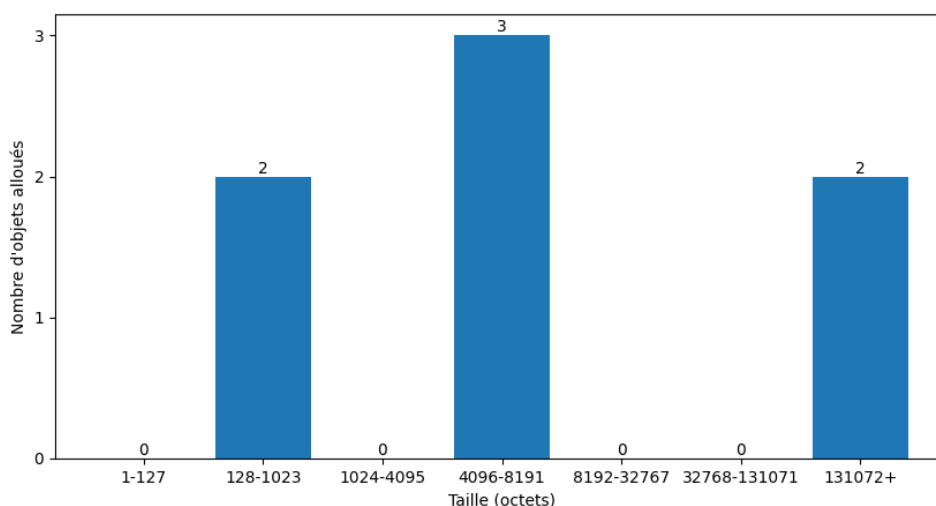


FIGURE 5.4 Distribution des tailles d'objets alloués pour lbm

tions arithmétiques de pointeur), peu importe si le pointeur en question est protégé ou non. Puisque notre technique tire avantage du fait que chaque accès mémoire impliquant une zone protégée lève un signal SIGSEGV, les accès en mémoire impliquant des zones non protégées sont ignorés naturellement. Dans le cas où un développeur sait qu'un objet d'une certaine taille cause une erreur en mémoire dans son programme, utiliser une telle approche est très avantageux.

5.3.2 Minimiser le surcoût en mémoire

Une attention particulière a été portée au surcoût en mémoire. En effet, en industrie, plusieurs systèmes embarqués sont très près de leur capacité maximale en mémoire. En effet, ceci arrive par exemple lorsque le même matériel reçoit de nombreuses mises à jour au fil du temps, qui à chaque fois ajoutent des fonctionnalités et augmentent la consommation de mémoire. Les deux techniques pour protéger la mémoire viennent chacune avec un surcoût en mémoire différent.

La technique de protection par pages, en utilisant `mprotect` avec le drapeau `PROT_NONE`, possède actuellement un coût en mémoire conséquent, puisque chaque page ne contient pas plus d'un objet à la fois, peu importe si l'objet en question est de petite taille. Pour combattre ce problème, il est discuté dans l'article qu'il est possible de permettre à plusieurs objets de cohabiter dans une même page, si la taille le permet.

Dans ce cas, une considération additionnelle doit être prise en compte. Contrairement à

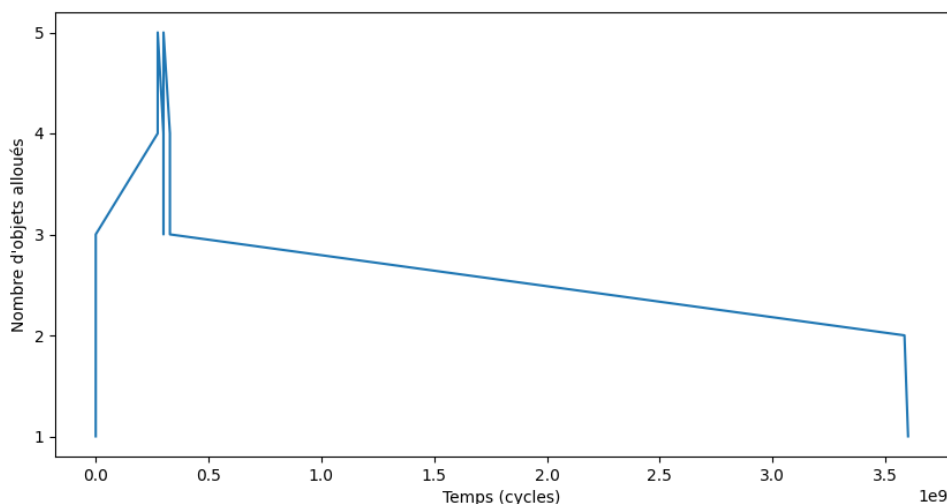


FIGURE 5.5 Nombre d'objets alloués dans le temps pour lbm

l'utilisation d'étiquettes de pointeurs, où l'étiquette permet à notre approche d'extraire les informations d'allocations sur l'objet, et ainsi d'assurer une vérification des bornes précise, cette information n'est pas disponible lorsque les allocations sont protégées par pages avec `mprotect`. Ainsi, dans le cas où plusieurs objets partagent la même page, des zones rouges doivent être utilisées entre les objets afin de détecter avec une certaine probabilité les accès hors borne. Comme mentionné dans la revue de littérature, la détection d'erreurs mémoire hors borne par zones rouges dépend de la taille de la zone rouge, une taille plus grande permettant une meilleure détection, au prix d'un plus grand surcoût en mémoire. Prenant inspiration d'*AddressSanitizer* [2], une taille de 32 ou 64 octets peut être un bon point de départ.

La technique de protection par étiquette de pointeur a un surcoût en mémoire relié à son tableau de métadonnées. Puisque chaque entrée dans la table prend 24 octets (8 octets pour chacun des 3 champs), on se retrouve avec un surcoût en mémoire d'au plus 1.5 Mo, assumant que les 16 premiers bits sont utilisés pour l'étiquette. Avec 16 bits d'étiquette, nous avons une possibilité de 65534 objets protégés (excluant les étiquettes tout 0 ou tout 1). Il est aussi possible de réduire la taille de chaque entrée de la table en réduisant la taille du drapeau.

5.3.3 Minimiser l'invasivité de l'approche

L'invasivité était un facteur important lors de la conception des différentes techniques. En effet, contrairement à plusieurs outils détaillés dans la revue de littérature, qui instrumentent

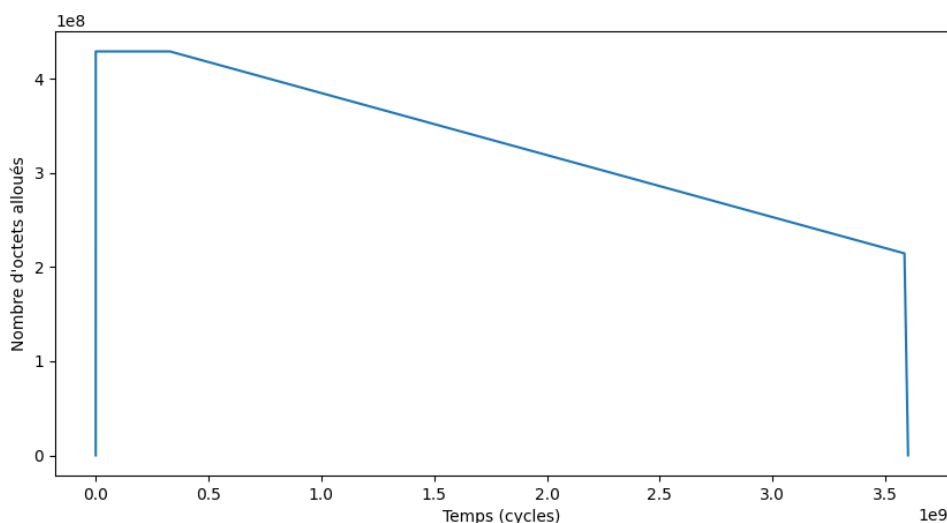


FIGURE 5.6 Nombre d'octets alloués dans le temps pour lbm

le programme lors de la compilation, nos 4 différentes techniques n'ont pas besoin de recompiler le programme. Elles utilisent toutes une technique qui consiste à charger une librairie partagée avant le programme, en définissant la variable d'environnement `LD_PRELOAD`. Aucune autre configuration n'est requise, outre le fait de compiler le programme avec l'option `-g`, qui s'assure de générer des informations de débogage.

Tel que mentionné plus haut, lorsqu'un appel système a un argument protégé (qu'il est protégé par la fonction `mprotect` ou par une étiquette de pointeur), il est impossible de le déprotéger avec la procédure par défaut, puisqu'il ne va pas générer de signal `SIGSEGV`. Un correctif peut être appliqué sur le noyau afin de déprotéger l'argument avant l'appel système, mais cette approche est très invasive, puisqu'il faut recompiler le noyau. Une approche alternative, qui consiste à définir des fonctions enveloppantes pour tous les appels système, est donc généralement préférable. Cette approche est toutefois assez fastidieuse à implémenter, et son implémentation n'est pas complétée pour tous les appels système dans le prototype actuel. Cette limitation est explorée plus en détail dans la section Conclusion de ce mémoire.

5.4 Application des techniques à d'autres architectures

Les travaux de ce mémoire se concentrant sur l'application des techniques d'analyse de mémoire sous l'architecture `x86_64`, l'application de ces techniques sur une autre architecture peut être envisageable. Cependant, il faut tenir compte de certaines considérations. En effet, si une instruction d'une architecture plus simple comme ARM permet de récupérer l'adresse

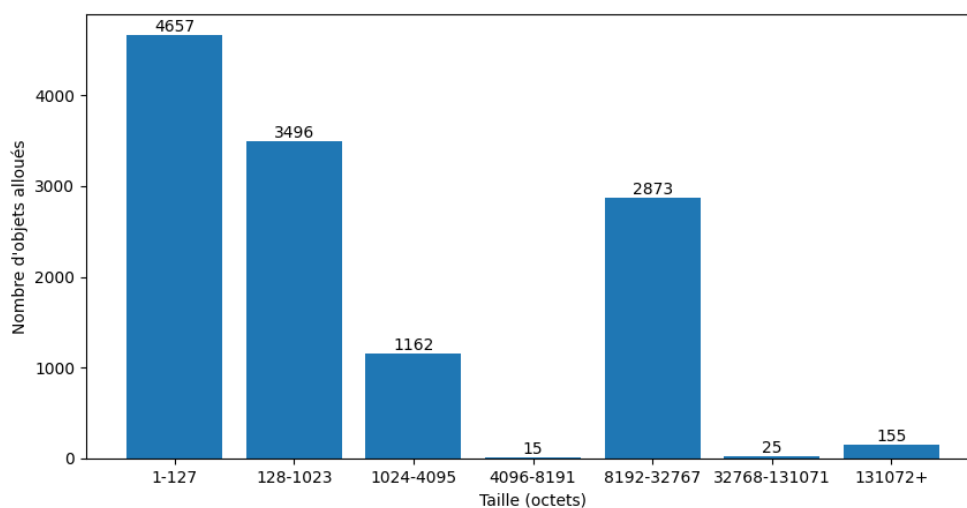


FIGURE 5.7 Distribution des tailles d'objets alloués pour blender

d'un accès mémoire plus facilement (comme vu dans la sous-section *Disassembly* du chapitre 4), dans le cas des approches utilisant *LibOLX* et *LibPatch*, il faut s'assurer que ces bibliothèques supportent cette architecture. Bien que le support pour l'architecture ARM soit prévu pour *LibPatch*, il pourrait être plus difficile d'assurer le support d'autres architectures pour ces bibliothèques. L'implémentation de ces techniques sur des architectures plus simples, nécessitant moins de traitement par *Capstone*, *LibOLX* et *LibPatch*, pourrait résulter en de meilleures performances pour l'outil présenté dans ce mémoire.

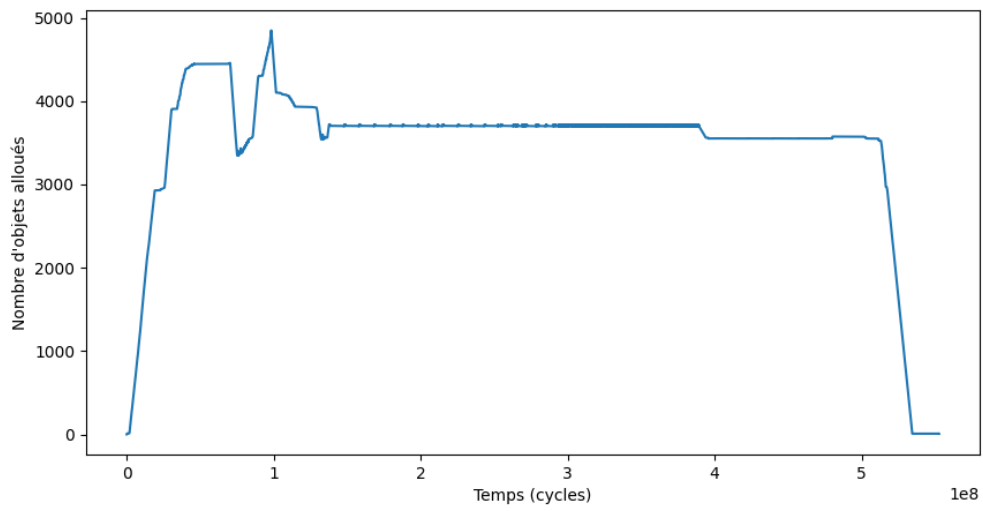


FIGURE 5.8 Nombre d'objets alloués dans le temps pour blender

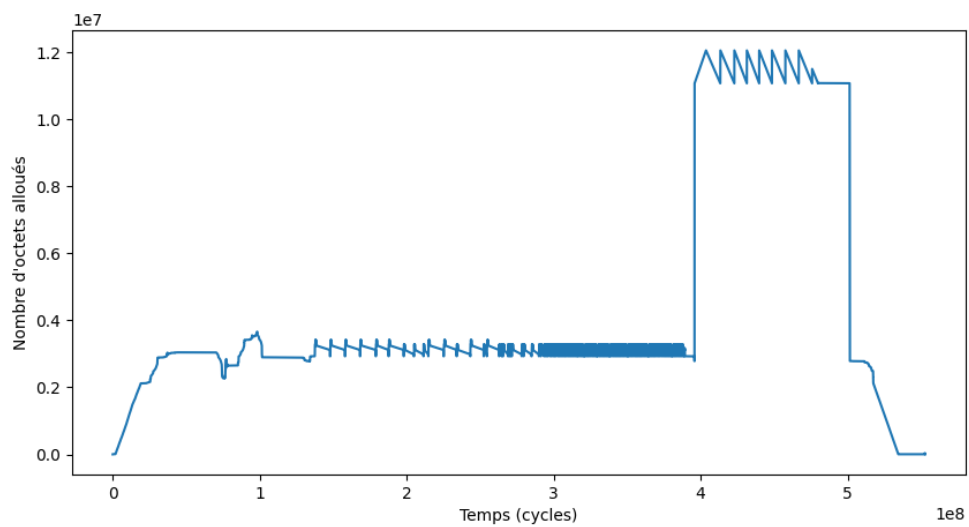


FIGURE 5.9 Nombre d'octets alloués dans le temps pour blender

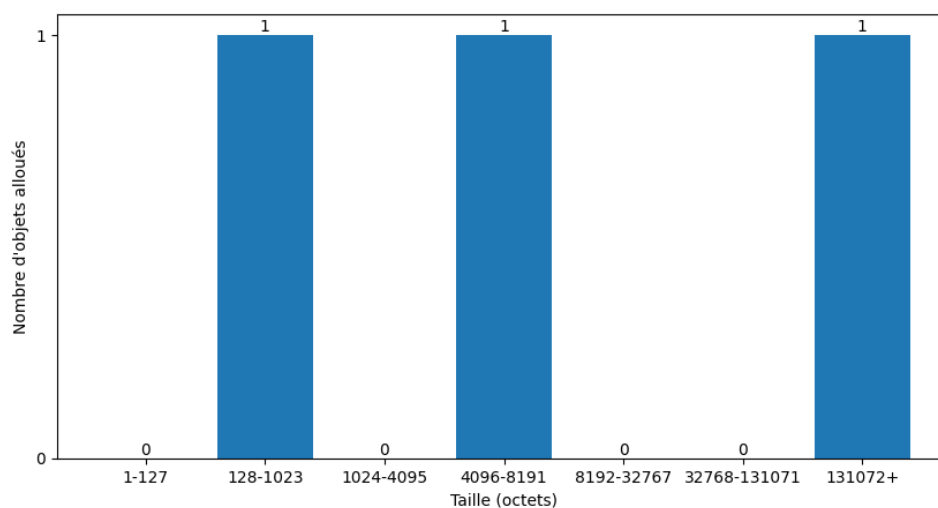


FIGURE 5.10 Distribution des tailles d'objets alloués pour deepsjeng

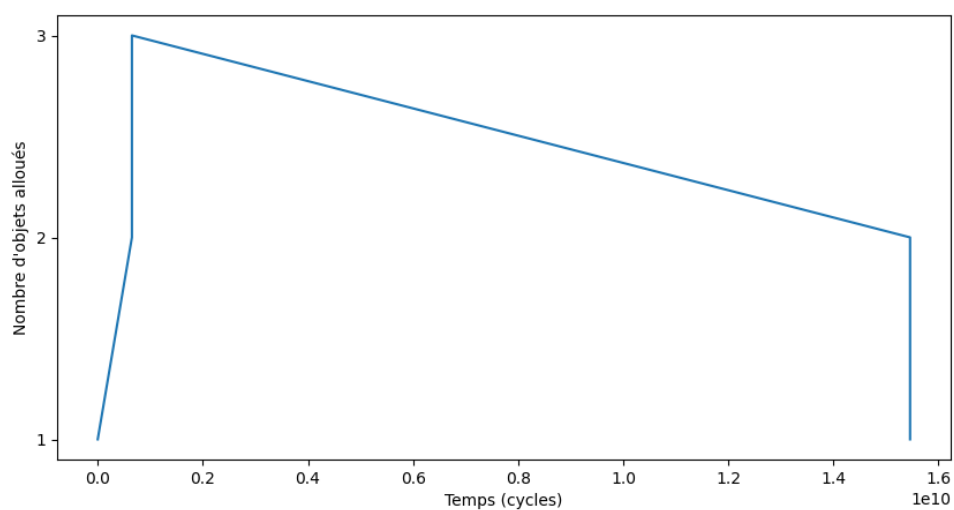


FIGURE 5.11 Nombre d'objets alloués dans le temps pour deepsjeng

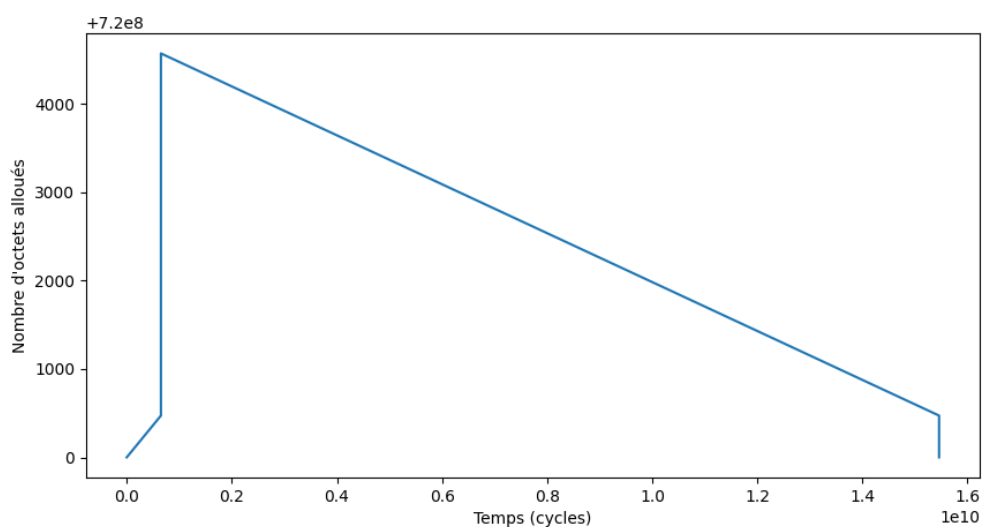


FIGURE 5.12 Nombre d'octets alloués dans le temps pour deepsjeng

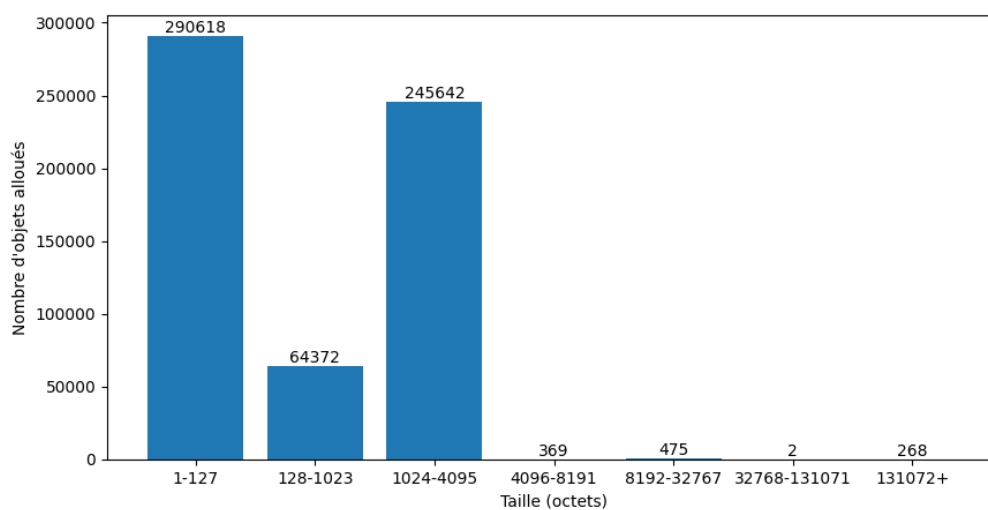


FIGURE 5.13 Distribution des tailles d'objets alloués pour leela

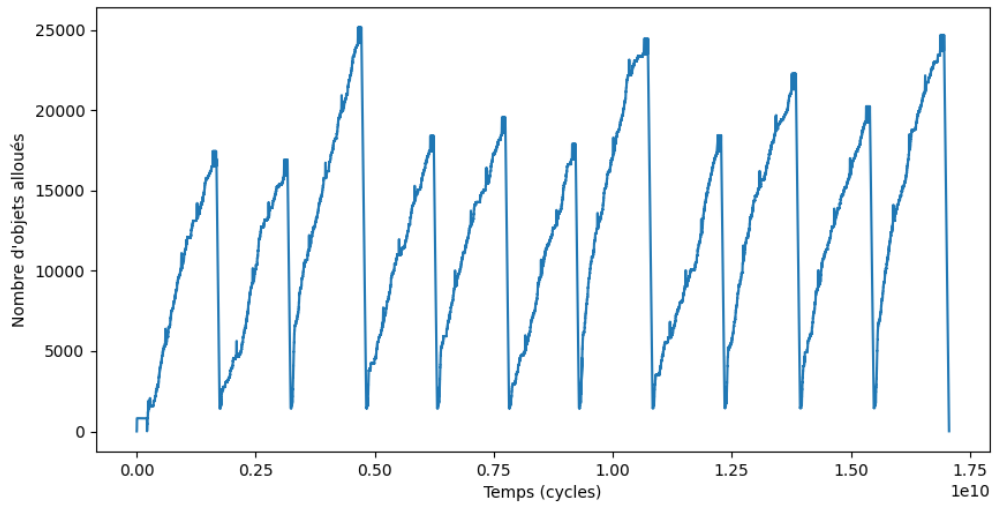


FIGURE 5.14 Nombre d'objets alloués dans le temps pour leela

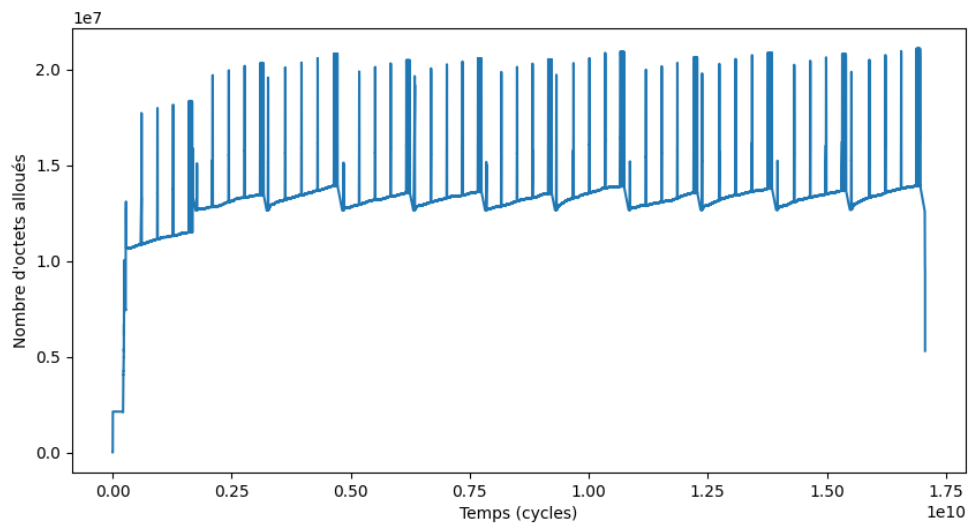


FIGURE 5.15 Nombre d'octets alloués dans le temps pour leela

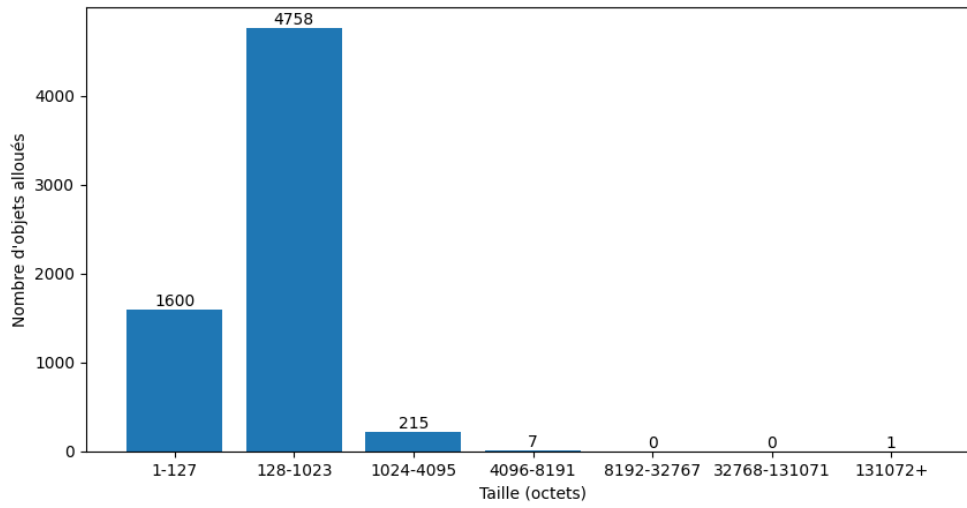


FIGURE 5.16 Distribution des tailles d'objets alloués pour nab

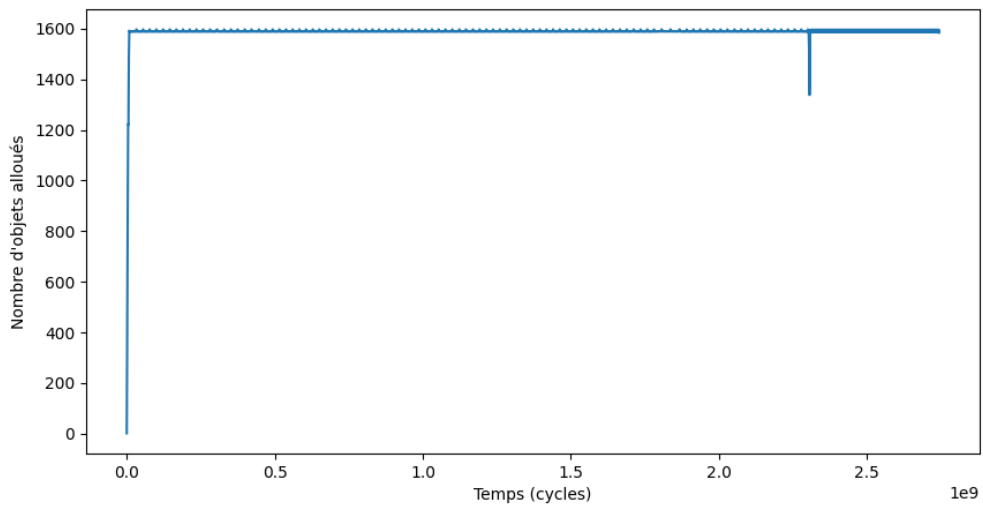


FIGURE 5.17 Nombre d'objets alloués dans le temps pour nab

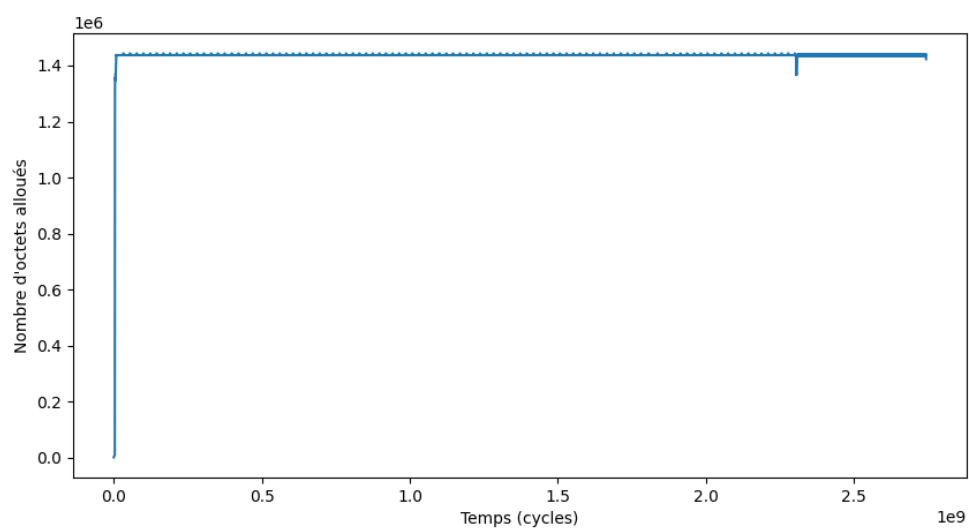


FIGURE 5.18 Nombre d'octets alloués dans le temps pour nab

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Ce mémoire présente plusieurs techniques qui ont été conçues pour la vérification d'erreurs de mémoire en temps réel pour les systèmes embarqués :

- Une technique utilisant `mprotect` avec le drapeau "PROT_NONE" pour la protection des zones de mémoire ainsi que *PTrace*
- Une technique utilisant une protection par pointeurs taintés ainsi que *PTrace*
- Une technique utilisant une protection par pointeurs taintés ainsi que *Libpatch* pour l'insertion de probes
- Une technique utilisant une protection par pointeurs taintés ainsi que *Libolx* pour l'émulation de probes

Une implémentation partielle a été réalisée pour chacune de ces 4 techniques.

Bien que l'approche utilisant *Libpatch*, la plus prometteuse en termes de surcoût en temps, ait toujours un surcoût de temps plus élevé que des outils comme *AddressSanitizer* si l'on protège chaque objet, ce surcoût peut être réduit significativement en sélectionnant un sous-ensemble de mémoire à protéger, par exemple, selon la taille de l'objet alloué. Pour chaque approche développée utilisant les pointeurs taintés, le surcoût en mémoire reste négligeable, et l'invasivité du système est grandement limitée.

6.2 Limitations de la solution proposée

Les différentes techniques présentent plusieurs limitations, ce qui explique le fait que les implémentations sont partielles.

En ce moment, lors de la protection de la mémoire utilisant `mprotect`, l'allocation est arrondie à la page supérieure. Ceci implique qu'une allocation de 8 octets prendra plutôt une page (4 Ko), ce qui représente un très grand surcoût en mémoire. Assumant 3 allocations de 16 octets, l'implémentation actuelle alloue une page pour chaque objet alloué, pour un total de 3 pages de mémoire. Cette implémentation servait surtout à estimer le surcoût en temps, à récolter des statistiques sur le nombre et la distribution des allocations dynamiques de mémoire, ainsi que le nombre d'instructions les accédant avec leur fréquence.

Lors de l'utilisation de pointeurs taintés, les arguments des appels système peuvent être taintés, ce qui pose problème, puisqu'ils ne lancent pas de signal, contrairement aux autres

accès mémoire. Un module noyau peut être utilisé pour nettoyer le pointeur avant l'exécution de l'appel système, mais cette approche demeure invasive. Des fonctions d'emballage peuvent aussi être utilisées, afin d'envelopper les appels système et ainsi s'assurer d'enlever l'étiquette. Par contre, notre couverture par fonctions d'emballage devra graduellement être complétée, afin de pouvoir traiter un plus grand nombre d'applications.

6.3 Améliorations futures

Heureusement, en dépit des nombreuses limitations soulevées, plusieurs améliorations peuvent être envisagées.

Lors de la protection de la mémoire utilisant `mprotect`, afin de minimiser le surcoût occasionné par la limitation soulevée plus haut, les allocations protégées peuvent être disposées de sorte que plusieurs objets puissent partager une page. Des zones rouges peuvent être arrangées entre chaque allocation, afin d'assurer une protection contre les accès hors bornes.

Pour ce qui est des fonctions d'emballage, pour les appels système recevant des pointeurs taintés, *Valgrind* [22] détaille qu'il a 15,000 lignes de code visant à prendre en compte chaque appel système et fournir des fonctions d'emballage pour chacune d'entre elles. Prendre inspiration de l'approche de *Valgrind* peut rendre la tâche de fournir des fonctions d'emballage pour chaque appel système moins fastidieuse.

Finalement, notre implémentation actuelle protège par défaut chaque allocation. Il sera intéressant de regarder plus en détail, pour différents programmes, comment la sélection d'un sous-ensemble d'allocations à protéger permettra en pratique de limiter le surcoût, tout en atteignant les objectifs de détection d'erreurs de mémoire. Pour ce faire, il faudra prendre certains paramètres en compte, par exemple la taille de l'objet, afin de déterminer quel sous-ensemble d'allocations est le plus susceptible aux erreurs de mémoire sur des systèmes embarqués réels utilisés en industrie.

Ce travail a permis de mettre en place les bases pour un coffre à outils servant à vérifier les accès mémoire d'applications en C/C++. En plus de caractériser les patrons d'allocations et d'accès aux objets alloués, les deux nouvelles techniques proposées, OLX-taint et Patch-taint, offrent une solution avancée et efficace pour la vérification à l'exécution, de manière sélective, des objets alloués dynamiquement.

RÉFÉRENCES

- [1] B. Perens, “efence(3).” [En ligne]. Disponible : <https://linux.die.net/man/3/efence>
- [2] K. Serebryany *et al.*, “Addresssanitizer : A fast address sanity checker,” dans *USENIX ATC 2012*, 2012.
- [3] J. Seward et N. Nethercote, “Using valgrind to detect undefined value errors with bit-precision.” USENIX Annual Technical Conference, avr. 2005. [En ligne]. Disponible : <https://valgrind.org/docs/memcheck2005.pdf>
- [4] N. Nethercote et J. Seward, “How to shadow every byte of memory used by a program,” 06 2007, p. 65–74.
- [5] D. Bruening et Q. Zhao, “Practical memory checking with Dr. Memory,” dans *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, p. 213–223.
- [6] K. Serebryany *et al.*, “Memory Tagging and how it improves C/C++ memory safety,” 2018. [En ligne]. Disponible : <https://arxiv.org/pdf/1802.09517.pdf>
- [7] T. Jim *et al.*, “Cyclone : A Safe Dialect of C,” dans *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC ’02. USA : USENIX Association, 2002, p. 275–288.
- [8] G. J. Duck et R. H. C. Yap, “Heap bounds protection with low fat pointers,” dans *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. New York, NY, USA : Association for Computing Machinery, 2016, p. 132–142. [En ligne]. Disponible : <https://doi.org/10.1145/2892208.2892212>
- [9] G. Duck, R. Yap et L. Cavallaro, “Stack bounds protection with low fat pointers,” 01 2017.
- [10] A. Kwon *et al.*, “Low-Fat Pointers : Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security,” dans *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13. New York, NY, USA : Association for Computing Machinery, 2013, p. 721–732. [En ligne]. Disponible : <https://doi.org/10.1145/2508859.2516713>
- [11] E. Q. Vintila, P. Zieris et J. Horsch, “MESH : A Memory-Efficient Safe Heap for C/C++,” dans *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ser. ARES 21. New York, NY, USA : Association for Computing Machinery, 2021. [En ligne]. Disponible : <https://doi.org/10.1145/3465481.3465760>
- [12] Valgrind, *Valgrind Official Documentation*, 2023. [En ligne]. Disponible : <https://valgrind.org/docs/manual/mc-manual.html>

- [13] Standard Performance Evaluation Corporation, “SPEC CPU 2006,” 2006. [En ligne]. Disponible : <https://www.spec.org/cpu2006/>
- [14] S. Nagarakatte *et al.*, “Softbound : Highly compatible and complete spatial memory safety for c,” vol. 44, 06 2009, p. 245–258.
- [15] A. Rogers *et al.*, “Supporting dynamic data structures on distributed-memory machines,” *ACM Trans. Program. Lang. Syst.*, vol. 17, n^o. 2, p. 233–263, mar 1995. [En ligne]. Disponible : <https://doi.org/10.1145/201059.201065>
- [16] Oracle, *SPARC ADI Documentation*, 2022. [En ligne]. Disponible : https://docs.oracle.com/cd/E88353_01/html/E37853/adi-7.html
- [17] V. Kuznetsov *et al.*, “Code-Pointer integrity,” dans *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO : USENIX Association, oct. 2014, p. 147–163. [En ligne]. Disponible : <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov>
- [18] EEMBC, “Core mark,” 2023. [En ligne]. Disponible : <https://www.eembc.org/coremark/>
- [19] M. Kerrisk, “mprotect(2),” 2023. [En ligne]. Disponible : <https://man7.org/linux/man-pages/man2/mprotect.2.html>
- [20] P. Akritidis *et al.*, “Baggy bounds checking : An efficient and backwards-compatible defense against out-of-bounds errors.” 01 2009, p. 51–66.
- [21] Standard Performance Evaluation Corporation, “SPEC CPU 2000,” 2007. [En ligne]. Disponible : <https://www.spec.org/cpu2000/>
- [22] N. Nethercote et J. Seward, “Valgrind : a framework for heavyweight dynamic binary instrumentation,” *ACM Sigplan Notices*, vol. 42, n^o. 6, p. 89–100, 2007.
- [23] D. e. a. Bruening, “An Infrastructure for Adaptive Dynamic Optimization,” dans *Proceedings of CGO’03*, mars 2003, p. 265–276.
- [24] C. Lattner et V. Adve, “LLVM : a compilation framework for lifelong program analysis & transformation,” dans *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, p. 75–86.
- [25] J. Puncher et F. Tétreault, “Data watch.” Tracing Summit, oct. 2017. [En ligne]. Disponible : <https://tracingsummit.org/ts/2017/files/TS17-datawatch.pdf>
- [26] O. Dion, “LibPatch,” 2023. [En ligne]. Disponible : <https://git.sr.ht/~old/libpatch>
- [27] —, “LibOlx,” 2023. [En ligne]. Disponible : <https://git.sr.ht/~old/libolx>
- [28] J. Puncher, “Data watch,” *Ciena*. [En ligne]. Disponible : tracingsummit.org/ts/2017/files/TS17-datawatch.pdf, 2017.

- [29] P. Naert, S. V. Azhari et M. Dagenais, “Interactive and targeted runtime verification using a debugger-based architecture,” *Journal of Systems Architecture*, vol. 115, p. 102001, 2021.
- [30] “Capstone : The Ultimate Disassembler,” 2023. [En ligne]. Disponible : <http://www.capstone-engine.org/>
- [31] Standard Performance Evaluation Corporation, “SPEC CPU 2017,” 2017. [En ligne]. Disponible : <https://www.spec.org/cpu2017>
- [32] “DWARF Version 5,” 2023. [En ligne]. Disponible : <https://dwarfstd.org/dwarf5std.html>
- [33] Standard Performance Evaluation Corporation, “505.mcf_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/505.mcf_r.html
- [34] —, “519.lbm_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/519.lbm_r.html
- [35] —, “526.blender_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/526.blender_r.html
- [36] —, “531.deepsjeng_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/531.deepsjeng_r.html
- [37] —, “541.leela_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/541.leela_r.html
- [38] —, “544.nab_r SPEC CPU®2017 Benchmark Description,” 2017. [En ligne]. Disponible : https://www.spec.org/cpu2017/Docs/benchmarks/544.nab_r.html