POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Détection à faible surcoût des conditions de course et des violations de la sûreté de la mémoire de tas

FARZAM DOROSTKAR
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Octobre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Détection à faible surcoût des conditions de course et des violations de la sûreté de la mémoire de tas

présentée par **Farzam DOROSTKAR**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

François-Raymond BOYER, président
Michel DAGENAIS, membre et directeur de recherche
Heng LI, membre et codirecteur de recherche
Maxime LAMOTHE, membre
Shervin VAKILI, membre externe

DÉDICACE

À mes parents et à ma sœur...

REMERCIEMENTS

Je tiens à exprimer ma plus profonde gratitude à mon directeur de thèse, le professeur Michel Dagenais, et à mon co-directeur de thèse, le professeur Heng Li, pour leurs conseils inestimables et leur soutien indéfectible tout au long de mon parcours doctoral. Les travaux de recherche présentés dans cette thèse n'auraient pas été possibles sans leur encouragement et leur mentorat dévoué.

Je souhaite également adresser mes sincères remerciements à nos partenaires industriels, en particulier Ericsson et Ciena, pour leur collaboration. Leur soutien financier et la perspective critique qu'ils ont apportée sur les défis concrets de l'industrie ont eu un impact profond sur la structuration des travaux de recherche présentés dans cette thèse.

RÉSUMÉ

Parmi les outils existants de détection des conditions de course et des violations de la sûreté spatiale et temporelle de la mémoire de tas, les techniques de vérification dynamique se distinguent par leur précision et leur meilleure capacité de passage à l'échelle par rapport aux approches statiques. Toutefois, malgré ces atouts, les outils dynamiques les plus perfectionnés restent difficilement utilisables dans de nombreux contextes de test réels, en raison de leur forte consommation en mémoire et en ressources de calcul. Cette limitation freine leur adoption dans des environnements contraints en ressources, tels que les systèmes embarqués, et complique les tests effectués sous des charges réalistes. En outre, même les détecteurs les plus répandus présentent souvent des lacunes de détection, ce qui met en lumière la nécessité de développer de nouvelles approches capables d'élargir leur couverture sans alourdir le coût en temps d'exécution ou en mémoire.

En conséquence, cette thèse poursuit deux objectifs principaux : concevoir de nouveaux détecteurs dynamiques à faible surcoût pour les conditions de course et les violations de la sûreté de la mémoire de tas, et améliorer la couverture d'un détecteur existant sans accroître le surcoût en temps d'exécution ou en mémoire. Ces objectifs visent à surmonter les principales limites des approches actuelles et ont guidé les trois axes de recherche développés au cours de cette thèse.

Le premier axe de recherche a examiné dans quelle mesure les avancées matérielles récentes en matière de traçage d'exécution pouvaient être exploitées pour concevoir un détecteur de conditions de course (data races) à faible surcoût, adapté aux environnements à ressources limitées, tels que les systèmes embarqués, tout en conservant les capacités de détection offertes par les outils existants à fort surcoût. Dans cette optique, nous avons conçu et mis en œuvre ThreadMonitor (TMon), un détecteur post-mortem de conditions de course destiné aux programmes C et C++ multithreadés utilisant la bibliothèque Pthread. TMon a été spécifiquement développé pour offrir une couverture de détection équivalente à celle de ThreadSanitizer (TSan), tout en réduisant considérablement le surcoût d'exécution. Pour cela, TMon repose sur deux phases principales : une phase de traçage légère et une phase d'analyse post-mortem. Durant la phase de traçage, TMon utilise les paquets `ptwrite` d'Intel, une fonctionnalité de traçage matériel récemment répandue dans Intel Processor Trace (Intel PT), afin de tracer les mêmes événements que ceux suivis par TSan, avec un très faible impact sur les performances. Grâce à ces paquets, TMon enregistre pour chaque événement exactement les mêmes informations d'exécution que celles collectées par TSan à des

fins d'analyse. Par la suite, l'analyseur post-mortem de TMon reconstruit la séquence des événements à partir des traces afin de déterminer si l'exécution observée présente des conditions de course. TMon n'induit aucun surcoût direct sur la mémoire de données, introduit un surcoût minimal sur la mémoire d'instructions, et engendre un ralentissement très limité. Nos évaluations expérimentales montrent qu'en moyenne, TMon entraîne un surcoût d'exécution 2 à 6 fois inférieur à celui de TSan, y compris en prenant en compte le coût d'écriture des traces sur disque.

Le deuxième axe de recherche a visé à évaluer s'il était possible de réaliser une détection exhaustive des violations temporelles et spatiales d'accès à la mémoire de tas dans les programmes C en combinant la propagation dynamique de métadonnées associées aux pointeurs avec une analyse effectuée à la compilation, tout en limitant le surcoût induit. Dans ce cadre, nous avons développé AddressMonitor (AMon), un outil de vérification à l'exécution qui associe le marquage des pointeurs à une analyse statique et à des transformations réalisées lors de la compilation. Plus précisément, AMon permet de détecter les accès hors limites (out-of-bound accesses), les utilisations après libération (use-after-frees), les doubles libérations (double-frees) ainsi que les fuites de mémoire (memory leaks). Il identifie également deux pratiques de programmation à risque liées à la gestion mémoire : le passage d'un pointeur non-base aux fonctions standard de réallocation ou de libération, ainsi que les utilisations potentielles après libération survenant à la suite d'une réallocation.

À la compilation, AMon identifie les opérandes pointeurs alloués sur le tas, génère des variantes non marquées, et remplace les pointeurs d'origine en conséquence. À l'exécution, il attribue une valeur de marquage unique à chaque objet alloué, insère cette valeur dans le pointeur, et maintient une table des objets permettant de suivre les informations spatiales, temporelles et de débogage pendant toute l'exécution du programme. Nos évaluations expérimentales montrent qu'AMon présente un meilleur compromis entre surcoût en temps d'exécution et surcoût mémoire que les outils de pointe existants. Son efficacité et sa pertinence pratique ont été confirmées par son intégration réussie dans un système informatique embarqué propriétaire déployé par notre partenaire industriel Ericsson.

Le troisième axe de recherche a étudié comment les compromis d'implémentation visant à réduire le surcoût dans les détecteurs dynamiques de conditions de course peuvent engendrer des angles morts de détection non documentés, et si de telles limitations peuvent être atténuées sans augmenter le temps d'exécution ni la consommation mémoire. Nous nous sommes penchés sur TSan, un détecteur de conditions de course de pointe intégré aux chaînes de compilation Clang et GCC. Nous avons analysé son sous-système de mémoire d'ombre — responsable de l'enregistrement et de la vérification de l'historique des accès mémoire — et

mis en évidence deux angles morts jusqu'alors non documentés : le premier dû à l'éviction aléatoire lors du remplacement des entrées en mémoire d'ombre, le second lié au caractère non atomique de la vérification des courses et de la mise à jour des valeurs d'ombre. Nous avons formalisé les conditions dans lesquelles ces angles morts se manifestent, et proposé des stratégies d'atténuation qui permettent d'améliorer la couverture de détection sans modifier le surcoût initial de l'outil.

ABSTRACT

Compared to static approaches, dynamic verification techniques offer superior detection accuracy and scalability for identifying data races and heap-related memory safety violations. However, despite these advantages, state-of-the-art dynamic tools often introduce significant runtime and memory overhead, which limits their practicality in real-world testing scenarios—particularly in resource-constrained environments such as embedded systems, or when evaluating software under realistic workloads. Furthermore, even widely adopted detectors exhibit detection blind spots resulting from implementation-level compromises, highlighting the need for novel mitigation strategies that broaden detection coverage without increasing overhead.

This thesis addresses these challenges through two primary objectives: the design of novel low-overhead dynamic detectors for data races and heap-related memory safety violations, and the enhancement of detection coverage in an existing dynamic tool without introducing additional performance cost. These objectives directly target key limitations in current approaches and form the basis of the three research tracks pursued in this work.

The first research track examined whether recent hardware support for execution tracing could be leveraged to design a low-overhead data race detector suitable for resource-constrained environments, while preserving the detection coverage of existing high-overhead tools. To this end, we developed ThreadMonitor (TMon), a postmortem data race detector for multithreaded C and C++ programs using the Pthread library. TMon is specifically designed to match the detection capabilities of ThreadSanitizer (TSan)—a state-of-the-art race detector integrated into both the Clang and GCC toolchains—while significantly reducing the runtime overhead. It features two main phases: a lightweight tracing phase and a postmortem analysis phase. During tracing, TMon utilizes the low-overhead `ptwrite` packet, a hardware-assisted tracing feature provided by Intel Processor Trace, to record the same program events monitored by TSan. This mechanism captures, with minimal overhead, the essential runtime information required for race detection. The postmortem analyzer then reconstructs the execution trace to determine whether any data races occurred. TMon incurs no direct data memory overhead, introduces only minimal instruction memory overhead, and causes a negligible performance impact. Experimental results show that TMon imposes 2× to 6× less execution time overhead than TSan, including the cost of trace generation.

The second research track investigated whether comprehensive detection of temporal and spatial heap access violations in C programs could be achieved by combining dynamic meta-

data propagation with compile-time analysis, without incurring high runtime overhead. To address this, we designed and implemented AddressMonitor (AMon), a runtime verification tool that integrates pointer tainting with compile-time analysis and code transformations. Specifically, AMon detects out-of-bound accesses, use-after-frees, double-frees, and memory leaks. Additionally, it is able to identify two other unsafe programming practices related to heap access: passing a non-base pointer to standard memory reallocation or deallocation functions, and potential use-after-free conditions arising from memory reallocation. At compile-time, AMon identifies heap pointer operands, generates their untainted counterparts, and replaces the original pointers with these variants. At runtime, it assigns a unique taint value to each allocated object, embeds it into the pointer, and maintains an object table to track spatial, temporal, and debugging information throughout execution. Our evaluation studies demonstrate that AMon offers a more favorable trade-off between runtime and memory overhead than existing state-of-the-art tools. The practicality of AMon has been demonstrated through its successful integration into a proprietary embedded computing system deployed by our industry partners at Ericsson.

The third research track explored how implementation-level tradeoffs in dynamic data race detectors—introduced to reduce runtime and memory overhead—can lead to undocumented detection blind spots, and whether such limitations can be addressed without compromising performance. Focusing on TSan, we analyzed its shadow memory subsystem responsible for recording and validating memory access histories. We identified two previously undocumented detection blind spots in this tool: one resulting from random eviction in the shadow slot replacement policy, and the other from the non-atomicity of race checking and shadow value updates. We formally characterized the conditions under which these detection blind spots arise and proposed mitigation strategies that improve detection coverage without affecting the initial overhead of the tool.

In conclusion, this thesis makes an important contribution in achieving low-overhead, high-coverage dynamic identification of data races and memory safety violations, advancing state-of-the-art tools. The industry adoption of the research outcomes further proves the practical significance of the contributions of this thesis.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AMon	AddressMonitor
ASan	AddressSanitizer
CWE	Common Weakness Enumeration
ELF	Executable and Linkable Format
Intel PT	Intel Processor Trace
SSA	Static Single Assignment
TMon	ThreadMonitor
TSan	ThreadSanitizer

CHAPITRE 1 INTRODUCTION

Dans les programmes multithreadés, les threads partagent généralement l'accès à une partie ou à l'ensemble de la mémoire de l'application. Bien que la mémoire partagée permette une communication efficace entre les threads, elle expose également un programme multithreadé aux *conditions de course* (*data races*). Une condition de course survient lorsque deux threads ou plus accèdent à la même adresse mémoire, sans synchronisation ni exclusion mutuelle, et qu'au moins l'un de ces accès est une opération d'écriture. Lorsqu'une condition de course se produit, le comportement à l'exécution du programme peut être influencé par l'ordre temporel des accès constituant cette condition. Une condition de course est considérée comme une erreur de concurrence, à moins que le non-déterminisme qui en résulte ne soit un choix de conception.

L'identification des conditions de course constitue une tâche particulièrement ardue, en raison de l'enchevêtrement complexe des interdépendances entre threads, du potentiel d'exécution non déterministe et de la complexité inhérente de la gestion de la mémoire partagée, en particulier dans des langages tels que C et C++, où la gestion manuelle de la mémoire, les comportements indéfinis et les sémantiques d'accès mémoire de bas niveau accentuent encore cette difficulté. Ces réalités font de la détection des conditions de course un défi de recherche à la fois persistant et crucial pour garantir la correction des programmes multithreadés.

Bien que les conditions de course constituent une manifestation critique, propre aux environnements concurrents, d'interactions mémoire non sécurisées, la corruption de mémoire dans les programmes en C dépasse largement ce cadre.

Le langage de programmation C offre aux développeurs un contrôle explicite sur la mémoire. Cela inclut notamment la possibilité d'accéder à la mémoire de manière arbitraire par manipulation directe des pointeurs et déréférencement, ainsi que la gestion explicite de la durée de vie des zones mémoire allouées dynamiquement, ce qui permet d'éviter le recours à un ramasse-miettes automatique coûteux. Bien que ce contrôle de bas niveau soit particulièrement avantageux dans les domaines nécessitant des performances élevées, tels que la programmation système, il s'accompagne de l'absence de mécanismes intégrés garantissant la sûreté des accès mémoire. Cette lacune constitue une source bien connue d'erreurs dans les programmes en C, menant fréquemment à des vulnérabilités représentant des risques de sécurité importants.

Les conséquences graves associées aux violations de la sûreté des accès mémoire ont motivé de nombreux travaux de recherche dans le domaine de la sûreté mémoire. Bien que des stratégies

bien établies aient considérablement renforcé, au fil du temps, la protection de la mémoire de pile (stack memory), les efforts visant à garantir la sûreté des accès à la mémoire de tas (heap memory) sont demeurés relativement moins aboutis et moins efficaces. Cette limitation a entretenu un intérêt marqué, tant dans le milieu académique que dans l'industrie, pour le développement d'outils destinés à détecter les violations de la sûreté des accès à la mémoire de tas.

1.1 Objectifs de la recherche

Parmi les outils existants pour la détection des conditions de course et des violations de la sûreté spatiale et temporelle de la mémoire de tas, les techniques de vérification dynamique sont reconnues pour leur précision de détection et leur capacité de passage à l'échelle supérieures à celles des approches statiques. Néanmoins, en dépit de ces avantages, les outils de vérification dynamique les plus avancés demeurent impraticables dans de nombreux scénarios de test réels, en raison de leur important coût en mémoire et en ressources de calcul. Cette contrainte limite leur applicabilité dans des environnements à ressources restreintes, tels que les systèmes embarqués, et pose des défis majeurs lors de tests effectués dans des conditions de charge réalistes. De plus, même les détecteurs les plus largement utilisés présentent souvent des zones d'ombre en matière de détection, ce qui souligne la nécessité de concevoir de nouvelles méthodes permettant d'élargir leur couverture sans accroître la surcharge en temps d'exécution ni en mémoire.

En conséquence, cette thèse se concentre sur deux objectifs principaux : la conception de nouveaux détecteurs dynamiques à faible surcoût pour les conditions de course et les violations de la sûreté de la mémoire liées au tas, ainsi que l'élargissement de la couverture d'un détecteur existant sans entraîner de surcoût supplémentaire en temps d'exécution ou en mémoire. Ces deux objectifs répondent directement aux principales limites des approches actuelles et constituent les axes centraux de cette recherche.

Plus précisément, cette thèse s'attache à répondre aux questions de recherche suivantes : **QR1.** Comment les avancées matérielles récentes en matière de traçage d'exécution peuvent-elles être exploitées pour concevoir un nouveau détecteur de conditions de course à faible surcoût destiné aux programmes C et C++ multithreadés, garantissant une adéquation avec des environnements de test contraints en ressources tout en offrant une couverture de détection comparable à celle des outils existants de pointe mais à fort surcoût ? **QR2.** Comment les techniques dynamiques de propagation de métadonnées associées aux pointeurs peuvent-elles être combinées avec une analyse à la compilation afin de permettre une détection exhaustive des violations temporelles et spatiales d'accès à la mémoire de tas dans les programmes

C, tout en maintenant un faible surcoût ? **QR3.** Comment les compromis d'implémentation visant à réduire le surcoût d'exécution et mémoire donnent-ils lieu à des angles morts de détection non documentés dans les détecteurs dynamiques de conditions de course existants, et est-il possible d'atténuer ces angles morts sans introduire de surcoût supplémentaire ?

1.2 Contributions originales

Nous avons poursuivi les objectifs présentés ci-dessus à travers trois axes principaux de recherche, dont les contributions majeures sont introduites ci-après.

1. **ThreadMonitor : Détection à faible surcoût de conditions de course à l'aide d'Intel Processor Trace :** Dans le premier axe de cette thèse, nous avons conçu et mis en œuvre *ThreadMonitor* (TMon), un détecteur postmortem à faible surcoût de conditions de course pour les programmes C et C++ multithreadés utilisant la bibliothèque Pthread. À l'exécution, TMon trace les informations nécessaires à la détection d'occurrences de conditions de course (c'est-à-dire les accès à la mémoire partagée et les contraintes temporelles entre threads) en s'appuyant sur *Intel Processor Trace* (Intel PT), une fonctionnalité matérielle non intrusive dédiée au traçage de l'exécution logicielle. Par la suite, son analyseur postmortem examine les données de trace collectées afin de déterminer si l'exécution du programme tracé présentait des conditions de course. TMon n'entraîne aucun surcoût direct en mémoire de données, occasionne un surcoût minimal en mémoire d'instructions et induit un ralentissement très faible, ce qui en fait un choix idéal dans les environnements de test aux ressources limitées.
2. **AddressMonitor : Sûreté spatiale et temporelle de la mémoire de tas à faible surcoût pour le C :** Dans le cadre de notre deuxième axe de recherche, nous avons conçu et implémenté *AddressMonitor* (AMon), un outil de vérification à l'exécution à faible surcoût, destiné à détecter un large éventail de violations temporelles et spatiales des accès à la mémoire de tas dans les programmes en C. AMon repose sur une combinaison de marquage des pointeurs (*pointer tainting*) et d'analyses et transformations de code effectuées à la compilation. Il est spécifiquement conçu pour offrir des capacités de détection exhaustives tout en conservant un faible surcoût, garantissant ainsi sa pertinence pour un déploiement dans des environnements contraints en ressources, tels que les systèmes embarqués. Une illustration concrète de cette pertinence est l'intégration réussie d'AMon par nos partenaires industriels chez Ericsson dans l'un de leurs systèmes informatiques embarqués propriétaires internes.
3. **Identification et atténuation des angles morts de détection des conditions**

de course induits par l'implémentation dans ThreadSanitizer v3 : Dans notre troisième projet de recherche, nous examinons ThreadSanitizer (TSan) v3, la version la plus récente de ce détecteur de conditions de course largement utilisé, intégré à la fois dans LLVM et GCC, et mettons en évidence des angles morts de détection jusqu'alors non documentés, résultant de décisions prises au niveau de l'implémentation. Nous caractérisons formellement les conditions dans lesquelles ces angles morts apparaissent et illustrons comment ils conduisent TSan à produire des faux négatifs. En outre, nous proposons des stratégies d'atténuation visant à réduire la probabilité que ces angles morts se produisent, sans introduire de surcoût en temps d'exécution ou en mémoire par rapport à la version originale de TSan.

1.3 Plan de la thèse

Le reste de cette thèse est organisé comme suit. Le chapitre 2 présente une revue de la littérature couvrant les concepts clés et les travaux existants en lien avec nos travaux de recherche. Le chapitre 3 détaille la méthodologie employée pour atteindre les objectifs de recherche dans chacun des trois volets de cette thèse. Les chapitres 4, 5 et 6 présentent nos contributions dans chacun des volets de recherche, structurées sous la forme des trois articles soumis en tant que résultats respectifs de ces volets. Le chapitre 7 présente une discussion générale des résultats obtenus. Enfin, le chapitre 8 conclut cette thèse par un résumé de nos constatations et contributions, et présente également les pistes potentielles pour des travaux futurs.

1.4 Publications

Les travaux de recherche menés dans cette thèse ont donné lieu à trois articles de recherche, chacun correspondant à une piste de recherche distincte, comme indiqué ci-dessous.

1. Farzam Dorostkar, Michel Dagenais, Ankush Tyagi et Vince Bridgers, "ThreadMonitor : Low-overhead data race detection using Intel Processor Trace", révision soumise à Concurrency and Computation : Practice and Experience
2. Farzam Dorostkar, Michel Dagenais, Heng Li, Vince Bridgers, Ankush Tyagi et Vladislav Aranov, "AddressMonitor : Low-overhead heap spatial and temporal safety for C", soumis à Journal of Systems and Software
3. Farzam Dorostkar, Michel Dagenais, Heng Li, Vladislav Aranov et Ankush Tyagi, "Identifying and mitigating implementation-induced data race detection blind spots

in ThreadSanitizer v3", soumis à Concurrency and Computation : Practice and Experience

CHAPITRE 2 REVUE DE LITTÉRATURE

Les conditions de course et les violations de la sûreté de la mémoire comptent parmi les problèmes les plus persistants et les plus complexes dans les programmes en C et C++, représentant des risques importants pour la sécurité et la fiabilité des logiciels. La complexité de leurs manifestations et la difficulté de leur détection en ont fait un sujet d'intérêt majeur et durable pour la recherche. Ce chapitre présente principalement une revue ciblée de la littérature sur ces thématiques, constituant la base des contributions exposées dans cette thèse.

Nous commençons par la détection des conditions de course, organisée en quatre sections. La première définit le concept et expose les défis inhérents à son identification. La deuxième passe en revue les principales catégories d'approches de détection automatisée, tandis que la troisième examine les stratégies algorithmiques qui sous-tendent ces approches. La quatrième présente les détecteurs de conditions de course les plus connus pour les programmes multithreadés en C et C++.

La discussion se poursuit ensuite avec la sûreté des accès mémoire, abordée dans les trois sections suivantes. La première en présente les principes généraux, la deuxième examine les défis spécifiques liés à la sûreté des accès à la mémoire de tas, et la troisième passe en revue les principales catégories de techniques de détection, en mettant l'accent sur les programmes en C.

En outre, nous incluons une section dédiée à la présentation de l'Intel Processor Trace (Intel PT), cette fonctionnalité matérielle étant utilisée dans l'un des outils proposés dans cette thèse.

Nous concluons le chapitre en identifiant les principales limites des travaux existants ayant motivé notre recherche.

2.1 Condition de course

Par définition, deux opérations mémoire sont dites *en conflit* (*conflict*) si elles accèdent à la même adresse mémoire et qu'au moins l'un de ces accès est une opération d'écriture. Deux opérations mémoire en conflit forment une *condition de course* (*data race*) lorsqu'elles proviennent de threads distincts et qu'elles se produisent en l'absence de synchronisation ou d'exclusion mutuelle.

La localisation des conditions de course peut s'avérer extrêmement difficile pour les pro-

grammeurs, et ce pour deux raisons principales. La première difficulté consiste à s'assurer que chaque accès aux variables partagées respecte les contraintes temporelles appropriées. La réalisation de cette vérification peut être particulièrement complexe, même pour un projet légèrement sophistiqué. La seconde difficulté réside dans le fait qu'une condition de course ne conduit à la corruption des données partagées que dans le cadre d'entrelacements spécifiques des threads. Par conséquent, il est possible qu'un programmeur ne détecte pas une condition de course, même après des tests intensifs. De plus, une variable partagée corrompue ne provoque pas nécessairement une défaillance immédiate, mais peut se propager dans le programme et finir par se manifester sous forme de comportements inattendus ou d'erreurs dans d'autres parties de celui-ci. L'identification de la cause première de telles anomalies peut alors s'avérer particulièrement ardue.

2.2 Détection automatisée de conditions de course

Les difficultés évoquées précédemment concernant la localisation des conditions de course ont suscité un intérêt considérable pour la recherche et le développement d'outils automatisés visant à détecter ces conditions. De manière générale, les détecteurs de conditions de course peuvent être classés en deux grandes catégories : les outils statiques et les outils dynamiques. Certains outils adoptent une approche hybride afin d'améliorer à la fois les performances et la précision.

Les outils statiques analysent le code source ou les représentations intermédiaires d'un programme, dans le but de détecter avec précision tous les entrelacements potentiels d'accès à la mémoire partagée pouvant conduire à des conditions de course. Les approches statiques se heurtent à des limites intrinsèques lorsqu'il s'agit d'analyser avec exactitude des constructions du langage impliquant un comportement dynamique ou des interactions complexes, notamment les pointeurs, l'allocation dynamique de mémoire, la création dynamique de threads, les mécanismes de concurrence et les structures de contrôle complexes. L'un des défis récurrents réside dans la précision du modèle mémoire utilisé. Par exemple, l'analyseur statique clang effectue son analyse sur l'*Abstract Syntax Tree* (AST) et utilise une approximation analytique de la mémoire qui souffre d'une analyse d'alias imprécise.

L'efficacité constitue également un enjeu majeur. Bien que l'avantage principal de l'analyse statique, par rapport à l'analyse dynamique, réside dans sa capacité à offrir une couverture maximale du code grâce à un examen complet de l'ensemble du programme, l'analyse minutieuse de tous les chemins d'exécution possibles s'avère computationnellement irréalisable [2, 3]. Ces contraintes compromettent inévitablement la précision et l'utilisabilité des outils statiques. Pour garantir la justesse (*soundness*), les détecteurs statiques de conditions

de course adoptent généralement un certain niveau de conservatisme dans leurs techniques d'analyse, ce qui peut, en contrepartie, générer un nombre excessif de fausses alertes [4–7]. Trouver un équilibre optimal entre la réduction des faux négatifs et la limitation des faux positifs constitue donc un obstacle fondamental dans la détection statique des conditions de course.

Pour atténuer ce problème, certains outils s'appuient sur des annotations fournies par le programmeur, servant d'indices explicites pour guider l'outil vers des rapports plus précis [4, 8–10]. L'application de ces outils, qui sont donc moins automatisés, à des bases de code relativement volumineuses s'avère peu pratique [11]. D'autres outils appliquent des algorithmes heuristiques afin de filtrer les fausses alertes parmi les avertissements générés. Cependant, il existe toujours un risque d'éliminer par erreur des avertissements légitimes de conditions de course [12, 13].

Les outils dynamiques utilisent des mécanismes de traçage pour surveiller les événements d'intérêt qui se produisent au cours d'une exécution particulière du programme. Leur objectif principal est d'examiner si le chemin d'exécution observé présente des conditions de course manifestes. Les outils dynamiques se classent en deux catégories : les outils *on-the-fly* et les outils post-mortem. Les techniques *on-the-fly* analysent en ligne le flux des événements du programme, tandis que les approches post-mortem les enregistrent dans un fichier journal et reportent l'analyse de détection des conditions de course à la fin de l'exécution du programme.

Les principales informations d'exécution à surveiller sont : les accès à la mémoire partagée, l'allocation et la désallocation dynamiques de mémoire, l'invocation de primitives de gestion de threads imposant des contraintes temporelles entre threads, ainsi que la pile d'appels (*call stack*) de chaque thread. Pour les collecter, les approches dynamiques s'appuient couramment sur une instrumentation invasive du programme, entraînant un ralentissement significatif et une surcharge mémoire, au point où même les détecteurs dynamiques de conditions de course les plus avancés ne peuvent être utilisés dans de nombreux scénarios de test réels.

Pour atténuer ce problème, certaines approches dynamiques recourent à l'échantillonnage des accès mémoire [14–16]. Étant donné que la fréquence des accès mémoire est plus élevée que celle des autres événements d'intérêt, n'en analyser qu'un sous-ensemble peut réduire la surcharge à l'exécution. Toutefois, cette approche introduit un risque de manquer des conditions de course réelles, puisque tous les accès mémoire ne sont pas examinés.

Une autre contrainte de la détection dynamique des conditions de course est la couverture limitée du code. En effet, tous les détecteurs dynamiques de conditions de course sont intrinsèquement sujets à des faux négatifs, car à chaque exécution, ils n'examinent qu'un seul chemin parmi les nombreuses trajectoires possibles d'un programme. Autrement dit, seuls

les chemins de code exercés par les bancs de tests (*test harnesses*) sont analysés. Bien qu'en théorie il soit possible d'atténuer cette limitation en explorant différents ordonnancements des événements observés afin de détecter des conditions de course potentielles, sa mise en œuvre pratique pose des défis importants [17]. Évaluer toutes les permutations des événements observés est computationnellement irréalisable et peut entraîner des fausses alertes, car tous les ordonnancements d'événements ne sont pas valides. Il est donc nécessaire de concevoir un mécanisme permettant d'identifier les ordonnancements valides au moyen d'une analyse complète du programme.

Pour surveiller l'exécution d'un programme, les outils modernes de débogage dynamique de la concurrence s'appuient couramment sur l'instrumentation logicielle. Dans cette approche, également appelée *traçage logiciel* (*software-based tracing*), l'outil insère du code d'instrumentation dans le programme en cours de test afin d'observer son exécution à travers les événements générés par les instructions injectées. Bien que le traçage logiciel offre une flexibilité considérable pour la surveillance, il entraîne généralement un surcoût d'exécution important par rapport à une exécution native.

Ces dernières années, de nombreux processeurs commerciaux ont intégré un matériel dédié à la traçabilité de l'exécution, désigné sous le terme de *traçage assisté par matériel* (*hardware-assisted tracing*). L'exploitation de ces fonctionnalités matérielles — telles que *Intel Processor Trace* (Intel PT) [18] et *ARM CoreSight* [19] — permet d'obtenir une surveillance détaillée de l'exécution avec un impact minimal sur les performances à l'exécution. Néanmoins, malgré sa disponibilité croissante et son faible surcoût, le traçage assisté par matériel a reçu relativement peu d'attention en tant qu'alternative aux approches conventionnelles de traçage logiciel dans les outils de débogage existants.

Malgré les limitations mentionnées ci-dessus, les détecteurs dynamiques de conditions de course ont suscité plus d'attention que les détecteurs statiques, en raison de leur précision et de leur capacité de passage à l'échelle supérieure.

2.3 Algorithmes de détection de conditions de course

L'algorithme fondé sur la *relation de précédence (happens-before)* [20], l'algorithme *Lockset* [21], ainsi que leurs variantes, constituent les méthodes les plus couramment utilisées pour détecter les conditions de course, aussi bien dans les outils dynamiques que statiques. Les deux premières parties de cette sous-section sont consacrées à l'étude de ces deux types d'algorithmes de détection des conditions de course. Nous expliquerons comment ils identifient ces erreurs, ainsi que les limitations qui leur sont associées. Dans la troisième partie de cette

section, nous examinerons des exemples d'algorithmes hybrides de détection des conditions de course. Ces algorithmes combinent généralement les approches fondées sur la relation de précédence et sur Lockset, ou certaines de leurs variantes.

2.3.1 Algorithme de relation de précédence

L'idée principale de la méthode de détection des conditions de course fondée sur la relation de précédence consiste à examiner s'il est possible de définir une relation de *précédence* entre deux accès mémoire en conflit. La relation de précédence, introduite par Lamport [20], est un ordre partiel sur les événements se produisant dans un système distribué basé sur l'échange de messages. Comme mentionné précédemment, une condition de course se produit lorsque des accès mémoire en conflit, issus de différents threads, peuvent survenir de manière *concurrente*. Par conséquent, une étape essentielle dans la détection des conditions de course consiste à déterminer si deux accès mémoire sont effectivement concurrents, si l'on souhaite s'appuyer sur la définition générale d'une condition de course (plutôt que sur une hypothèse plus restrictive, comme celle utilisée par l'algorithme Lockset, présenté à la section 2.3.2). Comme nous le verrons, la notion de relation de précédence, introduite par Lamport, peut être appliquée avec succès à la programmation multithread afin de définir un ordre partiel sur les événements d'accès mémoire. Pour ce faire, nous commencerons par examiner la définition originale de cette relation de précédence proposée par Lamport [20].

Définition 1 (Ordre partiel de précédence proposé par Lamport)**.** *La relation de précédence, notée $\rightarrow$, est un ordre partiel sur les événements dans un système distribué. Dans le cas d'un système distribué basé sur l'échange de messages, elle est définie comme suit pour deux événements distincts a et b :*

- *Si a et b sont des événements d'un même processus, et que a précède b, alors $a \rightarrow b$.*
- *Si a est l'événement d'envoi d'un message par un processus, et b l'événement de réception de ce même message par un autre processus, alors $a \rightarrow b$.*
- *S'il existe un événement c tel que $a \rightarrow c$ et $c \rightarrow b$, alors $a \rightarrow b$.*

Deux événements distincts a et b sont dits concurrents s'il n'existe pas de relation de précédence entre eux, c'est-à-dire si $a \nrightarrow b$ et $b \nrightarrow a$. Autrement dit, il n'est pas possible de définir un ordre partiel de précédence entre ces deux événements.

Comme l'avait anticipé Lamport [20], le concept d'ordre partiel fondé sur la relation de précédence peut également être appliqué à d'autres domaines que l'informatique distribuée. Cette définition peut être adaptée de manière à établir un ordre partiel sur les événements dans un environnement de programmation multithread.

Définition 2 (Ordre partiel de précédence modifié)**.** *Dans un programme multithread, la relation de précédence, notée* $\rightarrow$*, est définie comme suit pour deux événements distincts* a *et* b *:*

- *Si* a *et* b *sont des événements d'un même thread, et que* a *précède* b*, alors* $a \rightarrow b$*.*
- *Si* (a, b) *forme une paire de synchronisation, alors* $a \rightarrow b$*.*
- *S'il existe un événement* c *tel que* $a \rightarrow c$ *et* $c \rightarrow b$*, alors* $a \rightarrow b$*.*

Deux événements distincts a *et* b *sont dits concurrents si* $a \nrightarrow b$ *et* $b \nrightarrow a$*.*

Dans la Définition 2, une paire de synchronisation désigne essentiellement toute opération de synchronisation utilisée pour imposer certaines contraintes d'ordre sur les événements exécutés par différents threads. Par conséquent, cette notion dépend du langage de programmation utilisé. Une opération de synchronisation courante est l'opération de verrouillage, dans laquelle les threads doivent acquérir un verrou d'exclusion mutuelle pour accéder à une zone mémoire partagée. La figure 2.1 illustre comment une paire de synchronisation (en l'occurrence, le déverrouillage puis le verrouillage d'un même verrou) impose une relation de précédence.

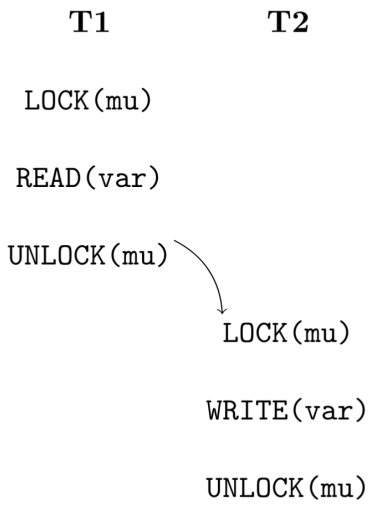

FIGURE 2.1 Exemple d'une paire de synchronisation définissant une relation de précédence

Dans cette trace, le thread $T1$ lit une variable partagée *var* tout en détenant un verrou d'exclusion mutuelle *mu*. Après que le thread $T1$ a libéré *mu*, le thread $T2$ l'acquiert et écrit sur *var*. Le déverrouillage de *mu* par $T1$, suivi de son verrouillage par $T2$, établit une relation de précédence entre les deux threads à cet endroit précis (la flèche orientée), car les accès mémoire effectués par $T1$ avant la libération du verrou doivent précéder ceux réalisés par $T2$ après son acquisition.

Une relation de précédence peut également résulter d'autres opérations de synchronisation. Par exemple, en langage C, lorsqu'un thread signale une variable de condition et qu'un autre thread attend ce signal, une relation de précédence peut être établie entre les deux threads. En plus d'être dépendantes du langage, les opérations de synchronisation considérées peuvent varier selon les outils de détection de conditions de course. Par exemple, Helgrind [22] et ThreadSanitizer [23] sont deux détecteurs de conditions de course de pointe, basés sur la relation de précédence, conçus pour les programmes C/C++ utilisant les primitives POSIX pthreads.

Bien qu'elle soit largement utilisée, la relation de précédence de base présentée ci-dessus présente des limitations importantes en tant que méthode de détection des conditions de course. Elle n'est ni sûre, ni complète [24]. Dans [25], les auteurs soutiennent que la relation de précédence classique n'est valide que pour la première condition de course détectée. De plus, comme nous le verrons à la section 2.3.2, la méthode fondée sur la relation de précédence pour détecter une condition de course est fortement sensible à l'ordonnancement.

2.3.2 Algorithme Lockset

La méthode de détection des conditions de course fondée sur Lockset repose sur une discipline de verrouillage simple, selon laquelle chaque zone mémoire partagée doit être protégée de manière cohérente par au moins un verrou tout au long de l'exécution du programme. L'algorithme Lockset de base a été proposé dans [21]. À titre d'exemple introductif, considérons l'entrelacement des threads illustré à la figure 2.2, où les instructions sont exécutées de haut en bas dans un programme comportant deux threads.

Dans cette exécution, le thread $T2$ verrouille le mutex mu après qu'il ait été déverrouillé par $T1$. Cela établit une relation de précédence entre $T1$ et $T2$ à ce moment de l'exécution, car les accès mémoire effectués par $T1$ avant le déverrouillage du mutex sont, par définition, considérés comme antérieurs à ceux de $T2$ après l'acquisition du mutex. Ainsi, du point de vue d'un détecteur de conditions de course basé sur la relation de précédence (sous-section 2.3.1), les deux écritures sur la variable x sont ordonnées — c'est-à-dire que l'écriture de $T1$ sur x précède celle de $T2$ — et, par conséquent, aucune condition de course n'est signalée. Cependant, il s'agit ici d'un faux négatif. Il existe une condition de course potentielle sur les accès non synchronisés à x, mais l'algorithme de relation de précédence ne parvient pas à la détecter à partir de cette exécution. Il ne pourrait identifier la condition de course que si le segment de code de $T2$ était planifié avant celui de $T1$. Cet exemple illustre une fois de plus la forte sensibilité de l'algorithme de relation de précédence à l'ordonnancement. En revanche, comme nous le verrons, l'algorithme Lockset est capable de détecter la condition de course

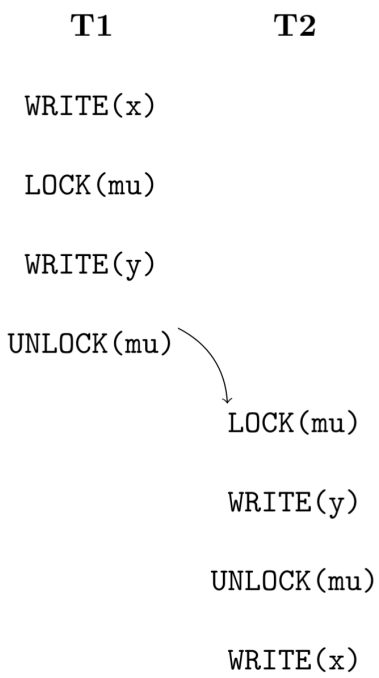

FIGURE 2.2 Exemple d'un entrelacement de threads pour lequel l'algorithme de relation de précédence produit un faux négatif

de la figure 2.2 indépendamment de l'entrelacement des threads choisi par le planificateur.

L'algorithme Lockset impose une discipline de verrouillage simple, selon laquelle chaque objet partagé doit être protégé par un verrou donné, ce qui implique que tout thread souhaitant accéder à une variable partagée doit au préalable acquérir ce verrou [26]. L'algorithme Lockset surveille tous les accès mémoire lors de l'exécution du programme et vérifie que cette discipline est bien respectée [27]. Il tente d'inférer, à partir de l'historique d'exécution, quels verrous sont supposés protéger quels objets partagés, car il a été conçu à l'origine pour être intégré dans un outil entièrement dynamique appelé *Eraser* [21].

Pour chaque variable partagée v, l'algorithme maintient l'ensemble de tous les verrous candidats susceptibles de la protéger. Cet ensemble est noté $C(v)$, et il contient tous les verrous qui ont protégé v de manière cohérente au cours de l'exécution du programme. Par définition, un verrou l appartient à $C(v)$ si, jusqu'à l'instant considéré dans l'exécution, chaque thread ayant accédé à v détenait le verrou l au moment de l'accès. Lorsqu'une variable v est initialisée, on considère que $C(v)$ contient tous les verrous possibles. Lors d'un accès à la variable, l'algorithme met à jour $C(v)$ en prenant l'intersection entre $C(v)$ et l'ensemble des verrous actuellement détenus par le thread réalisant l'accès. Cette procédure, appelée *raffinement de l'ensemble de verrous* (lockset refinement), garantit que tout verrou protégeant systématiquement la variable v restera présent dans son ensemble candidat $C(v)$ à mesure

que celui-ci est raffiné. Une conséquence possible de ce raffinement est que $C(v)$ devienne vide s'il n'existe aucun verrou protégeant systématiquement la variable v. L'algorithme 1 présente le fonctionnement de l'algorithme Lockset pour la détection des conditions de course [21].

Algorithm 1 Algorithme Lockset

1: Soit $locks(t)$ l'ensemble des verrous détenus par le thread t.
2: Pour chaque variable partagée v, initialiser $C(v)$ avec l'ensemble de tous les verrous.
3: À chaque accès à v par le thread t :
4: $C(v) := C(v) \cap locks(t)$ ▷ Effectuer le raffinement de l'ensemble de verrous.
5: **if** $C(v) = \emptyset$ **then**
6: Émettre un avertissement.
7: **end if**

La figure 2.3 présente un entrelacement de threads similaire à celui illustré précédemment dans la figure 2.2, avec quelques modifications apportées aux deux accès à la variable x afin de mieux illustrer la capacité de l'algorithme Lockset à détecter les violations de la discipline de verrouillage simple. Dans cette figure, les deux colonnes de gauche montrent les instructions exécutées, de haut en bas, par deux threads $T1$ et $T2$. Les colonnes $locks(T1)$ et $locks(T2)$ représentent les verrous détenus respectivement par $T1$ et $T2$ après l'exécution de chaque instruction de verrouillage ou de déverrouillage. Les colonnes $C(x)$ et $C(y)$ contiennent les ensembles de verrous candidats pour x et y respectivement, mis à jour après chaque accès mémoire. Le programme utilise deux verrous au total : $mu1$ et $mu2$. Ainsi, $C(x)$ et $C(y)$ sont initialisés à $mu1, mu2$. Lorsque $T1$ accède à x, le seul verrou qu'il détient est $mu1$. Par conséquent, après cet accès, $C(x)$ est raffiné à $mu1$ ($mu1, mu2 \cap mu1$). De même, lors de l'accès à y, $T1$ détient uniquement $mu2$, ce qui entraîne la mise à jour de $C(y)$ à $mu2$. Le thread $T2$ détient $mu2$ lors de ses accès à y et à x. Ainsi, $C(y)$ reste $mu2$ ($mu2 \cap mu2$), tandis que $C(x)$ devient l'ensemble vide ($mu1 \cap mu2$), ce qui indique que la variable x n'est pas protégée de manière cohérente par un verrou. À ce stade, l'algorithme Lockset émet un avertissement de condition de course sur x, puisqu'aucun verrou ne la protège systématiquement.

Le principal inconvénient de l'algorithme Lockset de base réside dans sa dépendance exclusive à la discipline de verrouillage. Il s'agit d'une hypothèse très stricte, qui peut entraîner de nombreux faux positifs, car toute violation de cette hypothèse ne correspond pas nécessairement à une erreur de programmation [28] [29].

2.3.3 Algorithmes hybrides

Pour gagner en sûreté et en précision, plusieurs détecteurs de conditions de course combinent les approches fondées sur la relation de précédence et sur Lockset [30]. Dans ce qui suit, nous

T1	**T2**	$locks(T1)$	$locks(T2)$	$C(x)$	$C(y)$
		{}	{}	$\{mu1, mu2\}$	$\{mu1, mu2\}$
LOCK(mu1)		$\{mu1\}$			
WRITE(x)				$\{mu1\}$	
UNLOCK(mu1)		{}			
LOCK(mu2)		$\{mu2\}$			
WRITE(y)					$\{mu2\}$
UNLOCK(mu2)		{}			
	LOCK(mu2)		$\{mu2\}$		
	WRITE(y)				$\{mu2\}$
	WRITE(x)			{}	
	UNLOCK(mu2)		{}		

FIGURE 2.3 Exemple du processus de raffinement dans l'algorithme Lockset

passerons en revue deux algorithmes hybrides de détection de conditions de course.

Acculock est un algorithme hybride fondé sur les approches de relation de précédence et de Lockset [31]. Il utilise une détection de conditions de course basée sur les époques (epochs) pour définir des arêtes de précédence faibles dans une exécution, indépendamment de la discipline de verrouillage imposée par le programme. Si les arêtes ainsi définies révèlent une violation potentielle, l'algorithme applique ensuite une approche fondée sur Lockset afin de filtrer les accès mémoire protégés par un verrou. Bien que cette méthode améliore la précision de la détection des conditions de course, elle s'avère plus lente que l'algorithme Lockset de base lorsqu'elle est implémentée dans le même environnement logiciel [30].

Autre exemple, l'algorithme Goldilocks utilise la méthode Lockset pour définir des relations de précédence entre les événements mémoire [32]. Bien qu'il s'agisse fondamentalement d'un détecteur basé sur Lockset, Goldilocks est capable de prendre en charge d'autres disciplines de synchronisation, telles que les transactions. De plus, l'algorithme enregistre des métadonnées, comme l'identifiant du dernier thread (PID) ayant accédé à une zone mémoire partagée. Ces

métadonnées contribuent à améliorer la précision de l'algorithme. Goldilocks est également capable d'effectuer une analyse statique afin de réduire la surcharge en performance.

2.4 Outils de détection de conditions de course

Dans cette section, nous passons en revue divers outils de détection de conditions de course conçus pour les programmes C/C++ multithreadés. La première sous-section est consacrée à ThreadSanitizer (TSan), en mettant en évidence ses principaux composants ainsi que l'évolution de ce détecteur de conditions de course de pointe. La seconde sous-section présente un aperçu approfondi des autres outils proposés dans la littérature.

2.4.1 ThreadSanitizer (TSan)

ThreadSanitizer (TSan) est un détecteur dynamique de conditions de course à la pointe de la technologie, open source, qui prend actuellement en charge plusieurs architectures 64 bits. L'outil se compose de deux composants principaux : un module d'instrumentation et une bibliothèque d'exécution. Le module d'instrumentation annote les accès mémoire dans le code utilisateur par des appels à la bibliothèque d'exécution, chaque appel fournissant des informations d'exécution sur l'accès instrumenté et déclenchant une vérification de l'éventuelle présence d'une condition de course impliquant cet accès. En plus de mettre en œuvre la logique de détection des conditions de course, la bibliothèque d'exécution intercepte certaines fonctions de la bibliothèque standard, réparties en deux grandes catégories : celles qui concernent la gestion des threads, la coordination de l'exclusion mutuelle et l'application des mécanismes de synchronisation ; et celles qui réalisent des opérations d'accès mémoire. Les premières permettent à l'outil de capturer les contraintes temporelles entre les threads, un aspect essentiel pour l'analyse des conditions de course. Les secondes lui permettent de surveiller les accès à la mémoire partagée qui ne sont pas explicitement présents dans le code utilisateur, mais qui se produisent au sein des fonctions de la bibliothèque standard.

À ce jour, TSan a évolué à travers trois grandes versions, se distinguant notamment par leur approche en matière d'instrumentation du code et/ou de cadre d'analyse des conditions de course.

La première version de TSan (TSan v1) utilisait une instrumentation à l'exécution, avec une implémentation initiale basée sur PIN [33], suivie d'une transition vers Valgrind [34], qui offrait de meilleures performances [35]. Concernant l'algorithme de détection des conditions de course sous-jacent, TSan v1 proposait deux modes d'analyse : un mode hybride par défaut combinant les techniques de relation de précédence (happens-before) et de Lockset, ainsi

qu'un mode alternatif fondé exclusivement sur la relation de précédence. Bien que ce mode purement happens-before génère moins de faux positifs, son comportement est généralement moins prévisible, car sa capacité à détecter certaines véritables conditions de course dépend de l'ordre d'exécution spécifique des accès en conflit, déterminé de manière non déterministe par le planificateur [35, 36].

TSan v1 souffrait d'une surcharge importante à l'exécution, avec des ralentissements rapportés allant de 20× à 300× [37]. Cette surcharge était principalement attribuée à sa dépendance à Valgrind pour l'instrumentation, ce qui limitait également la portabilité de l'outil aux seules plateformes prises en charge par Valgrind [38]. Pour pallier ces limitations, la deuxième version de l'outil (TSan v2) a adopté une instrumentation à la compilation, intégrée directement au processus de construction du programme [38]. TSan v2 a initialement conservé la logique de détection des conditions de course de son prédécesseur, mais a ensuite évolué vers un mode d'analyse exclusivement basé sur la relation de précédence, en partie en raison de la tendance de l'analyse Lockset à produire des faux positifs dans le code sans verrou [39]. Dans sa bibliothèque d'exécution repensée, TSan v2 utilise un mécanisme d'horloges vectorielles afin d'évaluer l'existence d'une relation de précédence entre des accès mémoire en conflit. Il repose sur une mémoire dite shadow memory, qui conserve des informations sur les accès précédents à la mémoire. Dans ce modèle, chaque bloc de huit octets de la mémoire de l'application est directement mappé à un groupe composé typiquement de quatre cellules shadow. Chaque cellule shadow, d'une taille de huit octets, encode un accès mémoire et se compose de quatre champs : 16 bits pour identifier le thread ayant effectué l'accès, 42 bits représentant une capture de son horloge vectorielle au moment de l'accès, 5 bits codant la position des octets accédés, et 1 bit indiquant s'il s'agissait d'une écriture [37].

Dans [40], Sadowski et Yi interrogent un groupe de développeurs expérimentés chez Google afin de comprendre comment ils utilisent TSan v2 et ThreadSafety [10], un détecteur statique de conditions de course basé sur LLVM, dans différents scénarios de débogage. Leur étude empirique révèle que les participants considéraient que TSan v2 était plus efficace pour détecter des conditions de course difficiles à identifier.

La troisième et plus récente version de TSan (TSan v3) conserve la stratégie d'instrumentation à la compilation introduite dans TSan v2 et fait l'objet d'un maintien actif au sein des compilateurs LLVM et GCC [41, 42]. Son module d'instrumentation est implémenté sous forme de passes dédiées du compilateur, agissant respectivement sur LLVM IR et GIMPLE, les représentations intermédiaires indépendantes de l'architecture propres à chacun de ces compilateurs.

À l'instar de son prédécesseur, TSan v3 utilise une analyse exclusivement fondée sur la rela-

tion de précédence comme unique logique de détection des conditions de course. Toutefois, il introduit une refonte majeure de la bibliothèque d'exécution, visant principalement à réduire la surcharge à l'exécution ainsi que l'empreinte mémoire de l'outil [43]. Parmi les modifications apportées, TSan v3 réduit de moitié la taille de chaque valeur shadow par rapport à TSan v2, ce qui permet de diviser par deux le ratio entre la mémoire shadow et la mémoire de l'application. Dans TSan v3, sur l'architecture `x86_64`, chaque région mémoire de l'application, alignée sur 8 octets, est par défaut directement mappée à quatre valeurs shadow. Chaque valeur shadow occupe quatre octets et encode cinq composantes caractérisant un accès mémoire : un masque de 8 bits indiquant quels octets de la région correspondante ont été accédés, 8 bits enregistrant l'identifiant du thread ayant effectué l'accès, 14 bits représentant son horloge logique au moment de l'accès, 1 bit spécifiant s'il s'agit d'un accès en écriture, et 1 bit indiquant s'il s'agit d'une opération atomique.

2.4.2 Autres détecteurs de conditions de course

Helgrind [22] est un autre détecteur dynamique de conditions de course open source largement utilisé. Il s'agit d'un outil de la suite Valgrind, spécialisé dans la détection des erreurs liées à la gestion des threads, y compris les conditions de course, dans les programmes C/C++ et Fortran utilisant la bibliothèque Pthread. La logique de détection de Helgrind repose sur la relation de précédence (happens-before). L'outil surveille tous les accès mémoire ainsi que les primitives de synchronisation qui imposent des contraintes temporelles entre les threads, notamment les opérations de verrouillage et de déverrouillage de mutex, la création et la jonction de threads, les variables de condition et les sémaphores. Les informations collectées sont utilisées pour construire un graphe orienté représentant les dépendances happens-before entre les événements du programme. Lorsqu'il identifie deux threads accédant à la même zone mémoire, Helgrind détermine s'il est possible d'établir une relation de précédence entre les événements respectifs, en examinant l'existence d'un chemin dans le graphe happens-before reliant les deux accès. Helgrind entraîne une surcharge importante à l'exécution, se traduisant généralement par un ralentissement considérable de l'ordre de 100×.

DILP [44] est un outil d'analyse postmortem conçu pour détecter des conditions de course dans les pilotes de périphériques Linux. À l'exécution, il enregistre les contextes d'appel des accès aux variables globales, aux arguments de fonction de type pointeur, ainsi qu'aux variables influencées par ces derniers. En raison de sa dépendance exclusive à l'analyse Lockset, l'outil est incapable d'interpréter les directives de synchronisation. DILP impose une surcharge significative à l'exécution, avec un ralentissement moyen rapporté d'environ 7.2×.

LiteRace [14] est un outil dynamique qui exploite des techniques d'échantillonnage afin de

réduire la surcharge à l'exécution. L'étude propose un algorithme d'échantillonnage adaptatif fondé sur l'hypothèse des cold regions, selon laquelle les conditions de course sont plus susceptibles de se produire dans les régions du code accédées peu fréquemment. L'algorithme commence avec un taux d'échantillonnage de 100% pour toutes les régions du code, puis réduit progressivement ce taux pour les régions dites chaudes. L'outil génère une version instrumentée de chaque fonction, permettant l'enregistrement de tous les accès mémoire. De plus, il intègre un test dynamique à l'entrée de chaque fonction. Ce test, basé sur des informations d'échantillonnage propres à chaque thread, détermine si l'exécution doit se faire dans la version instrumentée ou non instrumentée de la fonction.

RACEZ [15] est un autre outil basé sur l'échantillonnage, dans lequel l'échantillonnage des accès mémoire est réalisé en s'appuyant sur l'unité matérielle de surveillance des performances (PMU – Performance Monitoring Unit), plutôt que sur l'instrumentation de chaque accès mémoire. Les adresses sont extraites à partir des échantillons fournis par la PMU. L'outil repose sur une analyse de type Lockset, dans laquelle les primitives de verrouillage et de déverrouillage doivent être instrumentées.

Ces dernières années, certaines études ont exploré le potentiel de l'apprentissage automatique pour la détection des conditions de course. Par exemple, DeepRace [45] propose une approche fondée sur l'apprentissage profond afin de détecter et de mettre en évidence les conditions de course dans du code source C utilisant la bibliothèque Pthread ou OpenMP. L'outil s'appuie sur un réseau de neurones convolutif à une seule couche pour apprendre de manière autonome les motifs associés aux conditions de course. Actuellement, l'outil est limité à la détection de trois formes de contraintes d'accès mémoire. Dans le cas du code utilisant la bibliothèque Pthread, il peut uniquement identifier les mécanismes d'exclusion mutuelle. Pour le code reposant sur OpenMP, il est capable de détecter uniquement l'utilisation de la clause `private` ainsi que des directives de section critique.

2.5 Sûreté des accès mémoire

Il existe deux aspects fondamentaux de la sécurité des accès mémoire qui s'appliquent aussi bien à la mémoire de tas (heap memory) qu'à la mémoire de pile (stack memory) : *la sécurité spatiale* (*spatial safety*) et *la sécurité temporelle* (*temporal safety*).

La sécurité spatiale garantit que chaque accès mémoire respecte strictement les limites prévues de l'objet alloué. Une violation de la sécurité spatiale est communément appelée *erreur d'accès hors limites* (*out-of-bounds access error*). La sécurité temporelle, quant à elle, garantit que les objets ne sont pas accédés au-delà de leur durée de vie prévue. Deux types

courants de violations de la sécurité temporelle associés à la mémoire de tas sont *la double libération* (*double-free*) et *l'utilisation après libération* (*use-after-free*). Une erreur de double libération survient lorsqu'un programme tente de libérer plus d'une fois une région de mémoire allouée dynamiquement, tandis qu'une erreur d'utilisation après libération apparaît lorsqu'un programme accède à une zone mémoire après qu'elle a été libérée. Dans ce dernier cas, le pointeur qui référence encore la mémoire désallouée est appelé *pointeur suspendu* (*dangling pointer*). L'analogue en mémoire de pile d'une erreur d'utilisation après libération est *l'erreur d'utilisation après retour* (*use-after-return*), qui se produit lorsqu'une variable locale est accédée après le retour de la fonction qui l'avait allouée.

L'incapacité à maintenir la sécurité spatiale ou temporelle constitue une source importante de vulnérabilités de sécurité, car elle peut permettre à des attaquants d'accéder à des régions mémoire non autorisées. Cela peut avoir des conséquences graves, notamment la corruption ou le vol de données sensibles, ainsi que la compromission du flot de contrôle (control flow hijacking). En outre, de telles violations peuvent introduire des corruptions mémoire particulièrement difficiles à diagnostiquer. Ces corruptions peuvent ne pas provoquer de défaillance immédiate, mais plutôt se propager au cours de l'exécution du programme, éventuellement s'accumuler et, à terme, conduire à des comportements inattendus ou à des plantages dans d'autres parties du programme, rendant l'identification précise de la cause première extrêmement complexe.

2.6 Sûreté des accès à la mémoire de tas

Malgré plusieurs décennies de recherche, les violations de la sûreté mémoire demeurent un défi majeur dans le domaine de la vérification logicielle. Bien que la protection de la pile ait bénéficié de l'élaboration de techniques robustes et largement adoptées, notamment à travers des extensions spécifiques des compilateurs [46–48], les efforts visant à corriger les violations de la sûreté du tas n'ont pas atteint un niveau comparable d'efficacité pratique. Cette limite est confirmée par des rapports récents documentant des vulnérabilités dans des logiciels utilisés en production. Par exemple, la classification des causes profondes (root cause mapping) du Common Weakness Enumeration (CWE) appliquée au catalogue 2023 des vulnérabilités activement exploitées (Known Exploited Vulnerabilities, KEV), qui recense les failles couramment exploitées dans les systèmes logiciels largement déployés dans les domaines de la cybersécurité et des réseaux, met en évidence les erreurs d'utilisation après libération (CWE-416) et les débordements de tampon sur le tas (CWE-122) comme étant respectivement les première et deuxième faiblesses les plus fréquemment exploitées [49].

À titre d'exemple supplémentaire, le rapport 2023 sur les risques de sécurité des produits

publié par Red Hat indique que quatre des cinq types de faiblesses logicielles les plus fréquemment observées étaient des problèmes de corruption mémoire, affectant principalement Red Hat Enterprise Linux ainsi que d'autres bases de code écrites en C [50]. Il est à noter que ce rapport identifie l'erreur d'utilisation après libération comme la faiblesse la plus répandue parmi ces quatre types.

2.6.1 Approches pour la sécurité de la mémoire de tas

Les attaquants exploitent généralement les violations de la sécurité de la mémoire de tas en tirant parti des faiblesses d'implémentation des allocateurs mémoire dynamique sous-jacents. Par exemple, `ptmalloc`, l'allocateur de base de la bibliothèque GNU C, est vulnérable à des attaques par injection de code pouvant résulter d'accès hors limites dans le tas ou de tentatives de double libération [51]. En réponse à ces menaces, certains travaux de recherche orientés vers la sécurité proposent la conception d'allocateurs mémoire dynamiques personnalisés, offrant une meilleure résistance aux vulnérabilités découlant de violations de la sécurité de la mémoire de tas [52–55]. À titre d'exemple, FreeGuard [53] est un allocateur sécurisé qui vise à atténuer les vulnérabilités liées au tas en intégrant des allocations aléatoires, une isolation des objets et une protection des métadonnées. Toutefois, ces approches présentent plusieurs limitations qui freinent leur adoption à grande échelle, notamment une surcharge significative, une efficacité limitée à certains types d'attaques, et une granularité de protection restreinte.

Un autre ensemble de travaux explore les défenses assistées par le matériel pour empêcher les violations d'accès mémoire ou atténuer leurs conséquences en matière de sécurité. Ces approches combinent généralement des fonctionnalités matérielles avec des techniques logicielles afin de garantir une certaine sûreté des accès mémoire, tout en maintenant une surcharge réduite [56,57]. Toutefois, leur dépendance à des extensions matérielles spécifiques limite leur applicabilité. De plus, leur viabilité repose sur le maintien du support de ces technologies. Par exemple, l'abandon du support des Intel Memory Protection Extensions (MPX) [58], une extension matérielle conçue pour faciliter la vérification des bornes, a réduit la pertinence pratique des outils reposant sur cette technologie [59, 60].

Les méthodes de détection purement logicielles constituent une autre catégorie de techniques visant à identifier les violations d'accès mémoire. Ces méthodes reposent uniquement sur des mécanismes logiciels pour effectuer l'analyse des programmes [61]. Comparées aux approches assistées par le matériel, elles offrent une flexibilité accrue, puisqu'elles ne requièrent pas de support matériel spécialisé. Contrairement aux solutions basées sur des allocateurs personnalisés, ces techniques préservent en général la disposition mémoire originale des objets alloués, ce qui facilite leur intégration dans des bases de code existantes. De plus, les méthodes logi-

cielles permettent généralement de détecter un éventail plus large de types de violations [62], tout en offrant une granularité supérieure, ce qui améliore la précision et la profondeur de l'analyse.

Dans les environnements industriels, selon l'expérience de nos partenaires, les approches logicielles sont souvent privilégiées par rapport aux techniques mentionnées précédemment, en raison de leur adaptabilité, mieux alignée avec les exigences opérationnelles et économiques. Les organisations tendent à favoriser des solutions de détection généralisées, capables d'être déployées sur une grande diversité de plateformes logicielles et matérielles. Cette préférence repose sur deux avantages clés :

— Couverture unifiée de la détection : Une approche généralisée garantit des capacités de détection cohérentes sur différents systèmes, ce qui facilite la mise en place de pratiques de sécurité standardisées et améliore ainsi la prévisibilité des évaluations de vulnérabilité inter-systèmes.
— Développement et maintenance efficaces : L'adoption d'une méthode de détection généralisée permet aux entreprises de centraliser leurs efforts. Cette stratégie réduit les redondances et simplifie les flux de travail, en limitant la gestion d'outils distincts aux fonctionnalités chevauchantes, ce qui améliore l'efficacité opérationnelle et diminue les coûts.

Malgré ces avantages considérables, les outils logiciels existants présentent des limitations importantes qui restreignent leur applicabilité. L'une des principales limitations réside dans la surcharge qu'ils induisent généralement, ce qui les rend peu adaptés aux environnements à ressources limitées.

2.7 Analyses pour la détection des violations de la sûreté de la mémoire

Les conséquences critiques liées au non-respect de la sûreté mémoire, tant sur le plan spatial que temporel, ont suscité un intérêt marqué, tant dans le milieu académique qu'industriel, pour le développement d'outils automatisés permettant de détecter ces violations. Les outils existants sont généralement conçus pour identifier un sous-ensemble des violations de sûreté mémoire et ciblent des architectures matérielles ainsi que des plateformes logicielles spécifiques. La littérature propose une diversité de techniques d'analyse visant à relever les défis associés à la vérification de la sûreté mémoire.

En dépit de l'important corpus de travaux dans ce domaine, il demeure un champ de recherche particulièrement actif, les approches existantes présentant encore des limitations majeures, telles qu'une détection incomplète, l'absence de garanties formelles, une surcharge d'analyse

élevée ou une portabilité restreinte. Pour faciliter un examen systématique de ces techniques, nous les regroupons en deux grandes catégories : les outils d'analyse statique et les outils d'analyse dynamique. Nous examinons ces deux types d'analyse dans des sous-sections dédiées. Chaque sous-section discute des avantages et des inconvénients propres à sa catégorie, propose un aperçu de certains outils représentatifs, et met en lumière les études expérimentales présentes dans la littérature comparant les solutions existantes.

2.7.1 Analyse statique des violations de la sûreté de la mémoire

Les approches d'analyse statique visant à vérifier le respect des contraintes de sûreté mémoire examinent les accès mémoire dans le code source ou dans ses représentations intermédiaires, sans exécuter le programme. L'avantage principal de ces approches par rapport aux techniques dynamiques réside dans leur capacité à couvrir l'ensemble du code, grâce à une analyse complète du programme dans sa globalité. Cependant, cette force potentielle est compromise par des limitations fondamentales pouvant conduire à des analyses imprécises. Ces limitations proviennent des difficultés à analyser avec précision certaines propriétés complexes des programmes, telles que l'aliasing des pointeurs, les dépendances inter-procédurales ou encore les structures de contrôle complexes comme les boucles, les appels indirects et les branches conditionnelles. De telles propriétés dépendent souvent d'informations disponibles uniquement à l'exécution, ce qui rend leur analyse précise irréalisable dans un contexte purement statique.

En outre, l'analyse approfondie de bases de code même modérément volumineuses est souvent inabordable sur le plan computationnel, ce qui introduit de graves problèmes de passage à l'échelle. Pour contourner ces contraintes, les outils statiques reposent généralement sur des analyses incomplètes, s'appuyant sur des approximations telles qu'une analyse simplifiée des pointeurs, une analyse insensible aux chemins d'exécution ou encore une analyse strictement intra-procédurale ou partiellement inter-procédurale [63, 64]. De telles approximations introduisent inévitablement un certain degré d'incertitude dans l'analyse. Pour gérer cette incertitude, les outils statiques combinent souvent des stratégies conservatrices, susceptibles de surestimer les problèmes potentiels et ainsi produire des faux positifs, avec des techniques minimisant cette incertitude, ce qui peut conduire à des faux négatifs [65, 66]. Trouver un compromis optimal entre la réduction des faux positifs et celle des faux négatifs constitue ainsi un défi central dans l'analyse statique des programmes.

Pereira et al. [67] mènent une analyse expérimentale afin d'évaluer la capacité des outils d'analyse statique les plus utilisés, ainsi que celle de certaines caractéristiques statiques du code, à détecter les vulnérabilités de type débordement de tampon dans des projets C/C++

à grande échelle. Dans cette étude, les auteurs utilisent le noyau Linux, Mozilla et Xen comme trois projets logiciels représentatifs, caractérisés par une base de code importante et un nombre significatif de débordements de tampon identifiés au cours de leur développement. Les auteurs évaluent l'efficacité de CppCheck et Flawfinder, deux outils d'analyse statique C/C++, couramment utilisés dans la détection des débordements de tampon dans ces projets. L'équipe analyse en particulier les différences entre les alertes générées par ces outils avant et après la correction des erreurs de débordement de tampon, en notant la disparition, la persistance ou l'apparition de nouvelles alertes dans le code corrigé. Leur analyse révèle que, dans la majorité des cas, les outils produisent les mêmes alertes aussi bien pour le code vulnérable que pour sa version corrigée, ce qui témoigne de leur capacité limitée à détecter ce type de vulnérabilités et souligne la nécessité de définir de nouvelles règles spécifiques dans ces outils pour cibler ces erreurs. L'étude examine ensuite l'éventuelle corrélation entre certaines métriques statiques du code fréquemment utilisées et la présence de débordements de tampon. En adoptant une démarche méthodologique similaire à l'analyse précédente, les auteurs comparent les variations de ces métriques entre les versions fautives et corrigées des unités de code étudiées. Leurs résultats indiquent qu'aucune relation causale n'existe entre les corrections des débordements de tampon et les variations observées dans les métriques analysées, suggérant que ces métriques, prises isolément, sont insuffisantes pour détecter les vulnérabilités de type débordement de tampon.

Shiraishi et al. [68] présentent la Toyota ITC test suite, un ensemble de tests unitaires accessible publiquement, spécifiquement conçu pour évaluer l'efficacité des outils d'analyse statique dans la détection des erreurs courantes en programmation C et C++, y compris les violations des bornes de tampon. Les auteurs procèdent également à une évaluation comparative de trois outils d'analyse statique—CodeSonar, Code Prover et Bug Finder—à l'aide de cette suite de tests. Les résultats indiquent que ces outils atteignent des taux de détection quasi complets tout en maintenant de faibles taux de faux positifs dans l'identification des défauts mémoire statiques et dynamiques, catégories auxquelles appartiennent les débordements et les sous-dépassements de tampon. Cependant, il convient de souligner que les trois outils évalués sont des logiciels commerciaux. Arusoaie et al. [69] évaluent quant à eux dix outils d'analyse statique non commerciaux, en utilisant une version légèrement modifiée du banc d'essai Toyota ITC, et révèlent des performances nettement insuffisantes. Les outils atteignent un taux de détection moyen de 12.8% pour les défauts mémoire dynamiques et de 30.9% pour les défauts mémoire statiques, avec des taux de faux positifs correspondants de 8.4% et 7.0% respectivement. Parmi les outils testés, Frama-C [70] s'est montré le plus efficace pour détecter les défauts mémoire dynamiques, avec un taux de détection de 83%, mais au prix d'un taux de faux positifs élevé de 23%. Pour les défauts mémoire statiques,

l'analyseur statique de Clang a obtenu les meilleurs résultats avec un taux de détection de 82%, accompagné toutefois d'un taux de faux positifs non négligeable de 15%.

Certains outils d'analyse statique sont spécifiquement conçus pour examiner les binaires afin de détecter des violations des accès mémoire, notamment dans les situations où le code source n'est pas disponible ou lorsque l'analyse au niveau binaire est essentielle, par exemple pour révéler des vulnérabilités introduites par le compilateur [71, 72]. À titre d'exemple, UAFDetector [63] est conçu pour détecter les violations de type use-after-free dans le code binaire. Son analyse repose principalement sur la construction du graphe de flot de contrôle (CFG), obtenu grâce à une approche hybride combinant l'analyse statique — utilisée pour extraire le flot de contrôle de base — et l'instrumentation dynamique pilotée par le fuzzing, permettant de capturer les sauts indirects. Pour assurer la scalabilité, qui constitue un objectif central de conception, l'outil résume les comportements critiques des fonctions — tels que la création, l'utilisation et la destruction de pointeurs suspendus — et remplace les appels de fonction par ces résumés, afin de réduire les calculs redondants lors de l'analyse inter-procédurale. Toutefois, l'effort visant à assurer la scalabilité implique certains compromis sur la précision de la détection. L'analyse insensible aux chemins d'exécution est la principale cause de faux positifs, tandis que l'analyse incomplète de l'aliasing des pointeurs et le traitement imprécis des boucles figurent parmi les principales sources de faux négatifs.

Autre exemple, cwe_checker [64] est un cadre open source qui emploie un ensemble varié de techniques d'analyse statique pour identifier les types de vulnérabilités CWE courants dans les binaires, avec un accent particulier sur les fichiers ELF. Il combine notamment des analyses points-to et value-set pour détecter les débordements de tampon (CWE-119). Cependant, cette approche présente des limitations notables [73]. L'imprécision de l'analyse de pointeurs utilisée est une source bien connue de faux positifs. En outre, l'outil est incapable d'inférer la taille des objets alloués dans toute fonction appelée autre qu'un allocateur standard, en raison de la portée limitée de son analyse inter-procédurale. Cette limitation peut entraîner des faux négatifs, puisque les débordements liés à ces objets ne sont pas détectés. De plus, l'outil ne peut pas détecter les débordements affectant la mémoire globale, et dans le cas de la mémoire de pile, il ne peut identifier que les débordements dépassant les limites de la trame de pile (stack frame).

Plusieurs études récentes ont exploré l'utilisation de l'apprentissage automatique dans l'analyse statique de code en vue de détecter des défauts, la majorité d'entre elles se concentrant sur l'identification d'erreurs sémantiques et de vulnérabilités, notamment celles liées aux opérations d'accès mémoire [74]. Nombre de ces travaux concluent que les techniques d'apprentissage automatique actuellement disponibles ne parviennent pas à produire des résultats

satisfaisants dans ce domaine.

Par exemple, dans une série d'études empiriques, Chappell et al. [75] cherchent à concevoir des modèles d'apprentissage automatique indépendants capables de détecter efficacement les erreurs de codage en C, principalement celles liées aux accès mémoire. Ils constatent que l'efficacité des modèles demeure très faible, sauf lorsqu'ils sont entraînés à partir de caractéristiques extraites via des outils d'analyse statique, ce qui contredit leur objectif initial d'indépendance vis-à-vis de ces techniques. De plus, les auteurs rapportent que, même avec de telles caractéristiques, les modèles entraînés ne parviennent pas à se généraliser à d'autres ensembles de données.

Dans une étude similaire, Zhao et al. [76] examinent le potentiel des modèles d'apprentissage automatique pour remplacer les outils d'analyse statique manuellement développés dans la détection d'erreurs de programmation courantes, en se concentrant sur les débordements de tampon dans le code C. Ils expérimentent des stratégies d'apprentissage supervisé et non supervisé pour extraire des caractéristiques pertinentes à partir du code source ou de sa représentation intermédiaire LLVM, en les combinant avec certains classificateurs existants. Leurs résultats montrent que les modèles ainsi entraînés obtiennent des performances très médiocres lorsqu'ils sont appliqués à des bases de code réelles et inédites, mettant en évidence les grandes difficultés rencontrées par ces modèles pour apprendre à détecter les débordements de tampon dans les programmes C.

Adoptant une approche différente, Krishnan et al. [77] étudient l'intégration de techniques d'apprentissage automatique dans des cadres d'analyse statique afin de faciliter la détection de vulnérabilités de sécurité. Leurs expériences, incluant la détection de bogues liés aux accès mémoire dans les programmes C, révèlent que l'application de ces approches à l'analyse statique est en grande partie inefficace. Les auteurs identifient quatre principales limitations liées à l'utilisation de l'apprentissage automatique : la difficulté à générer un volume suffisant de données étiquetées pour l'entraînement, l'incapacité des modèles à comprendre pleinement le comportement dynamique du code, les évaluations trompeuses de performance dues à l'utilisation de jeux de données non représentatifs, et le manque d'explicabilité des résultats produits par ces modèles comparativement aux techniques traditionnelles d'analyse de programmes.

Bien que les outils d'analyse statique présentent l'avantage d'offrir une couverture complète du code sans nécessiter l'exécution du programme, leur efficacité est limitée par des contraintes inhérentes de précision et de passage à l'échelle. Ces défis obligent les approches statiques à recourir à des approximations qui impliquent souvent un compromis entre faux positifs et faux négatifs, restreignant ainsi leur fiabilité pratique pour la détection des viola-

tions de la sécurité mémoire.

2.7.2 Analyse dynamique des violations de la sûreté de la mémoire

Les techniques dynamiques de vérification de la sécurité des accès mémoire fonctionnent en capturant et en analysant les opérations liées à la mémoire qui se produisent lors d'une exécution donnée du programme. Dans ce contexte, un outil dynamique s'appuie sur des informations collectées à l'exécution pour identifier les violations de la sécurité des accès mémoire le long du chemin d'exécution observé. Plus précisément, les informations essentielles à surveiller à l'exécution sont les points d'allocation, de désallocation et d'accès à la mémoire. De nombreux détecteurs dynamiques s'appuient également sur certaines formes d'analyse statique. Cette dépendance provient généralement de la nécessité de réaliser, à la compilation, des analyses et des transformations du code afin de permettre et d'optimiser la surveillance à l'exécution [61, 62]. Une couverture de code limitée constitue une contrainte inhérente aux approches de détection dynamique, ce qui les rend sujettes aux faux négatifs [78]. Cette limitation découle du fait que, lors de chaque exécution, les outils dynamiques n'analysent que le chemin d'exécution observé, qui n'est qu'un parmi un nombre potentiellement élevé de chemins possibles dans le programme. Cela souligne l'importance de recourir à un ensemble diversifié de bancs de test lors de l'utilisation d'outils de vérification dynamique, puisque seuls les chemins de code exercés par ces bancs sont analysés. Malgré ces limitations, les techniques d'analyse dynamique sont plus largement adoptées dans les applications réelles que les approches statiques, principalement en raison de leur précision de détection supérieure et de leur meilleure capacité de passage à l'échelle [79].

Azhari et al. [80] proposent un cadre post-mortem à faible surcharge pour la détection des fuites de mémoire et l'identification de leurs causes profondes, en retraçant les problèmes jusqu'à la pile d'appels de l'allocation. La méthode présentée recueille périodiquement des statistiques d'exécution sur les blocs mémoire alloués et désalloués en interceptant les appels aux fonctions d'allocation et de désallocation sur le tas. Les données collectées sont ensuite analysées à l'aide d'un algorithme de détection des fuites fondé sur une analyse de croissance, conçu pour minimiser la probabilité de faux positifs.

Kroes et al. [62] présentent Delta Pointers, un mécanisme de protection contre les débordements de tampon fondé sur le marquage des pointeurs. Cette approche encode deux métadonnées dans les bits de poids fort des pointeurs 64 bits : la distance entre le pointeur courant et la borne supérieure de l'objet mémoire correspondant, appelée delta tag, et un indicateur binaire de débordement stocké dans le bit le plus significatif. Le cadre repose sur une instrumentation à la compilation pour gérer l'encodage du delta tag dans les pointeurs originaux

et le propager au travers des opérations arithmétiques sur pointeurs. Plus précisément, le module d'instrumentation identifie les pointeurs retournés par les sites d'allocation mémoire et leur applique l'encodage du delta tag. Lors d'une opération arithmétique sur pointeur, le delta tag est ajusté afin de refléter l'opération, garantissant qu'il conserve la distance actualisée à la borne supérieure. L'indicateur de débordement, initialement positionné à zéro, est implicitement mis à un lorsqu'une opération arithmétique entraîne un dépassement de la borne supérieure ; cela se produit lorsque le delta tag déborde dans le bit le plus significatif à la suite de l'opération. Lors de la déréférence du pointeur, le delta tag est masqué tandis que le bit de débordement reste inchangé. Si ce bit est activé, le pointeur déréférencé devient non canonique, provoquant la génération d'une erreur par l'unité de gestion mémoire. Delta Pointers ne permet pas de détecter les débordements vers le bas (buffer underflows) ni aucun type de violations temporelles.

AddressSanitizer (ASan) [61] est un outil dynamique open source conçu pour détecter un ensemble de violations spatiales et temporelles courantes dans les programmes C et C++ lors de l'accès à des objets situés sur le tas, la pile et en mémoire globale. Il est intégré à LLVM depuis la version 3.1 et à GCC depuis la version 4.8. Plus précisément, il s'agit d'un outil de vérification à l'exécution composé d'un module d'instrumentation du code à la compilation et d'une bibliothèque d'exécution. L'outil combine l'utilisation d'une mémoire fantôme (shadow memory) et de zones tampons (red zones) pour détecter les violations d'accès spatial, chaque octet de la mémoire fantôme indiquant quels octets de la mémoire originale de l'application sont adressables. Au moment de la rédaction, sa capacité de détection des fuites de mémoire demeure « expérimentale » selon sa documentation, avec un problème majeur non résolu, lié en particulier à des faux négatifs [81].

CETS [78] est une approche fondée sur le compilateur visant à assurer la sûreté temporelle dans les programmes C. À l'exécution, chaque pointeur est associé à deux métadonnées : une clé unique et une adresse de verrou. Un pointeur est considéré comme valide tant que la clé et le verrou associé, stocké à l'adresse de verrou, sont identiques. Pour la recherche dans l'espace fantôme (shadow space), l'outil utilise une structure de données en trie à deux niveaux afin d'associer chaque pointeur à sa clé et à son adresse de verrou correspondantes. Le traitement des pointeurs dérivés repose sur l'hypothèse que le pointeur dérivé fait référence à la même allocation mémoire que le pointeur original (hypothèse de sûreté spatiale). Lorsque le compilateur rencontre une opération arithmétique sur pointeur, il insère du code pour propager les métadonnées du pointeur original vers le pointeur dérivé. CETS s'appuie sur une analyse à la compilation pour suivre la propagation des pointeurs. En termes de portée de détection, la méthode proposée se limite à identifier uniquement les accès dépassant la limite supérieure prévue d'un tampon.

ViK [79] est une protection à l'exécution, fondée sur les pointeurs, contre les violations de la sûreté temporelle de la mémoire. Elle attribue un identifiant d'objet aléatoire (Object ID) à chaque objet alloué, l'insérant dans les bits inutilisés du pointeur et vérifiant sa validité lors de la déréférenciation ou de la désallocation. Pour optimiser les performances, elle exploite une analyse statique afin d'omettre les vérifications sur les pointeurs garantis sûrs vis-à-vis des accès après libération (use-after-free) et utilise des techniques assistées par matériel pour réduire la surcharge. La mise en œuvre purement logicielle de l'outil présente, selon les rapports, une surcharge d'exécution d'environ 20 %. ViK est incapable de détecter toute forme de violation de la sûreté spatiale de la mémoire.

En résumé, les outils existants de vérification dynamique présentent généralement soit un champ de détection très limité, soit une surcharge d'exécution importante. Le premier cas restreint leur capacité à détecter de manière exhaustive les violations de la sûreté de la mémoire, tandis que le second les rend peu pratiques à déployer dans des environnements de test disposant de ressources limitées.

2.8 Intel Processor Trace (Intel PT)

Intel Processor Trace (Intel PT) est une fonctionnalité matérielle qui collecte des informations sur l'exécution logicielle avec un impact limité sur le programme analysé [18]. Elle enregistre ces informations sous forme de paquets de données fortement compressés. Intel PT est capable de tracer le flot de contrôle en générant différents types de paquets. Un outil de post-traitement combine ensuite ces paquets avec les binaires du programme analysé pour reconstruire précisément son exécution. Les données de trace ne sont générées que pour les changements de flot de contrôle non connus statiquement. Intel PT génère les quatre types de paquets suivants pour encoder le flot de contrôle [18] :

- Paquets Taken-Not-Taken (TNT) : Ils suivent la direction des branches conditionnelles directes (si elles ont été prises ou non). Chaque point de données TNT est une indication sur un seul bit. Un paquet TNT peut contenir jusqu'à six de ces points. Si un autre paquet doit être envoyé avant que les six points aient été enregistrés, Intel PT envoie un paquet TNT partiel, c'est-à-dire contenant moins de six points.
- Paquets Target IP (TIP) : Ces paquets enregistrent l'adresse cible (IP – Instruction Pointer) des branches indirectes, interruptions et exceptions.
- Paquets Flow Update (FUP) : Ils fournissent l'adresse source de l'instruction lorsque celle-ci ne peut pas être déterminée à partir du binaire, notamment dans le cas d'événements asynchrones.
- Paquets MODE : Ils fournissent au décodeur de trace des informations importantes sur

l'état d'exécution du processeur afin qu'il puisse interpréter correctement les binaires désassemblés et les données de trace.

La Figure 2.4 illustre comment Intel PT encode le flot de contrôle à l'exécution. Dans cet exemple, le flot de contrôle change de manière non déterministe à deux endroits : une branche conditionnelle directe (ligne verte) et une branche indirecte (ligne orange). Intel PT génère un paquet TNT d'un bit pour encoder la branche conditionnelle directe (afin d'indiquer si elle a été prise ou non), et un paquet TIP pour encoder la branche indirecte (afin d'indiquer l'adresse cible IP).

La Figure 2.5 montre comment un décodeur reconstruit le flot de contrôle exact en combinant les paquets Intel PT avec les binaires du programme lors de l'analyse. Le décodeur interprète que la branche conditionnelle directe (ligne verte) a été prise selon le paquet TNT correspondant. Il extrait également l'adresse IP cible de la branche indirecte (ligne orange) à partir du paquet TIP correspondant.

En plus du flot de contrôle, Intel PT peut également capturer d'autres informations, notamment des données de synchronisation. Il peut enregistrer à la fois le nombre de cycles et des horodatages. Intel PT a été introduit pour la première fois dans les microarchitectures Broadwell et Goldmont [18]. Intel PT est le successeur des technologies de traçage matérielles Last Branch Record (LBR) et Branch Trace Store (BTS) [82] [83] [84]. Intel PT impose une surcharge bien moindre que LBR et BTS. Pour un ensemble de benchmarks, une étude présentée dans [85] montre que la surcharge moyenne à l'exécution liée à l'utilisation d'Intel PT se situe entre 2 et 5%, bien qu'en général elle dépende du programme et de l'architecture. En particulier, il a été observé qu'une génération excessive de paquets TIP peut augmenter considérablement la surcharge, pouvant atteindre 20%.

Bien que les paquets Intel PT soient fortement compressés, la quantité de données de trace générée peut devenir très importante [86], ce qui rend le décodage et l'analyse ultérieure complexes pour extraire les informations pertinentes.

2.9 Résumé

En résumé, ce chapitre a présenté un état de l'art sur la détection des conditions de course et sur la sûreté des accès mémoire. Nous avons examiné la définition formelle des conditions de course, les difficultés inhérentes à leur détection, les principales catégories d'approches automatisées de détection, leurs stratégies algorithmiques sous-jacentes ainsi que les détecteurs les plus connus pour les programmes C et C++ multithreadés. Nous avons ensuite introduit le concept de sûreté des accès mémoire, discuté des défis spécifiques liés à la sûreté des accès

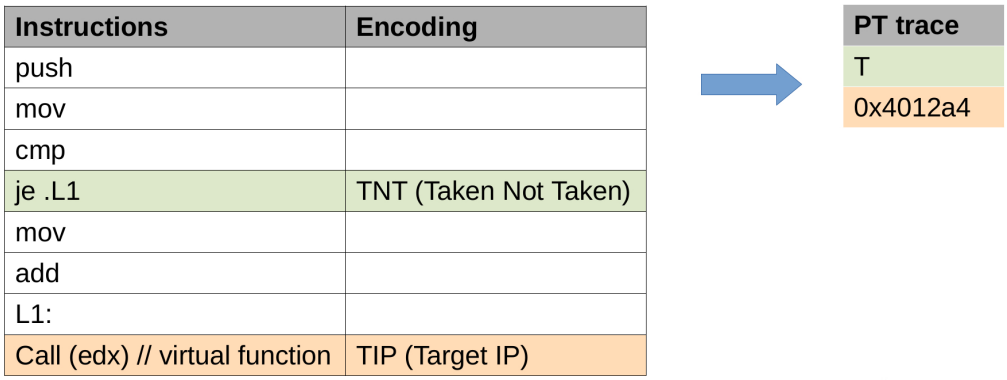

FIGURE 2.4 Exemple de l'encodage du flot de contrôle avec Intel PT [1]

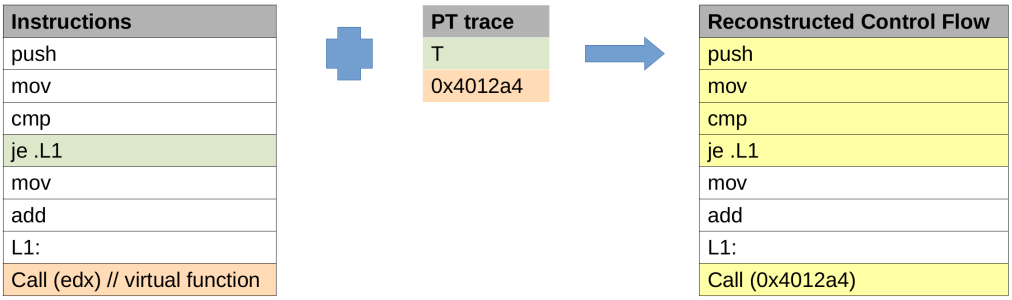

FIGURE 2.5 Exemple de décodage des paquets de trace Intel PT [1]

à la mémoire de tas et passé en revue les principales techniques de détection des violations d'accès mémoire dans les programmes C. Enfin, la fonctionnalité matérielle Intel Processor Trace (Intel PT) a été présentée, puisqu'elle est utilisée dans l'un des outils proposés dans cette thèse.

En dépit de leur précision de détection et de leur capacité de passage à l'échelle supérieure, les outils existants de vérification dynamique pour la détection des conditions de course et des violations de la sûreté de la mémoire liées au tas imposent un surcoût important en termes de temps d'exécution et de consommation mémoire. De tels coûts restreignent leur applicabilité dans des environnements à ressources limitées, y compris les systèmes embarqués, et représentent par ailleurs un défi majeur lors de tests réalisés dans des conditions de charge de travail réalistes. En outre, même les détecteurs les plus largement adoptés présentent d'importantes lacunes en matière de détection. Ainsi, la conception de nouveaux détecteurs dynamiques à faible surcoût et l'élargissement de la couverture des outils existants sans accroître leur surcoût constituent deux axes de recherche importants, tous deux abordés dans cette thèse.

CHAPITRE 3 METHODOLOGIE

Ce chapitre présente la méthodologie employée pour atteindre les objectifs de recherche dans chacun des trois axes de cette thèse. Il est structuré en trois sections, chacune consacrée à un axe de recherche, exposant ses objectifs spécifiques ainsi que les procédures mises en œuvre pour les réaliser.

3.1 Premier axe de recherche

Dans le premier axe de recherche de cette thèse, nous avons conçu et mis en œuvre Thread-Monitor (TMon), un détecteur postmortem de conditions de course à faible surcoût pour les programmes C et C++ multithread utilisant la bibliothèque Pthread. L'objectif central guidant la conception de TMon est d'offrir la même analyse exhaustive de détection de conditions de course que ThreadSanitizer (TSan), tout en induisant un surcoût d'exécution nettement inférieur. Cette orientation vise à garantir la capacité d'utilisation de TMon dans des environnements de test réels soumis à des contraintes de ressources.

Afin d'éviter le surcoût lié à l'analyse à la volée, nous avons développé TMon comme un outil postmortem comportant deux phases : la phase de traçage et la phase d'analyse postmortem. Contrairement à TSan, qui analyse l'exécution du programme pendant son déroulement, TMon trace les événements du programme jugés pertinents et en reporte l'analyse à l'issue de l'exécution. Outre la réduction du surcoût d'exécution, un autre avantage de l'analyse fondée sur le traçage est qu'elle peut être appliquée aux exécutions en environnement simulé. De plus, la détection de conditions de course à la volée introduit des perturbations temporelles plus importantes le long des chemins de contrôle que l'analyse postmortem fondée sur le traçage. TMon trace principalement deux types d'événements : les accès mémoire et les contraintes temporelles entre threads. Plus précisément, il trace exactement les mêmes événements que ceux surveillés par TSan et s'assure, pour chacun d'eux, d'enregistrer strictement les mêmes informations d'exécution que TSan utilise à des fins d'analyse. Cela garantit qu'au moment de l'analyse postmortem des données de trace, TMon peut effectuer sans difficulté la même analyse exhaustive de détection de conditions de course que TSan.

Pour maintenir un surcoût d'exécution minimal, il est impératif d'adopter une approche de traçage efficace. Les méthodes classiques de traçage assisté par logiciel sont souvent intrusives et entraînent un surcoût important. Conscients de cette limitation, nous avons choisi de nous appuyer sur le traçage assisté par matériel, une alternative prometteuse grâce à son

impact limité sur le programme tracé. Le traçage assisté par matériel est intégré à diverses architectures, notamment chez Intel, où il est connu sous le nom d'Intel Processor Trace (Intel PT). Ce mécanisme repose sur des blocs matériels dédiés pour collecter efficacement des informations sur l'exécution du programme sous forme de différents paquets de trace, garantissant ainsi une approche de traçage à très faible surcoût. Toutefois, le traçage assisté par matériel se concentre principalement sur le traçage détaillé des instructions et ne propose pas de fonctionnalité spécifique pour tracer les accès mémoire, ce qui constitue pourtant un besoin essentiel dans notre cas. Néanmoins, le paquet `ptwrite` d'Intel, fonctionnalité récemment devenue largement disponible dans Intel PT, constitue un compromis pertinent pour notre usage. Un paquet `ptwrite` est une charge utile de 64 bits générée par l'utilisateur, produite lors de l'exécution de l'instruction `ptwrite`. Cette instruction transmet la valeur de l'opérande 64 bits qui lui est passée au matériel Intel PT, où elle est encodée dans un paquet `ptwrite`. La génération de paquets `ptwrite` respecte strictement l'ordre d'exécution des instructions `ptwrite` correspondantes. Introduite initialement dans les processeurs Atom, l'instruction `ptwrite` est désormais largement disponible sur la 12^{e} génération de processeurs Intel Core. Son exécution est remarquablement rapide, ne nécessitant que deux cycles CPU sur notre machine équipée d'un processeur Intel Core i7-12700 de 12^{e} génération.

3.2 Deuxième axe de recherche

Dans le deuxième axe de recherche, nous avons développé AddressMonitor (AMon), un outil de vérification à l'exécution à faible surcoût conçu pour détecter les violations de la sûreté spatiale et temporelle de la mémoire de tas dans les programmes C. Plus précisément, AMon détecte les accès hors limites (out-of-bound accesses), les utilisations après libération (use-after-frees), les doubles libérations (double-frees) et les fuites de mémoire (memory leaks). Il est également capable d'identifier deux autres pratiques de programmation dangereuses liées à l'accès mémoire sur le tas : le passage d'un pointeur non de base aux fonctions standards de réallocation ou de libération de mémoire, ainsi que les situations potentiellement à risque d'utilisation après libération résultant d'une réallocation mémoire. AMon est spécifiquement conçu pour offrir des capacités de détection exhaustives tout en maintenant un faible surcoût, garantissant ainsi son aptitude au déploiement dans des environnements contraints en ressources tels que les systèmes embarqués.

AMon repose sur une combinaison de marquage des pointeurs (pointer tainting) et d'analyses et transformations effectuées à la compilation. Lors de la compilation, il détecte les opérandes de pointeurs sur le tas dans les opérations d'accès mémoire, crée une variante non marquée (untainted) pour chaque pointeur sur le tas, puis remplace les opérandes originaux par leurs

équivalents non marqués. À l'exécution, il assigne un identifiant de marquage unique à chaque objet alloué sur le tas, insère cet identifiant dans les bits inutilisés du pointeur retourné (pointeur marqué) et maintient une table des objets pour suivre, tout au long de l'exécution du programme, les informations spatiales, temporelles et de débogage relatives aux objets du tas alloués par l'utilisateur.

AMon se compose de deux modules principaux : un module d'analyse et de transformation à la compilation, et une bibliothèque d'exécution. La transformation réalisée à la compilation illustre comment une analyse statique peut détecter les pointeurs dérivés du tas et imposer des schémas d'accès mémoire sûrs sans modifier la sémantique du programme. En combinant une analyse des flux de valeurs pour identifier les pointeurs dérivés, un masquage de bits (bitmasking) pour garantir un déréférencement sûr, et une instrumentation sélective pour la validation à l'exécution, le passage traite à la fois les erreurs de mémoire spatiales et temporelles. La bibliothèque d'exécution d'AMon prend en charge la surveillance et l'analyse dynamiques en construisant et gérant la table des objets, en interceptant les fonctions standards C d'allocation et de libération de mémoire, et en définissant les mécanismes d'analyse utilisés par AMon.

3.3 Troisième axe de recherche

Dans le troisième axe de recherche de cette thèse, nous présentons une analyse détaillée, au niveau de l'implémentation, de ThreadSanitizer v3, la version la plus récente de ce détecteur de conditions de course largement utilisé et intégré à la fois dans LLVM et GCC, et nous mettons en évidence deux angles morts de détection, jusqu'ici non documentés, qui découlent de son implémentation.

Le premier angle mort concerne l'éviction des valeurs d'ombre (shadow values) lorsque toutes les cases sont occupées. Nous montrons que la politique de remplacement aléatoire par défaut appliquée par ThreadSanitizer peut conduire à l'écrasement prématuré d'informations critiques relatives aux courses. Pour y remédier, nous proposons une stratégie d'écrasement sensible à la taille (size-aware overwrite), qui privilégie l'éviction des entrées correspondant à l'accès impliquant le plus petit nombre d'octets. Notre évaluation sur les bancs d'essai PARSEC montre une amélioration mesurable des indicateurs de détection.

Le second angle mort résulte de la non-atomicité de la séquence complète de vérification et de mise à jour (check-and-update). Bien que les lectures et écritures individuelles des valeurs d'ombre soient atomiques, l'entrelacement pendant l'opération composite peut conduire à manquer certains rapports de courses. Nous formalisons cette condition. Dans l'ensemble, nos

résultats soulignent que garantir la justesse des détecteurs de conditions de course requiert une évaluation attentive à la fois des aspects algorithmiques et des détails d'implémentation.

3.4 Résumé

En résumé, ce chapitre a exposé la méthodologie adoptée pour atteindre les objectifs de recherche de cette thèse. Il a présenté, pour chacun des trois axes étudiés, leurs objectifs spécifiques ainsi que les procédures mises en œuvre pour les atteindre. Les trois chapitres suivants détailleront successivement chacun de ces axes de recherche.

CHAPITRE 4 ARTICLE 1 : THREADMONITOR : LOW-OVERHEAD DATA RACE DETECTION USING INTEL PROCESSOR TRACE

Auteurs : Farzam Dorostkar, Michel Dagenais, Ankush Tyagi et Vince Bridgers.

Soumis à Concurrency and Computation : Practice and Experience

Date : 23 Avril 2024

4.1 Abstract

Data races are among the most difficult multithreading bugs to find, due to their non-deterministic nature. This and the increasing popularity of multithreaded programming have led to the need for practical automated data race detection. In this context, dynamic data race detectors have received more attention, compared to static tools, owing to their higher accuracy and scalability. Yet, state-of-the-art dynamic data race detectors cannot be used in many real-world testing scenarios, since they cause significant slowdown and memory overhead. Notably, ThreadSanitizer (TSan), the default dynamic data race detector in both clang and gcc compilers, is reported to typically impose a 5×-15× slowdown and a 5×-10× memory overhead, which is not tolerable in many industrial use cases. To address this issue, this paper introduces ThreadMonitor (TMon), a low-overhead postmortem data race detector for multithreaded C/C++ programs that use the Pthread library. At runtime, TMon traces the information required for detecting occurrences of data races (i.e. shared memory accesses and timing constraints among threads) using Intel Processor Trace (Intel PT), a non-intrusive hardware feature dedicated to tracing software execution. Thereafter, its postmortem analyzer examines the collected trace data to determine whether the traced program execution exhibited data races, performing a verification similar to that carried out by TSan at runtime. Introducing algorithmic improvements in its postmortem analyzer, TMon can further achieve a higher data race detection coverage compared to TSan. TMon has no direct data memory overhead, incurs minimal instruction memory overhead, and causes a very small slowdown, making it an ideal choice in test environments with limited resources.

4.2 Introduction

In this section, we first define a data race and discuss the inherent challenges associated with its detection. Following that, we review distinct categories of automated data race detection tools and explore different algorithmic strategies that drive their functionality. We

conclude this section by introducing the main contributions of this research and explaining the organization of the paper.

4.2.1 Data race

In multithreaded programs, threads typically share access to a portion or the entirety of the application memory. While shared memory enables efficient thread communication, it also exposes a multithreaded program to *data races*. A data race arises when two or more threads access the same memory location, without synchronization or mutual exclusion, with at least one of the accesses being a write operation. When a data race occurs, the runtime behaviour of a program can be influenced by the temporal order of the accesses constituting the race [87]. A data race is considered a concurrency error, unless the resulting non-determinism is a design choice.

Throughout this paper, we distinguish between synchronization and mutual exclusion as distinct types of *timing constraints* [88], avoiding the misrepresentation found in some previous studies where these two concepts are erroneously considered either similar or a subset of one another. We use this terminology to collectively refer to both concepts.

Locating data races can be extremely challenging for programmers, due to two main reasons [89]. The first challenge is to ensure that each access to shared variables adheres to proper timing constraints. Performing this verification process can be quite complicated, even for a slightly complex project. The second challenge is that a data race may only lead to the corruption of shared data under particular thread interleavings. Therefore, it is possible for a programmer to miss a data race even with comprehensive testing. Furthermore, a corrupted shared variable may not result in immediate failure, but rather propagate through the program and eventually manifest as unexpected behaviour or errors in other parts of the program. Detecting the root cause of such anomalies can be extremely hard.

4.2.2 Automated data race detection

The above mentioned challenges have prompted considerable interest in the research and development of automated tools for detecting data races. In general, data race detectors can be categorized into two main classes : static tools and dynamic tools. Some tools employ a hybrid approach to enhance performance and precision.

Static tools analyze the source code or intermediate representations of a program, aiming to precisely detect all potential interleavings of shared memory accesses that can lead to data races. Static approaches encounter inherent limitations in accurately analyzing language

constructs, that involve dynamic behaviour or complex interactions, including pointers, dynamic memory allocation, dynamic thread creation, concurrency mechanisms, and complex control flow structures. A common challenge is the accuracy of the memory model used. For instance, the clang static analyzer drives analysis on the Abstract Syntax Tree (AST), and uses an analytical memory approximation that suffers from inaccurate aliasing analysis. Another serious consideration is efficiency. While the main advantage of static analysis, as compared to dynamic analysis, lies in its capacity to provide maximum code coverage by performing a comprehensive examination of the entire program, careful analysis of every potential execution path is computationally unfeasible [2, 3]. These constraints inevitably undermine the accuracy and usability of static tools. To achieve soundness, static race detectors typically employ a certain level of conservatism in their analysis techniques, which may in turn raise an overwhelming number of false alarms [4–7]. Therefore, striking an optimal balance between the occurrence of false negatives and false positives is a fundamental hurdle in static race detection. To address this issue, some tools rely on programmer annotations, as explicit hints for guiding the tool towards more accurate reports [4, 8–10]. Applying such less automatic tools to relatively large code bases is impractical [11]. Some other tools apply heuristic algorithms to filter out false alarms from the generated warnings. However, there will be a chance of erroneously discarding legitimate data race warnings [12, 13].

Dynamic tools utilize tracing mechanisms to monitor the events of interest, that occur during a particular program execution. Their primary objective is to examine whether the observed execution path exhibits any apparent data races. Dynamic tools are further classified into two categories : on-the-fly tools and postmortem tools. On-the-fly techniques analyze the stream of program events online, while postmortem approaches record them in a log file, and defer the data race detection analysis until after the program execution has finished. The key runtime information to monitor are accesses to shared memory, dynamic memory allocation and deallocation, invocation of threading primitives that impose timing constraints among threads, and the call stack of each thread. To capture them, dynamic approaches commonly rely on invasive program instrumentation, leading to significant slowdown and memory overhead, to the point where even state-of-the-art dynamic data race detectors cannot be used in many real-world testing scenarios. To mitigate this issue, some dynamic approaches employ memory access sampling [14–16]. Given the higher frequency of memory accesses compared to other events of interest, analyzing only a subset of them can help reduce the runtime overhead. However, this approach introduces the risk of missing actual data races, as not all memory accesses are examined. Another constraint of dynamic data race detection is limited code coverage. All dynamic data race detectors are inherently prone to false negatives, since each time they only examine one of possibly many execution paths of a program. In

other words, only code paths that are exercised by test harnesses are analyzed. Although to some extent theoretically feasible, mitigating this limitation by exploring different orderings of observed events, to detect potential data races, poses significant challenges in practical implementation [17]. Evaluating all permutations of observed events is computationally unfeasible and may result in false alarms, as not every event ordering is valid. Consequently, a mechanism must be devised to identify valid event orderings through comprehensive program analysis. Despite the above mentioned limitations, dynamic data race detectors have gained more attention than static detectors, due to their higher accuracy and scalability.

4.2.3 Data race detection algorithms

There are three main algorithmic strategies for detecting data races, used by both dynamic and static tools : the Happens-before algorithm, the Lockset algorithm, and combinations of the two, commonly referred to as hybrid algorithms.

The main idea behind the happens-before approach in data race detection is to determine whether it is possible to establish a timing constraint in the form of a *happened-before* relation between two conflicting memory accesses. The happened-before relation, proposed by Lamport, is a partial order on events happening in a message-passing distributed multiprocess system [90]. The fundamental essence of the happened-before relation lies in establishing a relationship between events, based on their causal dependencies, where an event is said to happen before another event if it can causally affect the later. The notion of happened-before ordering was later transferred to the domain of multithreaded programming, by regarding threads as processes and timing constraints as interprocess messages. The accuracy of happens-before data race detection can be affected by thread interleaving, which is generally non-deterministic. A simple example illustrating this influence on the detection precision can be found in [91], where the happens-before approach generates a false negative.

Lockset-based data race detection relies on the basic locking discipline, assuming that each shared object is consistently protected by at least one lock, throughout the entire program execution [91]. This presumed lock guarantees mutual exclusion. A data race is reported whenever this assumption is violated. The algorithm keeps track of the set of locks consistently protecting each shared object. Upon accessing an object, the algorithm updates the set by taking the intersection with the set of locks held by the thread performing the access. Consequently, if there is no lock consistently guarding the object, the set becomes empty, prompting the algorithm to issue a data race warning. Lockset-based data race detection is effective in capturing violations of mutual exclusion, but it does not handle synchronization violations, which can result in false positives.

Hybrid techniques combine happens-before and lockset-based approaches to data race detection, aiming to achieve higher accuracy compared to relying solely on either method individually [28, 92]. However, their potentially increased complexity can introduce challenges related to performance and resource usage. Striking a balance between improved accuracy and algorithmic complexity is crucial for the practical deployment of hybrid data race detection algorithms.

4.2.4 Contributions and paper organization

The main contributions of this study are as follows.

— As the main contribution of this research, we present ThreadMonitor (TMon), a low-overhead postmortem data race detector designed for multithreaded C/C++ programs that use the Pthread library. TMon has no direct data memory overhead and incurs a minimal slowdown, making it an ideal option in testing systems with constrained resources. TMon is tailored to offer the same level of data race detection capabilities as the latest version of ThreadSanitizer (TSan v3) but with significantly reduced overhead. TMon is publicly accessible as an open-source project, hosted on the following GitHub repository : `https://github.com/farzamdorostkar/tmon`.

— The postmortem analyzer of TMon expands on the data race detection logic employed by TSan v3, potentially enhancing its coverage through the introduction of two novel algorithmic improvements. These enhancements reduce the likelihood of missing data races caused by shadow memory overwriting.

— Parts of this study can potentially serve as a comprehensive technical documentation for TSan v3. Indeed, it details how TMon achieves data race detection capabilities similar to those of TSan v3. Section 4.5 not only presents the implementation of TMon, but also explains key implementation aspects of TSan v3 in parallel, covering both its compile-time instrumentation module and runtime library. To the best of our knowledge, no such documentation exists prior to this study. TSan has gone through major changes in its most recent version, but the associated articles and documentation, that detail the underlying algorithms, have not been updated.

The remainder of this paper is organized as follows. In Section 4.3, we review a range of data race detectors developed to debug C/C++ multithreaded programs. In Section 4.4, we explain the motivation behind developing TMon, its architecture, and our key design choices. Section 4.5 is devoted to providing a comprehensive explanation of the technical aspects of implementing TMon, highlighting the building blocks and mechanisms that enable its functionality. In Section 4.6, we present the performance evaluation of TMon in comparison with TSan v3, on a set of C/C++ benchmarks from the SPEC CPU 2017 suite and a

parallelized implementation of the Fourier transform. Section 4.7 concludes this paper by summarizing contributions, and suggesting potential directions for future work.

4.3 Related Work

In this section, we review various data race detection tools tailored for multithreaded C/C++ programs. We start with a comprehensive introduction to ThreadSanitizer, providing insights into its key components and evolution.

ThreadSanitizer (TSan) is the state-of-the-art open source runtime verification tool, designed to detect data races in C/C++ code. It is the default dynamic data race detector in both clang and gcc compilers. The tool comprises a code instrumentation module and a runtime library. Within user code, memory accesses are instrumented through specific calls to the runtime library, where the race detection logic resides. The runtime library also incorporates interceptors, designed to wrap library functions that are involved in synchronization, mutual exclusion, or memory access operations. TSan has undergone three major versions, each differing in code instrumentation approach and/or analysis behaviour. In what follows, we outline the main ideas behind the first two versions of the tool.

TSan v1 is based on run-time instrumentation. It was initially developed as a PIN [33] tool but subsequently transitioned to a Valgrind variant [35] due to its superior performance, exhibiting a twofold increase in speed. The tool affords users the flexibility to select from two implemented race detection algorithms. The default configuration employs a hybrid state machine that combines the happens-before and lockset algorithms. Alternatively, users may opt for a pure happens-before mode which, although yielding fewer false positives, also exhibits a higher rate of missed real data races. In terms of performance impact, TSan v1 has been reported to incur a 20×-300× slowdown in comparison to native execution.

Part of the significant slowdown exhibited by TSan v1 can be attributed to the utilization of Valgrind for runtime instrumentation. In order to address this issue, TSan v2 [93] was developed, employing compile-time instrumentation instead. This version of the tool instruments user code at the LLVM Intermediate Representation (IR) level. In its initial release, the instrumentation code recorded the effective address of each memory access within a basic block into a thread-local buffer, whose contents were subsequently analyzed upon reaching the end of the respective basic block. This initial release incorporated the same race detection logic as TSan v1, despite the fact that its complex nature also contributed to the slowdown observed in TSan v1. Recognizing this issue, TSan v2 underwent a shift to solely utilizing a pure happens-before mechanism [94], albeit with the potential trade-off of missing a grea-

ter number of real data races. Another paradigm shift was the substitution of thread-local buffers with shadow memory, to maintain a history of memory accesses. Each consecutive eight bytes of application memory were mapped to a set of four *shadow cells*. Each shadow cell, spanning eight bytes, represented a memory access, featuring four components : 16 bits reserved for storing the thread ID of the accessing thread, 42 bits designated for capturing a snapshot of its scalar clock at the time of access, 5 bits utilized for encoding the position of the accessed bytes, and 1 bit denoting whether the access was a write or a read. TSan v2 has been reported to be 1.7×-10× faster than TSan v1.

In [40], Sadowski and Yi interview a group of experienced developers at Google to understand how they use TSan v2 and ThreadSafety [10], a LLVM based static race detector, under different debugging scenarios. Their empirical research reveals that the participants believed TSan v2 was more successful in detecting more difficult to find data races.

The latest version of TSan, TSan v3 [95], adheres to the same compile-time instrumentation approach as TSan v2. However, significant changes have been implemented concerning its shadow memory mechanism. In TSan v3, the size of each shadow cell has been halved compared to TSan v2, while preserving the same functionality. We review TSan v3 in detail in Section 4.5 while elaborating how our proposed tool, TMon, replicates its analysis behavior. Throughout the remainder of this paper, when we refer to TSan without specifying a version, we are specifically addressing TSan v3.

Helgrind [22] is another widely used open source dynamic data race detector. It is a Valgrind tool specialized in detecting threading errors, including data races, in C/C++ and Fortran programs that use the Pthread library. The race detection logic of Helgrind is based on the happens-before relation. The tool monitors all memory accesses, and the threading primitives that enforce timing constraints among threads, including mutex lock and unlock operations, thread creation and joining, condition variables, and semaphores. The collected information is utilized to construct a directed graph that captures the collective happens-before dependencies among program events. When identifying two threads accessing the same memory location, Helgrind determines the possibility of establishing a happens-before relation between the respective events, by examining whether there is a path in the happens-before graph that connects the two accesses. Helgrind incurs a significant runtime overhead, typically resulting in a considerable slowdown of around 100×. In addition to very poor performance, Helgrind may occasionally fail to include the stack of the conflicting memory accesses in its data race reports, even when allocating unbounded storage for conflicting access information. The root cause of this issue remains unknown to the developers of the tool at the time of writing this paper.

Intel Inspector is a dynamic platform to debug memory and threading errors for C/C++ and Fortran programs. Intel Inspector is a successor of the Intel Thread Checker, with added capabilities for memory debugging. It supports detecting possible data races on stack and heap objects. Intel Inspector can be expected to cause slowdowns on the order of 2×-160× [96]. As a result, in many testing scenarios, it may not be possible to run a full production workload.

DILP [44] is a postmortem analysis tool designed to detect data races in Linux device drivers. At runtime, it logs the calling contexts of accesses to global variables, function arguments of pointer type, and variables affected by them. Due to its sole reliance on lockset analysis, the tool is unable to interpret synchronization directives. DILP imposes a considerable runtime overhead, with an average reported slowdown of approximately 7.2×.

LiteRace [14] is a dynamic tool, leveraging sampling-based techniques to achieve low runtime overhead. The study proposes an adaptive sampling algorithm based on the cold-region hypothesis, suggesting that data races are more likely to occur in infrequently accessed program regions. The algorithm starts with a 100% sampling rate for all code regions, and gradually reduces the sampling rate for hot regions. The tool creates an instrumented version of each function, enabling the logging of all memory accesses. Moreover, it incorporates a dynamic check at the entry of each function. This check, based on thread-specific sampling information, determines whether the execution should proceed in the uninstrumented or instrumented version of the function.

RACEZ [15] is another sampling-based tool, where the sampling of memory accesses is conducted by leveraging the hardware performance monitoring unit (PMU), instead of depending on the instrumentation of each memory access. Addresses are obtained from PMU samples. The tool utilizes a lockset-based analysis, where locking and unlocking primitives are required to be instrumented.

In recent years, some studies have investigated the potential of machine learning in data race detection. For instance, DeepRace [45] presents a deep learning approach to detect and highlight data races in C source code that utilize the Pthread library or OpenMP. It employs a one-layer convolutional neural network to autonomously learn and detect data race patterns. The tool is currently limited to detecting three forms of memory access constraints in total. In the case of code using the Pthread library, it can only identify mutual exclusion. For code making use of OpenMP, it can only detect the usage of the `private` clause and critical section directives.

4.4 ThreadMonitor (TMon)

In this section, we explain the motivation behind the development of ThreadMonitor (TMon), describe our design objectives, and provide an architectural overview of TMon. While discussing the tracing phase of TMon, we concurrently offer insights into Intel Processor Trace (Intel PT), highlighting its crucial role in shaping the low-overhead tracing mechanism of TMon.

4.4.1 Motivation

Detecting threading bugs stemming from data races poses a significant challenge in debugging multithreaded C/C++ programs. TSan, serving as the default dynamic data race detector in both clang and gcc compilers, is capable of conducting a thorough analysis to identify data races. However, it is not usable in many industrial testing setups due to their limited resources. TSan v3 is known to typically introduce a slowdown ranging from 5× to 10× and a memory overhead between 5× and 10× [97]. Such levels of performance degradation are often deemed unacceptable in many industrial testing environments.

The significant performance degradation exhibited by TSan is primarily attributed to its nature as an on-the-fly analysis tool. TSan analyzes the stream of program events online, leading to a substantial overhead in terms of both execution time and memory consumption. Therefore, although the data race detection logic employed by TSan is highly effective, its computational overhead makes it unsuitable for deployment in many use cases, for instance embedded systems. This notable performance impact, particularly in resource-constrained testing environments, underscores the need for an alternative tool capable of delivering robust data race detection with minimal effects on performance.

4.4.2 Architecture

The core objective driving the design of ThreadMonitor (TMon) is to offer the same comprehensive data race detection analysis as TSan, while causing significantly lower runtime overhead. This design goal aims to ensure the usability of TMon in real-world testing environments with constrained resources.

To avoid on-the-fly analysis overhead, we develop TMon as a postmortem tool featuring two phases : the tracing phase and the postmortem analysis phase. Unlike TSan, which analyzes program execution during runtime, TMon traces program events of interest and defers their analysis until after the program execution has finished. In addition to lower runtime overhead, another advantage of trace-based analysis is that it can be applied to simulation-based runs

of executable code. Moreover, on-the-fly race detection introduces larger timing disturbances along control paths than postmortem trace-based analysis. TMon mainly traces two types of program events : memory accesses and timing constraints among threads. Specifically, it traces the very events monitored by TSan, and ensures that, for each event, it records precisely the same runtime information captured by TSan for analysis purposes. This guarantees that, during the postmortem analysis of trace data, TMon can seamlessly conduct the same comprehensive data race detection analysis as TSan.

To maintain a minimal runtime overhead, adopting an efficient tracing approach is imperative. Conventional software-based tracing methods often introduce significant overhead, due to their intrusive nature. Recognizing this challenge, we focused on using hardware tracing as a promising alternative, given its limited impact on the traced program. Hardware tracing is incorporated in various architectures, notably in Intel, where the feature is referred to as Intel Processor Trace (Intel PT). Hardware tracing leverages dedicated hardware blocks to efficiently collect information about program execution in the form of different trace packets, ensuring a very low-overhead tracing approach. However, hardware tracing primarily focuses on detailed instruction tracing, and lacks a specialized feature for tracing memory accesses, a key requirement for our purposes. Nevertheless, Intel's `ptwrite` packet, a recently widely available feature within Intel PT, serves as a valuable compromise for our use case. A `ptwrite` packet is a user-generated 64-bit payload, produced by executing a `ptwrite` instruction. This instruction transmits the value of the 64-bit operand passed to it to the Intel PT hardware, where it is encoded within a `ptwrite` packet. The generation of `ptwrite` packets follows the same order as the execution sequence of the corresponding `ptwrite` instructions. Initially introduced in Atom processors, the `ptwrite` instruction is now widely available in the 12th generation of Intel Core processors. The execution of a `ptwrite` instruction is remarkably fast, requiring only two CPU cycles on our machine featuring a 12th Gen Intel i7-12700 CPU.

The tracing phase of TMon is realized through the implementation of two components : compile-time instrumentation of user code, and interception of the Pthread library functions. Our primary goal in implementing the compile-time instrumentation framework is to identify and instrument the same types of memory accesses, within user code, as monitored by TSan. Our framework instruments each memory access with a `ptwrite` instruction, whose payload encapsulates the identical runtime information captured by TSan for the analysis of that particular access. This instrumentation causes a very small instruction memory overhead. Additionally, we intercept Pthread library functions to account for timing constraints among program threads. Implemented as a static library, these interceptors emit `ptwrite` packets encoding runtime information about the invoked Pthread functions. The utilization of TMon remains independent of the tracer chosen to capture `ptwrite` packets. At the time of writing

this paper, the Linux `perf` tool stands as the most reliable choice for collecting `ptwrite` packets within Linux systems. Hence, we have selected the `perf` tool as our preferred trace collector.

The TMon postmortem analyzer processes decoded trace data, to identify potential data races in the traced program execution. It achieves this by reconstructing the sequence of memory access events, synchronization events, and mutual exclusion events. The information required for this reconstruction is gathered from the `ptwrite` packets and their associated metadata. The TMon postmortem analysis builds upon the data race detection logic employed by TSan v3. Additionally, it introduces two novel algorithmic improvements, aimed at enhancing data race detection coverage by reducing the likelihood of missing data races resulting from shadow memory overwriting.

4.5 Implementation

This section delves into the implementation of TMon, providing insights into each building block and the technical details involved in its deployment. We explain the rationale behind our design choices, offering a deep understanding of the mechanisms that enable the robust data race detection capabilities of TMon.

In each subsection, we first review the technical details of TSan's implementation, providing an in-depth look at its architecture, algorithms, and the strategies it employs. Following that, we explain the implementation details of TMon, focusing on its operational similarities to TSan.

4.5.1 Compile-time instrumentation of user code

The primary objective of our compile-time instrumentation framework is to identify and instrument various types of memory accesses within user code. It is also responsible for handling calls to memory intrinsic functions, as well as instrumenting function entry and exit points, if necessary, to allow providing a precise stack trace when reporting a potential data race. We implemented our framework as a function pass for the LLVM 17 compiler infrastructure, drawing inspiration from the function pass utilized by TSan for a similar purpose. In what follows, we introduce the main components of the TMon function pass, explaining how its instrumentation mechanism captures exactly the same runtime information as TSan, for the aforementioned events, by using the very low-overhead `ptwrite` instruction for instrumentation. This provides the groundwork for utilizing the same data race detection logic as TSan, and more, for analyzing memory accesses.

Assessing instrumentation eligibility of a function

The function pass begins by examining the attributes of a given function to assess its eligibility for instrumentation. Specifically, it checks whether the function possesses the `naked` attribute, which indicates the absence of standard prologue and epilogue sequences. Similar to TSan, TMon leaves functions with this attribute uninstrumented, since precisely locating data races requires instrumenting function entry and exit points, as we will elaborate shortly. Furthermore, the function pass verifies whether the function is explicitly marked to prevent all TMon instrumentation, by inspecting the presence of the `DisableTMonInstrumentation` attribute. If this attribute is detected, the function is exempted from any form of instrumentation. This attribute is designed to provide programmers with the flexibility to selectively leave certain functions uninstrumented. TSan provides a similar function attribute, with the same functionality.

Function traversal

Once a function qualifies for instrumentation, the function pass proceeds to traverse it, to identify the memory access instructions. Notably, TMon targets the same set of instructions as TSan : non-atomic memory accesses, atomic memory operations, and calls to the `memset/memmove/memcpy` intrinsic functions. This phase involves examining each instruction within the given function. All instances of the three aforementioned instruction types are gathered and categorized into three separate containers, facilitating specialized treatment for each instruction type in the subsequent phases of the function pass. However, certain non-atomic accesses may be proven redundant concerning data race detection. To mitigate unnecessary instrumentation overhead, and optimize the data race detection process, TSan conducts an analysis at the end of each basic block to identify three such redundancy cases. One case is the read-before-write scenario happening within the same basic block, with no calls occurring between them. In this situation, the read instruction can be safely excluded from instrumentation if the read and write instructions target the same memory address and none of them is volatile. Consequently, the function pass marks the write instruction as a compound access. The second case is reading an address that points to constant data. Such read instructions can also be excluded from instrumentation, as they cannot be involved in a data race. The third case involves instructions that access addressable variables that are not captured. These instructions can also be safely left uninstrumented, as such variables cannot be referenced from a different thread and, therefore, cannot be involved in a data race. TMon employs the same redundancy analysis, thereby instrumenting exactly the same instructions as TSan.

Instrumenting non-atomic memory accesses

TSan instruments non-atomic memory accesses by inserting a call to a specialized runtime library function, immediately before the access occurs. This allows TSan to track memory accesses, and conduct its data race detection analysis, by comparing the current access with previous accesses to the same memory location. To achieve this, the underlying data race detection logic requires to obtain six properties pertaining to each non-atomic memory access. First, it is imperative to determine the access type, distinguishing between read and write operations. Secondly, it is required to calculate the access size. This is accomplished based on the data type of the accessed memory location, and the data layout of the target platform. Notably, TSan supports access sizes of 1, 2, 4, 8, and 16 bytes, reflecting common memory access patterns. Any other access size is deemed unusual and the corresponding access instances are not instrumented. Thirdly, it is needed to identify whether the access is aligned. Fourthly, it is necessary to check if a write operation is marked as a compound access, indicating consecutive read-write operations to the same memory location, occurring within the same basic block, as previously discussed. Fifthly, for non-compound accesses, it is essential to check if the instruction accesses a volatile memory location. Compound volatile accesses are considered to be invalid. These five properties contribute to a total of 50 distinct types of non-atomic memory accesses. TSan encodes these five properties with a dedicated instrumentation function for each specific case, resulting in 50 distinct functions being used to instrument various types of non-atomic memory accesses. For instance, the runtime function `__tsan_read4` is utilized to instrument aligned non-volatile read operations of size four bytes. The final property that must be captured for each non-atomic access is the address of the accessed memory location, which serves as the sole argument passed to the corresponding instrumentation function.

TMon supports the same 50 different types of non-atomic memory accesses, and traces the same six properties for each access, thereby enabling its postmortem analyzer to apply the same data race detection logic implemented in the TSan runtime, for analyzing non-atomic accesses. This is accomplished by inserting a single `ptwrite` instruction immediately before each access, where the most significant byte of its payload cumulatively encodes the aforementioned first five properties, distinguishing between 50 various categories of non-atomic memory accesses, using a predefined set of unique values, each exclusively associated with an instrumentation function employed by TSan. Additionally, the six least significant bytes of the payload are dedicated to storing the address of the accessed memory location. Indeed, on the Intel 64 bits architecture, the first 2 bytes are unused, and the six least significant bytes are thus sufficient.

Instrumenting atomic memory operations

Similar to TSan, TMon supports C++11/C1x atomic memory operations. TSan instruments atomic operations using a similar approach to that used for instrumenting non-atomic accesses. This involves the insertion of a call to a specialized runtime library function, immediately preceding the occurrence of an atomic memory operation, aiming to detect potential data races associated with it. To properly analyze atomic accesses, the underlying data race detection logic requires to obtain three properties pertaining to each atomic operation. First, it is required to identify the operation type. TSan supports four types of atomic memory operations : atomic load, atomic store, atomic read-modify-write (RMW), and atomic compare-and-swap (CAS). Secondly, it is needed to compute the access size. This is done based on the data type of the accessed memory location, and the data layout of the underlying architecture. Notably, similar to non-atomic accesses, TSan supports access sizes of 8, 16, 32, 64, and 128 bits. Any other access size is regarded unusual and the the corresponding access instances are excluded from the instrumentation process. These two properties contribute to a total of 20 different types of atomic memory operations supported by TSan. TSan encodes these two properties by instrumenting each specific case with a dedicated instrumentation function, resulting in 20 distinct functions being used to instrument various types of atomic memory operations. For instance, the runtime function `__tsan_atomic8_load` is intended to instrument atomic load operations of size eight bits. The third property that must be captured for each atomic operation is the address of the accessed memory location, which serves as an argument passed to the corresponding instrumentation function.

TMon offers support for the same 20 different types of atomic memory operations, and traces the same three properties for each operation, thus allowing its postmortem analyzer to employ the same data race detection logic utilized in the TSan runtime for analyzing atomic accesses. This is achieved by inserting a single `ptwrite` instruction immediately before each atomic operation, with the most significant byte of its payload cumulatively encoding the aforementioned first two properties using 20 unique values each, exclusively associated with an atomic operation type, and the six least significant bytes storing the address of the accessed memory location.

Handling memory intrinsics

In the event that a `memset/memmove/memcpy` intrinsic is inlined during the code generation process, the potential occurrence of data races within it will evade detection. Thus, it becomes imperative to either prevent the intrinsic from getting inlined or apply direct instrumentation to address this concern. Akin to TSan, TMon avoids instrumenting these intrinsics due to the

intricacies associated with instrumenting them. Instead, the alternative approach is taken, whereby these intrinsics are substituted with regular function calls. These calls are subsequently intercepted by the runtime library, to enable the necessary monitoring and analysis procedures.

Instrumenting function entry and exit

TSan instruments the entry and exit points of a function, if it contains instrumented memory accesses, preserving a precise stack trace for every access. This stack trace will be incorporated into the potential data race reports, offering developers a comprehensive view of the execution context, leading up to the identified data race. Upon function entry, TSan captures the return address of the current function, using the `llvm.returnaddress` code generator intrinsic, and passes it as the sole argument to the runtime library function `__tsan_func_entry`. TSan marks a function exit by invoking the runtime library function `__tsan_func_exit`, which takes no argument.

Similar to TSan, TMon instruments function entry and exit points, provided that the function includes instrumented memory accesses. TMon instruments a function entry using a single `ptwrite` instruction, with the most significant byte of its payload serving as an indicator for a function entry event, and the six least significant bytes storing the return address of the current function. The instruction is inserted immediately before the first `non-PHINode` instruction of the entry basic block. TMon uses a single `ptwrite` instruction to instrument a function exit, where the most significant byte of its payload serves as an indicator for a function exit event. The instruction is inserted immediately before all escape points from the function.

4.5.2 Intercepting Pthread functions

Pthread functions impose timing constraints among program threads. Therefore, it is essential for a data race detector to be aware of their invocations. Similar to TSan, TMon intercepts Pthread functions. We implement our interceptors as a static library. Inside each interceptor, we make a call to the actual pthread function and then, before returning the control flow, we emit a `ptwrite` packet, which encodes runtime information regarding that Pthread function invocation. The recorded information depends on the function, but it is typically a combination of the thread handle and the return status of the actual function.

The preceding details encompass the information gathered during runtime, specifically in the tracing phase of TMon. Subsequently, we will delve into the postmortem processing of the

trace data, which potentially executes on a separate host.

4.5.3 Trace decoder

As part of its trace decoder, TMon incorporates components of the Linux `perf` utility responsible for the implementation of the `script` command, which is utilized for decoding traces collected by the `perf` tool. TMon achieves this integration through the development of a specialized API that facilitates seamless communication between the decoder and the postmortem analyzer, where the data race detection logic resides. The decoder is designed to accept raw trace data as input, identify all the `ptwrite` packets within the input trace data, and subsequently return the decoded `ptwrite` packets to the analyzer, in the exact same order they appeared in the execution trace. Each decoded `ptwrite` packet consists of an eight-byte payload, which is the value of the operand passed to the corresponding `ptwrite` instruction, alongside metadata including the OS thread identifier (TID) and process identifier (PID) of the thread executing that instruction.

4.5.4 Postmortem analyzer

The postmortem analyzer is responsible for processing the decoded trace data, to determine whether the traced program execution exhibited data races. This is accomplished by reconstructing the sequence of memory access events, synchronization events, and mutual exclusion events that occurred during the traced execution of the program. It leverages the information encoded within the `ptwrite` packets and the associated metadata.

As previously elaborated, TMon traces the same runtime information as that captured by TSan, for analyzing memory access events, synchronization events, and mutual exclusion events. Consequently, its postmortem analyzer is capable of performing, at the very least, the same data race detection analysis carried out by TSan. The postmortem analyzer in TMon builds upon the data race detection algorithm utilized by TSan, enhancing its coverage through the introduction of novel algorithmic improvements.

The base data race detection logic is a pure happens-before algorithm, examining conflicting memory accesses to determine the existence of a timing constraint ordering them. Conflicting memory accesses manifest when distinct threads access overlapping memory locations, with the additional criterion that at least one of the accesses is a write operation. Establishing the happens-before relation, among conflicting memory accesses, is realized by maintaining a vector clock for each running thread, as formally defined bellow.

Définition 3 (Thread-associated Vector Clock)**.** *In a program with N running threads, the*

vector clock associated with a thread t_i is denoted by VC_i and defined as a vector of N logical clocks, where the entry $VC_i[j]$ represents the latest update of thread t_i on the state of the logical clock in thread t_j. Similarly, the entry $VC_i[i]$ stores the own logical clock of thread t_i. Thread-associated vector clocks are updated based on the following principles :

- *All vector clocks are initialized to zeros.*
- *For every event on a thread that involves imposing a timing constraint, its own logical clock is incremented by one.*
- *When a timing constraint is enforced between two threads (for instance, when thread t_j obtains a mutual exclusion lock after thread t_i releases it), the second thread (thread t_j) updates each element of its vector clock by taking the maximum of the current value in its own vector clock and the corresponding value in the vector clock of the first thread, when the timing constraint started to form (the vector clock of thread t_i right after it released the mutual exclusion lock).*

In order to ascertain whether two conflicting memory accesses are ordered by the happens-before relation, a comparative analysis of the vector clocks associated with the two executing threads is conducted. Definition 4 presents the general rule of establishing the happens-before relation through the comparison of the two vector clocks.

Définition 4 (Exhaustive Vector Clock Comparison)**.** *Consider A_a and A_b as two conflicting memory accesses, attributed respectively to threads T_i and T_j. Access A_a is considered to happen before access A_b, denoted by $A_a \rightarrow A_b$, if and only if, for every index k, $VC_{T_i}(A_a)[k] \leq VC_{T_j}(A_b)[k]$, where $VC_{T_i}(A_a)$ and $VC_{T_j}(A_b)$ denote the vector clocks of threads t_i and t_j, immediately after the execution of accesses A_a and A_b, respectively.*

This definition requires an exhaustive comparison of the two vector clocks for each entry, which is implemented by TSan v2. However, it can be proved that exhaustive vector clock comparisons can be simplified. Specifically, the refined approach proves that it is sufficient to limit the comparison to a specific entry within each vector clock – the entry corresponding to the thread whose memory access event was observed before the other. Definition 5 outlines this approach, which is implemented by the latest version of TSan (TSan v3) and also by TMon.

Définition 5 (Simplified Vector Clock Comparison)**.** *Consider A_a and A_b as two conflicting memory accesses, attributed respectively to threads T_i and T_j, with access A_a being observed before access A_b. Access A_a is considered to happen before access A_b, denoted by $A_a \rightarrow A_b$, if and only if $VC_{T_j}(A_b)[i] \geq VC_{T_i}(A_a)[i]$, where $VC_{T_i}(A_a)$ and $VC_{T_j}(A_b)$ denote the vector clocks of threads t_i and t_j, immediately after the execution of accesses A_a and A_b, respectively.*

Following Definition 5, the two conflicting memory accesses A_a and A_b give rise to a data race if they are not ordered by the happens-before relation, namely $A_a \not\rightarrow A_b$. This observation indicates the absence of a timing constraint that would establish an order between the two accesses.

TMon employs a postmortem adaptation of the shadow memory technique utilized by TSan to track conflicting memory accesses, and to retain essential information for their subsequent evaluations within the context of the happens-before relation, as elaborated upon earlier. The significant distinction lies in the fact that, unlike TSan, TMon does not impact the data memory consumption of the application under test, given that it constructs and maintains the shadow memory during postmortem analysis. In this context, the shadow memory functions as a data structure designed to store information derived from the trace data, regarding memory accesses.

Every consecutive eight bytes of application memory are mapped to a dedicated set of *shadow cells*. Each shadow cell encodes an access to the associated application memory region. Upon detecting a new memory access, it is compared with prior conflicting accesses encoded by shadow cells, to determine whether all these earlier accesses can be proved to have happened before the current access, following the principles outlined in Definition 5. Each shadow cell, spanning four bytes, features five components. Firstly, 8 bits are devoted to indicate which bytes of the corresponding eight-byte application memory region have been accessed by the operation, where a bit value of one signifies the involvement of the corresponding byte in the access. Following this, 8 additional bits are designated to store the internal thread ID of the accessing thread. This unique identifier is assigned by the data race detection logic to each thread. Additionally, 14 bits are reserved for saving the own logical clock of the accessing thread at the time of access. Furthermore, 1 bit denotes whether the access was a write or a read, and 1 additional bit encodes whether it was an atomic operation.

Increasing data race detection accuracy

A notable factor contributing to missing data races in TSan is the necessity to overwrite shadow cells, a constraint imposed to limit the memory overhead, by having a finite small number of cells for each memory location. When a shadow cell is overwritten, the memory access that was previously encoded by that particular cell is no longer preserved. Consequently, any potential data race that the overwritten access might have formed, with a subsequent memory access, remains undetected. This occurrence arises due to the inherent trade-off between memory consumption and race detection coverage in TSan. We present the following two algorithmic improvements to mitigate this issue.

Allocating more shadow cells By default, TSan allocates only four shadow cells for each consecutive eight bytes of application memory. Although it might be technically possible to increase the number of allocated shadow cells, through modifying the source code of TSan, such a strategy introduces significant runtime memory overhead, that can be impractical in many real-world testing scenarios. Even utilizing this modest count, of four shadow cells per eight bytes of application memory, is not without its implications. In various real-world testing scenarios, the resultant memory overhead can quickly become intolerable, posing a challenge for practical application and deployment. This tension between achieving a finer granularity of race detection and the ensuing memory demands remains a critical concern in the design and implementation of runtime analysis tools like TSan.

TMon, on the other hand, operates as a postmortem analysis tool, and is not constrained by the same runtime memory concerns as TSan. Consequently, the allocation of a greater number of shadow cells becomes a viable strategy to enhance the granularity of race detection. In the postmortem analysis context, the capability of TMon to allocate additional shadow cells, for enhanced race detection, is hinged on the available memory resources of the system running the postmortem analysis. This attribute positions TMon to potentially achieve heightened precision in detecting data races, without incurring the runtime data memory overhead concerns that TSan must address.

Better overwriting policy When it is required to overwrite a shadow cell, TSan adopts a random selection strategy, where a shadow cell is randomly chosen for overwriting. However, this approach does not prioritize the significance of the memory access being overwritten, potentially leading to the loss of important access information.

In contrast, TMon introduces a refined algorithmic improvement to tackle the overwriting dilemma more strategically. Instead of a random selection, TMon proposes a targeted approach : selecting the shadow cell associated with the access involving the least number of bytes. This choice is particularly advantageous, because it reduces the likelihood of the chosen access overlapping with subsequent accesses to the same eight-byte application region. Consequently, the risk of losing crucial data races, due to overwriting shadow cells, is diminished. By prioritizing the shadow cell associated with the shorter access, TMon minimizes the chances of missing data races due to overwriting shadow cells.

4.5.5 Example

As an illustrative example of how TMon operates, consider the sample C program shown in Listing 1, as provided in clang's documentation for TSan [97]. In this program, the `main`

thread and the thread it spawns (`Thread1`) form a data race, as they access the global variable `Global`, without any timing constraint ordering the two accesses.

Listings 2 and 3 represent the LLVM IR of the sample code in Listing 1, after being instrumented by TSan and TMon respectively. As previously elaborated, and as can be seen from these two listings, TMon instruments exactly the same memory access events as TSan. Using `ptwrite` instructions, TMon records the same runtime information for each memory access operation, as captured by TSan instrumentation functions. The recorded `ptwrite` packets will be decoded and analyzed by the postmortem stage of TMon, performing the same verification carried out by TSan at runtime, but at the postmortem stage.

Listing 4.1 Example Program

```
#include <pthread.h>

int Global;

void *Thread1(void *x) {
  Global = 42;
  return x;
}

int main() {
  pthread_t t;
  pthread_create(&t, NULL, Thread1, NULL);
  Global = 43;
  pthread_join(t, NULL);
  return Global;
}
```

Listing 4.2 TSan Instrumented IR Output

```
define dso_local ptr @Thread1(ptr noundef %x) {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
call void @__tsan_func_entry(ptr %0)
...
call void @__tsan_write4(ptr @Global)
store i32 42, ptr @Global, align 4
...
call void @__tsan_func_exit()
ret ...
}

define dso_local i32 @main() {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
call void @__tsan_func_entry(ptr %0)
...
%t = alloca i64, align 8
```

```
...
call void @__tsan_write4(ptr @Global)
store i32 43, ptr @Global, align 4
call void @__tsan_read8(ptr %t)
%1 = load i64, ptr %t, align 8
...
call void @__tsan_read4(ptr @Global)
%2 = load i32, ptr @Global, align 4
call void @__tsan_func_exit(),
ret ...
}
```

Listing 4.3 TMon Instrumented IR Output

```
define dso_local ptr @Thread1(ptr noundef %x) {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
%1 = ptrtoint ptr %0 to i64
%ptw.funcentry = or i64 %1, 72057594037927936
call void asm "ptwriteq $0", "rm"(i64 %ptw.funcentry)
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64
    864691128455135232))
store i32 42, ptr @Global, align 4
...
call void asm "ptwriteq $0", "rm"(i64 144115188075855872)
ret ...
}

define dso_local i32 @main() {
entry:
%0 = call ptr @llvm.returnaddress(i32 0)
%1 = ptrtoint ptr %0 to i64
%ptw.funcentry = or i64 %1, 72057594037927936
call void asm "ptwriteq $0", "rm"(i64 %ptw.funcentry)
...
%t = alloca i64, align 8
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64
    864691128455135232))
store i32 43, ptr @Global, align 4
%2 = ptrtoint ptr %t to i64
%ptw.rw = or i64 %2, 576460752303423488
call void asm "ptwriteq $0", "rm"(i64 %ptw.rw)
%3 = load i64, ptr %t, align 8
...
call void asm "ptwriteq $0", "rm"(i64 or (i64 ptrtoint (ptr @Global to i64), i64
    504403158265495552))
%4 = load i32, ptr @Global, align 4
call void asm "ptwriteq $0", "rm"(i64 144115188075855872)
ret ...
}
```

4.6 Evaluation

We compare the performance of TMon against TSan v3 on C/C++ benchmarks from the SPEC CPU 2017 suite, and a parallelized implementation of the Fourier transform. All evaluations are conducted on a desktop system, equipped with a 12th Gen Intel i7-12700 CPU and 16 GB of RAM, running Ubuntu 22.04 LTS, with Linux kernel version 6.2.0. In addition to evaluating the runtime impact of TMon, we also explore potential runtime effects introduced by the tracer, the Linux `perf` tool. Furthermore, we also provide insights into the time required for decoding trace data and performing the postmortem analysis, across various trace data sizes.

4.6.1 SPEC CPU 2017

In the first stage of our evaluation study, we assess the runtime efficiency of TMon using a set of four SPEC CPU 2017 benchmarks, comparing its performance with the latest version of TSan. All benchmarks are executed with the test input data provided by the suite. The findings of this assessment are summarized in Table 5.1. In this table, the "Native" column represents the executable size, execution time, and memory consumption - measured in terms of maximum resident set size (MRSS) - of each benchmark when compiled without any instrumentation. The executable size, execution time, and memory consumption of running each benchmark, with TSan and TMon enabled, are reported in distinct columns, as ratios relative to their respective uninstrumented native runs. Each benchmark was executed ten times, and the results reported in the table represent the average values across these runs.

Comparing the executable sizes between a native compilation and one with TMon instrumentation confirms that the instruction memory overhead of TMon, attributed to the instrumentation, is very small. Additionally, as discussed earlier, TMon incurs no direct data memory overhead. Consequently, it can be inferred that the reported memory overhead for TMon is predominantly associated with the tracer, the Linux `perf` tool, and not TMon itself. In essence, the observed increase in memory usage is directly attributed to the trace data collection activities conducted by the `perf` tool, throughout the program execution. This distinction is significant, as the memory overhead introduced by the tracer is more flexible and less restrictive than the direct memory overhead caused by TSan. The buffer sizes utilized by the tracer can be adjusted to accommodate resource limitations, providing a degree of control over the memory impact of the tracer. In our evaluations, we employed default configurations for the Linux `perf` tool. Similarly, the reported execution time for TMon is predominantly due to the overhead of tracing and writing the trace data to disk, using the

`perf` tool.

As shown in Table 5.1, on average, utilizing TMon results in more than a twofold reduction in memory overhead, compared to TSan. Moreover, on average, TMon imposes nearly two times less execution time overhead in comparison to TSan. The substantial reduction, in both memory and execution time overhead, underscores the performance gains offered by TMon over TSan.

TABLE 4.1 Performance Evaluation of TMon and TSan on SPEC CPU 2017 Benchmarks

Benchmark	**Native**			**TSan**			**TMon**		
	Exec. Size	Time (sec)	MRSS (MB)	Exec. Size (×)	Time (×)	MRSS (×)	Exec. Size (×)	Time (×)	MRSS (×)
mcf	113 KB	5.9	292	13.3×	3.7×	2.9×	1.2×	2.8×	2.1×
lbm	50 KB	1.0	420	28.5×	4.1×	3.0×	1.9×	5.1×	1.4×
namd	3 MB	1.8	160	1.8×	7.5×	3.1×	1.2×	2.2×	1.5×
parest	75 MB	2.0	99	1.3×	9.0×	4.7×	1.1×	2.9×	1.5×
Average				11.2×	6.1×	3.5×	1.3×	3.2×	1.6×

Each benchmark was executed ten times, and the reported results represent the average values across these runs.

4.6.2 Parallelized FT benchmark

In the second phase of our evaluation study, we employ a parallelized implementation of the Fourier transform - a prominent algorithm widely used in the domain of signal processing - as an experimental benchmark, available in [98]. This experimental study includes a diverse set of tests, varying across three distinct thread counts and two different input sizes, introducing different workloads. This yields a total of six unique scenarios, enabling a thorough comparison of the tools. The results of our experiments are presented in Table 4.2.

As shown in this table, TMon consistently imposes notably lower execution time overhead, across all the tests, in comparison to TSan. On average, the utilization of TMon exhibits a speed enhancement exceeding sixfold compared to TSan. In comparison to a native run, TMon introduces a modest 20% execution time overhead, on average, encompassing the overhead introduced by the Linux `perf` tool to trace the program execution and write the trace data to disk. In contrast, TSan results in an average slowdown of more than seven times. This benchmark inherently operates with a minimal memory footprint (2.7 MB - 4.0 MB), where even the default memory consumption of the tracer, the `perf` tool, surpasses its native usage. However, this circumstance is not a concern for two reasons. Firstly, applications with such low memory requirements are less susceptible to potential memory limitations in a testing environment, resulting in minimal impact. Secondly, as clarified earlier in Section 4.6.1, the flexibility to adjust tracer buffer sizes offers a solution to control the memory impact of the tracer. It is important to note that, similar to the SPEC CPU benchmarks analyzed in

Section 4.6.1, the instruction memory overhead of TMon remains very low for this benchmark. Specifically, the executable sizes for a native compilation, TMon instrumentation, and TSan instrumentation are 22 kB, 46 kB, and 1.3 MB, respectively.

TABLE 4.2 Performance Comparison of TMon and TSan on the Parallelized Fourier Transform Benchmark

#Threads	**#Input Values**	**Native**		**TSan**		**TMon**
		Time (sec)	MRSS (MB)	Time (×)	MRSS (×)	Time (×)
5	2^{15}	8.0	2.7	3.2×	9.8×	1.3×
5	2^{16}	32.0	4.0	3.0×	7.4×	1.1×
10	2^{15}	5.4	2.7	8.1×	11.4×	1.2×
10	2^{16}	21.5	3.2	8.0×	10.6×	1.1×
15	2^{15}	3.8	2.7	11.3×	13.1×	1.1×
15	2^{16}	15.2	3.2	11.1×	11.9×	1.2×
Average				7.5×	10.7×	1.2×

Table 4.3 provides insights into trace decoding and analysis times for varying sizes of trace data. It reports the size of the trace data produced in each testing scenario, the time required to decode each set of trace data, and the time needed to perform the postmortem analysis of TMon on each decoded trace data. This information may also provide a perspective on the correlation between program execution time and the volume of generated trace data. However, it is essential to note that, in general, the size of the generated trace data primarily depends on the number of memory accesses made by the application under test. Decoding the trace data and performing the postmortem analysis can be done on a machine different from the one where the program execution occurred. However, in this experiment, we performed them on the same machine. As shown in this table, the time required for conducting the postmortem analysis is almost comparable to the execution time with TSan. In contrast to the invasive runtime overhead of TSan, directly impacting program execution, the postmortem analysis time is much less problematic, occurring after the program has completed its execution.

TABLE 4.3 Trace Data Sizes and Trace Decoding and Postmortem Analysis Times

#Threads	**#Input Values**	**Trace Data Size (GB)**	**Trace Decoding (sec)**	**Postmortem Analysis (sec)**
5	2^{15}	8.7	91	20
5	2^{16}	14.3	188	33
10	2^{15}	9.9	122	24
10	2^{16}	22.6	235	48
15	2^{15}	10.1	126	26
15	2^{16}	25.0	267	55

4.7 Conclusion and Future Work

Identifying data races in multithreaded C/C++ programs is a very challenging task, due to intricate thread interdependencies, the potential for non-deterministic execution, and the complexity of shared memory management. While ThreadSanitizer (TSan) serves as a robust on-the-fly data race detector in clang and gcc compilers, its applicability in many industrial testing environments is hindered by its significant performance degradation, making it unsuitable for time-sensitive or resource-intensive applications.

To address this issue, this study introduces ThreadMonitor (TMon), a low-overhead postmortem data race detector designed for multithreaded C/C++ programs utilizing the Pthread library. TMon is specifically designed to provide the same comprehensive data race detection analysis as TSan, but with significantly reduced runtime overhead. To achieve this, TMon features two main phases : the tracing phase and the postmortem analysis phase. During its tracing phase, TMon leverages Intel's very low-overhead `ptwrite` packet, a recently widespread feature of Intel Processor Trace, to trace the same program events monitored by TSan. By utilizing `ptwrite` packets, for each event, TMon records precisely the same runtime information, captured by TSan, for analysis purposes. Later, the postmortem analyzer of TMon reconstructs the sequence of program events, using the trace data, to determine whether the traced program execution exhibited data races. The postmortem analyzer builds upon the data race detection logic implemented by the latest version of TSan, and further introduces two novel algorithmic improvements, to enhance its data race detection coverage by reducing the likelihood of missing data races resulting from shadow memory overwriting.

TMon has no direct data memory overhead, incurs minimal instruction memory overhead, and causes a very small slowdown. Our experiments demonstrate that, on average, TMon introduces 2 to 6 times less execution time overhead compared to TSan, factoring in the overhead associated with writing the trace data to disk.

The novel technique proposed in this paper is extendable to any other architecture or tracing facility, provided that the required information can still be traced at a reasonably low runtime overhead.

The evolution of this methodology over time holds the potential for identifying data races originating from Direct Memory Access (DMA) by embedded systems peripherals. In such environments, hardware blocks may access memory in manners that resemble data races, diverging from the conventional memory accesses initiated solely by CPU instruction execution. This necessitates the implementation of trace-based analysis for Direct Memory Access events. Although challenging, this approach is feasible and can significantly enhance data race

detection in complex embedded systems. Moreover, future development of this approach can lead to the hardware-based implementation of data race detection, accelerating this process and enabling real-time identification of data races during program execution.

As another possible future work, we aim to explore the adaptation of a similar approach to design other postmortem tools that emulate the analysis behaviour of other well-known runtime verification tools, such as AddressSanitizer (ASan).

CHAPITRE 5 ARTICLE 2 : ADDRESSMONITOR : LOW-OVERHEAD HEAP SPATIAL AND TEMPORAL SAFETY FOR C

Auteurs : Farzam Dorostkar, Michel Dagenais, Heng Li, Vince Bridgers, Ankush Tyagi et Vladislav Aranov.

Soumis à Journal of Systems and Software

Date : 21 Octobre 2025

5.1 Abstract

The C programming language enables developers to access and manage memory resources explicitly. Providing this level of control over memory, however, also introduces considerable verification challenges. Specifically, the lack of inherent mechanisms to verify the temporal and spatial safety of memory accesses necessitates developer-enforced verification that is an acknowledged source of critical errors in programs written in this language. While widely adopted robust techniques have greatly improved stack protection over time, efforts to enforce heap safety have not reached the same level of practical effectiveness. This limitation has triggered notable interest, within both academia and industry, to develop tools aimed at detecting violations of heap access safety. Among these contributions, dynamic verification techniques are recognized for their higher detection accuracy and better scalability compared to static approaches. However, state-of-the-art dynamic tools cannot be used in many real-world testing setups due to their substantial memory and computational overhead, which makes them particularly unsuitable for deployment in resource-constrained environments such as embedded systems. Moreover, these tools often have a limited detection scope, identifying only a small subset of violation types. To address these practical challenges, we present AddressMonitor (AMon), a low-overhead runtime verification tool designed to detect a broad range of temporal and spatial heap access violations in C programs. AMon is built upon a combination of pointer tainting and compile-time analysis and transformations. At compile-time, it identifies heap pointer operands in memory access operations, generates an untainted variant for each heap pointer, and replaces each original heap pointer operand with its untainted variant in those operations. At runtime, it assigns a unique taint value to each allocated heap objects, embeds this value into the unused bits of the returned pointer, and builds an object table to maintain spatial, temporal, and debugging information on user-allocated heap objects throughout program execution.

5.2 Introduction

The C programming language grants developers explicit control over memory. This notably includes the ability to arbitrarily access memory through direct pointer manipulation and dereferencing, as well as explicitly managing the lifetime of dynamically allocated memory regions, which eliminates the need for costly automated garbage collection. Although this low-level control over memory is particularly advantageous in domains demanding high performance, such as system programming, it comes at the cost of lacking built-in mechanisms to enforce memory access safety. This limitation is a recognized source of errors in C programs, frequently leading to vulnerabilities that pose significant security risks [99, 100].

There are two fundamental aspects to memory access safety that concern both heap and stack memory : *spatial safety* and *temporal safety* [101]. Spatial safety ensures that every memory access strictly adheres to the intended boundaries of the allocated object. A violation of spatial safety is commonly referred to as an out-of-bounds access error. On the other hand, temporal safety ensures that objects are not accessed beyond their intended lifetime. Two common types of temporal safety violations associated with heap memory are double-free and use-after-free errors. A double-free error occurs when a program attempts to free a dynamically allocated memory region more than once, while a use-after-free error arises when a program accesses memory after it has been freed. In the latter case, the pointer that still references the deallocated memory is known as a dangling pointer. The stack memory analogue of a use-after-free error is a use-after-return error, which occurs when a local variable is accessed after the function that allocated it has returned.

The failure to maintain spatial or temporal safety is a significant source of security vulnerabilities, as it can allow attackers to access unauthorized memory regions. This may result in severe consequences, including the corruption or theft of sensitive data and the hijacking of control flow [102–104]. Furthermore, such violations can introduce memory corruptions that are very difficult to diagnose. These corruptions may not cause immediate failure, but rather propagate through program execution, potentially accumulate, and ultimately lead to unexpected behavior or crashes in other parts of the program, making the precise identification of the root cause extremely challenging [105].

The severe consequences of violating memory access safety have motivated extensive research in memory safety. Although well-developed strategies have substantially enhanced stack protection over time [46–48], efforts to enforce heap access safety have remained comparatively less mature and effective. This limitation has sustained notable interest within both academia and industry toward developing tools for detecting violations of heap access safety. Among

these developments, dynamic verification techniques are widely acknowledged for their superior detection accuracy and scalability in comparison to static approaches. Despite these advantages, state-of-the-art dynamic verification tools are not practical in many real-world testing setups because of their notable memory and computational overhead, which specifically makes them unsuitable for deployment in test environments with limited resources such as embedded systems. Furthermore, these tools typically have a restricted detection scope, identifying only a small subset of violation types while leaving others undetected.

5.2.1 Contributions and paper organization

The main contributions of this research are as follows.

— In this paper, we present AddressMonitor (AMon), a low-overhead runtime verification tool designed to detect heap spatial and temporal safety violations in C programs. Specifically, AMon detects out-of-bound accesses, use-after-frees, double-frees, and memory leaks. Additionally, it is able to identify two other unsafe programming practices related to heap access : passing a non-base pointer to standard memory reallocation or deallocation functions, and potential use-after-free conditions arising from memory reallocation. To support replication and further development of our work, we provide a replication package that includes the implementation of AMon. This package is publicly accessible as an open-source project on GitHub [1].

— AMon is specifically designed to provide comprehensive detection capabilities while maintaining low overhead, ensuring its suitability for deployment in resource-constrained environments such as embedded systems. To assess its effectiveness, we evaluate the detection capabilities and runtime overhead of AMon using a set of synthetic and real-world benchmarks, and compared it with baselines. Furthermore, we present a detailed case study on the deployment of AMon in an industrial embedded environment. Our co-authors at Ericsson successfully integrated AMon into one of their internal proprietary Baseband embedded computing systems with minimal effort. In their initial tests, AMon has already identified potential safety violations in their code.

The remainder of this paper is organized as follows. In Section 5.3, we provide a background on heap access safety. In Section 5.4, we review a range of tools in the literature designed to detect violations of memory access safety. Section 5.5 is devoted to explaining the general architecture of AMon. In Sections 5.6 and 5.7, we explain the design and implementation of the main building blocks of AMon. In Section 5.8, we present our evaluation studies. Section 5.9 concludes this paper by summarizing the main points and contributions.

1. `https://github.com/farzamdorostkar/amon`

5.3 Background

This section first provides background on the continued importance of addressing heap safety violations. It then reviews the primary approaches proposed to combat such violations, discussing the underlying motivations behind each method and the key challenges associated with their implementation.

5.3.1 Heap access safety

Despite decades of research, addressing memory safety violations remains a major problem in software verification. Although stack protection has benefited from the development of robust and widely adopted techniques, particularly dedicated compiler extensions [46–48], efforts to address heap safety violations have not reached the same level of practical effectiveness. This is further evidenced by recent reports documenting vulnerabilities in real-world software. For instance, the Common Weakness Enumeration (CWE) root cause mapping of the 2023 Known Exploited Vulnerabilities (KEV) catalog, which documents actively exploited vulnerabilities in widely deployed software systems across cybersecurity and networking domains, highlights use-after-free (CWE-416) and heap-based buffer overflow (CWE-122) as the first and second most frequently exploited weaknesses, respectively [49]. As another example, the 2023 Red Hat Product Security Risk Report indicates that four of the five most frequently observed software weakness types were memory corruption issues, primarily affecting Red Hat Enterprise Linux and other codebases written in C [50]. Notably, the report identifies use-after-free as the most prevalent weakness among these four.

5.3.2 Addressing heap safety violations

Attackers typically exploit heap safety violations by leveraging implementation weaknesses of the underlying dynamic memory allocators. For instance, `ptmalloc`, the base memory allocator in the GNU C Library, is prone to code injection attacks resulting from heap out-of-bound accesses or double-free attempts [51]. Consequently, some security-focused research efforts propose developing custom dynamic memory allocators that are more secure, with improved resistance to potential vulnerabilities that arise from heap safety violations [52–55]. For instance, FreeGuard [53] is a security-enhanced memory allocator designed to mitigate heap-related vulnerabilities by incorporating randomized allocations, object isolation, and metadata protection. However, such approaches have limitations that prevent their widespread adoption, mainly including considerable overhead, effectiveness limited to specific attack types, and restricted protection granularity.

Another group of research works explore hardware-assisted defenses to prevent memory access violations or mitigate their potential security consequences. These approaches often utilize hardware features alongside software-based techniques to provide a level of memory access safety with lower overhead [56, 57]. However, their reliance on specific hardware extensions limits their applicability. Additionally, their viability is tied to the continued support for these technologies. For instance, the discontinuation of support for Intel Memory Protection Extensions (MPX) [58], a hardware feature designed to facilitate bounds verification, has reduced the practical relevance of tools reliant on this technology [59, 60].

Software-based detection methods represent another category of techniques that aim to identify memory access violations. These methods rely solely on software-driven events and mechanisms to perform program analysis [61]. Compared to hardware-assisted techniques, they provide enhanced flexibility by eliminating the need for specialized hardware extensions. In contrast to approaches employing custom memory allocators, they typically preserve the original memory layout of allocated objects, enabling easier integration with existing codebases. Furthermore, software-based methods generally enable the detection of a wider range of violation types [62]. They also usually provide greater granularity, enhancing the precision and depth of analysis for identified violations. In industrial environments, based on the experience of our industry partners, software-based detection approaches are often preferred over the aforementioned techniques because of their superior adaptability, which aligns with operational and economic considerations. Organizations typically prioritize a generalized detection solution that is flexible enough to be deployed across the diverse hardware and software platforms within their ecosystem. This preference comes from two key advantages :

— Unified detection coverage : A generalized approach ensures consistent detection capabilities across various systems, facilitating the adoption of standardized security practices and, as a result, enhancing predictability in cross-system vulnerability assessments.
— Efficient development and maintenance : Adopting a generalized detection method allows companies to centralize their efforts. This strategy minimizes redundancy and simplifies workflows, reducing the need to manage distinct tools with overlapping functionalities, thus improving operational efficiency and lowering costs.

Although software-based techniques offer advantages of such importance, existing tools exhibit serious limitations that restrict their applicability. A principal limitation is the typically high overhead they impose, which makes them impractical for setups with limited resources.

5.4 Related Work

The critical consequences of failing to ensure spatial and temporal memory safety have triggered significant interest in both academia and industry to develop automated tools to detect these violations. Existing tools are typically designed to detect a subset of memory safety violations and target specific hardware architectures and software platforms. The literature explores a variety of analysis techniques aimed at addressing the challenges associated with memory safety verification. Despite the extensive body of work in this area, it remains a highly active research domain, as existing approaches continue to suffer from serious limitations, such as incomplete detection, lack of soundness, significant analysis overhead, or limited portability. To facilitate a systematic examination of these techniques, we classify them into two main groups : static and dynamic analysis tools. We review these two types of analysis in dedicated subsections. Each subsection discusses the inherent advantages and disadvantages of the respective category, provides an overview of some notable tools within that class, and highlights experimental studies in the literature comparing existing solutions.

5.4.1 Static analysis for detecting memory safety violations

Static analysis approaches for verifying adherence to memory safety constraints analyze memory accesses in the source code or its intermediate representations, without executing the program. The principal advantage of static approaches over dynamic techniques is their capacity to achieve comprehensive code coverage through a complete analysis of the entire program. However, this potential strength is undermined by fundamental limitations that can lead to imprecise analysis. These limitations arise from the challenges associated with accurately analyzing complex program properties such as pointer aliasing, inter-procedural dependencies, and complicated control flow constructs including loops, indirect calls, and conditional branches. Such properties commonly depend on runtime information to be fully resolved, making their precise analysis infeasible in a purely static context. Furthermore, in-depth analysis of even moderately large code bases is often computationally infeasible, introducing serious scalability issues. To address these constraints, static tools typically employ incomplete analyses, relying on approximations such as simplified pointer analysis, path-insensitive analysis, and pure intra-procedural or partial inter-procedural analysis [63, 64]. Such approximations inevitably introduce uncertainty into the analysis. To handle this uncertainty, static tools often apply a combination of conservative strategies, which may overestimate potential issues and consequently lead to false positives, and techniques that downplay uncertainty, which may give rise to false negatives [65,66]. Achieving an optimal trade-off between reducing false positives and false negatives is another core challenge in static program

analysis.

Pereira et al. [67] conduct an experimental analysis to evaluate the capability of widely utilized static analysis tools and static code features in detecting buffer overflow vulnerabilities in large-scale C/C++ projects. In the study, they use the Linux Kernel, Mozilla, and Xen as three representative software projects with extensive codebases and significant numbers of detected buffer overflows throughout their development. The authors assess the effectiveness of CppCheck and Flawfinder, two commonly used C/C++ static analysis tools, in identifying buffer overflows in these projects. The team specifically analyze the differences in alerts generated by these tools before and after fixes were applied to buffer overflow errors, noting the disappearance, persistence, or emergence of new alerts in the corrected code. Their analysis reveals that, in most instances, the tools generate identical alerts for both code with buffer overflow and its debugged version, demonstrating their limited capability to detect buffer overflows, and consequently, the need for defining new rules in these tools to specifically target these vulnerabilities. The study then investigates the potential correlation between some frequently used static code metrics and the presence of buffer overflows. Adopting a methodological approach similar to their earlier analysis, the authors compare the variations in these metrics between the faulty and fixed versions of the code units under test. Their findings indicate that no causal relationship exists between the buffer overflow fixes and variations in the studied static metrics, suggesting that these metrics alone are insufficient for detecting buffer overflow vulnerabilities.

Shiraishi et al. [68] introduce the Toyota ITC test suite, a publicly accessible set of unit tests specifically designed to examine the efficacy of static analysis tools in identifying common errors in C and C++ programming, including buffer boundary violations. Furthermore, the authors benchmark three static analysis tools—CodeSonar, Code Prover, and Bug Finder—against the test suite. The results indicate that these tools achieved near-complete detection rates and maintained low false positive rates in identifying static and dynamic memory defects, under which buffer overruns and underruns fall. However, it is important to note that the three tested tools are commercial software. Arusoaie et al. [69] assess ten non-commercial static analysis tools using a slightly modified version of the Toyota ITC benchmark, revealing their inadequate performance. The tools achieved an average detection rate of 12.8% for dynamic memory defects and 30.9% for static memory defects, with corresponding false positive rates of 8.4% and 7.0% respectively. Frama-C [70] was the most effective at identifying dynamic memory defects, with a detection rate of 83% but a high false positive rate of 23%. For static memory defects, the Clang static analyzer performed the best with a detection rate of 82%, but it also came with a considerable false positive rate of 15%.

Some static tools are specifically designed to analyze program binaries for memory access violations, addressing situations where source code is unavailable or where binary-level analysis is essential, such as uncovering compiler-induced vulnerabilities [71, 72]. For instance, UAFDetector [63] is designed to detect use-after-free violations in binary code. Its analysis primarily relies on CFG construction, achieved through a hybrid approach combining static analysis to extract basic control flow with dynamic instrumentation driven by fuzzing to capture indirect jumps. To support scalability, a central design goal, the tool summarizes critical function behaviors, such as creating, using, and destroying dangling pointers, and replaces function calls with these summaries to reduce redundant computation during inter-procedural analysis. However, part of the effort to enable scalability involves compromising detection accuracy, with path-insensitive analysis being the primary cause of false positives, and incomplete pointer alias analysis and imprecise loop handling being among the main contributors to false negatives. As another example, cwe_checker [64] is an open-source framework that employs a diverse set of static analysis techniques to identify common CWE types in binaries, with a particular focus on ELF files. Specifically, it uses a combination of points-to and value-set analyses to detect buffer overflows (CWE-119). However, the approach has notable limitations [73]. The imprecision of the deployed pointer analysis is a well-recognized source of false positives. Moreover, the tool fails to capture the size of objects allocated in any called function except for standard memory allocators, due to its limited inter-procedural analysis scope. This limitation can lead to false negatives, as overflows associated with such objects go undetected. Furthermore, it cannot detect overflows in global memory, and for stack memory, it can only identify overflows that exceed the boundaries of the stack frame.

Several recent studies have explored the use of machine learning in static code analysis for detecting defects, with the majority concentrating on identifying semantic errors and vulnerabilities including those related to memory access operations [74]. Many of these studies have concluded that existing machine learning techniques fail to achieve effective results in this domain. For instance, in a series of empirical studies, Chappell et al. [75] seek to design independent machine learning models capable of effectively detecting C coding errors, mainly those related to accessing memory. They find that the models exhibit very low effectiveness unless trained on features extracted from static analysis, despite this contradicting their initial objective of maintaining independence from such features. Furthermore, the authors report that even when using these features for training, the models still are not able to generalize across different datasets. In a similar study, Zhao et al. [76] explore the potential of machine learning models to replace manually developed static analysis tools for identifying common programming errors, focusing on the detection of buffer overflows in C code as a

case study. They try supervised and unsupervised learning strategies to extract relevant features from source code or its LLVM intermediate representation, in combination with some existing classifiers. Their results show that the trained models performed very poorly on unseen real-world codebases, revealing the significant difficulty that these models encounter in learning to detect buffer overflows in C programs. Examining a different approach, Krishnan et al. [77] investigate the integration of machine learning techniques with static program analysis frameworks to facilitate the detection of security vulnerabilities. Their experiments, including those aimed at identifying memory access bugs in C programs, demonstrate that applying these approaches to static analysis is largely unsuccessful. The authors identify four primary limitations associated with using machine learning techniques : the difficulty in creating enough labeled data for training, the inability of machine learning models to fully comprehend the runtime behavior of code, misleading performance evaluations due to non-representative datasets, and the lack of explainability in results generated by these models compared to traditional program analysis techniques.

Although static analysis tools provide the advantage of full code coverage without requiring program execution, their effectiveness is constrained by inherent precision and scalability limitations. These challenges require static approaches to adopt approximations that often result in a trade-off between false positives and false negatives, limiting their practical reliability in detecting memory safety violations.

5.4.2 Dynamic analysis for detecting memory safety violations

Dynamic techniques for verifying memory access safety operate by capturing and analyzing memory-related operations that occur during a given program execution. In this context, a dynamic tool relies on runtime information to identify violations of memory access safety along the observed execution path. In particular, the essential runtime information to monitor are memory allocation, deallocation, and access points.

Many dynamic detectors also rely on certain forms of static analysis. This reliance typically stems from the need to perform compile-time analysis and code transformations to enable and optimize runtime monitoring [61, 62]. Limited code coverage is an inherent constraint of dynamic detection approaches, which makes them prone to false negatives [78]. This limitation arises from the fact that, during each run, dynamic tools analyze only the observed execution path, which is one of potentially many execution paths within a program. This emphasizes the significance of utilizing a diverse set of test harnesses when employing dynamic verification tools, as only the code paths exercised by these harnesses are analyzed. Despite these limitations, dynamic analysis techniques are more widely adopted in real-world

applications than static approaches, primarily due to their superior detection accuracy and scalability [79].

Azhari et al. [80] propose a low-overhead post-mortem framework for detecting memory leaks and reporting their root causes, tracing issues down to the allocation call stack. The presented method periodically gathers runtime statistics on allocated and deallocated memory blocks by intercepting calls to heap allocation and deallocation functions. The collected data are subsequently analyzed using a leak detection algorithm based on growth analysis, designed to minimize the likelihood of false positives.

Kroes et al. [62] introduce Delta Pointers, a buffer overflow protection mechanism based on pointer tainting. This approach encodes two pieces of metadata within the upper bits of 64-bit pointers : the distance from the current pointer to the upper bound of its corresponding memory object, known as delta tag, and a binary overflow flag stored in the most significant bit. The framework relies on compile-time instrumentation to manage delta tag encoding in original pointers and propagate it through pointer arithmetic. Specifically, the instrumentation module identifies pointers returned from memory allocation sites and encodes the delta tag into them. During pointer arithmetic, the delta tag is adjusted accordingly to reflect the operation, ensuring it maintains the updated distance to the upper bound. The overflow flag, initially set to zero, is implicitly set to one when a pointer arithmetic operation causes the pointer to exceed the upper bound. This occurs as the delta tag overflows into the most significant bit as a result of the arithmetic operation. Upon pointer dereference, the delta tag is masked out, while the overflow bit remains unchanged. If the overflow bit is set, the dereferenced pointer becomes non-canonical, causing the memory management unit to raise an error. Delta Pointers cannot be used to detect buffer underflows or any type of temporal violations.

AddressSanitizer (ASan) [61] is an open source dynamic tool designed to detect a range of common spatial and temporal violations in C and C++ programs when accessing heap, stack, and global objects. It has been part of LLVM starting from version 3.1, and GCC since version 4.8. Specifically, it is a runtime verification tool comprising a compile-time code instrumentation module and a runtime library. The tool uses a combination of shadow memory and red zones to detect spatial access violations, where each shadow byte encodes what bytes of the original application memory are addressable. As of this writing, its memory leak detection capability remains “experimental” according to its documentation, with a major unresolved issue specifically related to false negatives [81].

CETS [78] is a compiler-based approach to enforce temporal safety in C programs. At runtime, each pointer is associated with two pieces of metadata : a unique key and a lock address. A

pointer is considered valid as long as the key and the accosted lock stored at the lock address are identical. It terms of shadow space lookup, it uses a trie data structure with two levels to map each pointer to its corresponding key and lock address. Handling derived pointers is based on the assumption that the derived pointer points to the same memory allocation as the original pointer (Assumption of Spatial Safety). When the compiler encounters pointer arithmetic, it inserts code to propagate the metadata from the original pointer to the derived pointer. CEFT relies on compile-time analysis to track pointer propagation. In terms of detection scope, the proposed method is limited to only detecting accesses beyond the intended upper bound of a buffer.

ViK [79] is a pointer-based runtime protection against the violations of temporal memory safety. It assigns a random Object ID to each allocated object, embedding it in unused pointer bits and verifying its validity during dereferencing or deallocation. It optimizes performance by leveraging static analysis to skip checks on UAF-safe pointers and utilizing hardware-assisted techniques to reduce overhead. The pure-software implementation of the tool is reported to incur around 20% runtime overhead. ViK is not able to detect any kind of spatial memory safety violations.

In summary, existing dynamic verification tools are typically either very limited in detection scope or impose significant runtime overhead. The former restricts their ability to comprehensively detect memory safety violations, while the latter makes them impractical for deployment in test environments with limited resources.

5.5 AddressMonitor (AMon)

AMon is a runtime verification tool designed to detect violations of heap spatial and temporal safety. AMon leverages a combination of pointer tainting and compile-time analysis and transformations. During compilation, it detects heap pointer operands in memory access operations, creates an untainted variant for each heap pointer, and substitutes the original operands with their untainted counterparts. At runtime, it assigns a unique taint identifier to each allocated heap object, embeds this identifier into the unused bits of the returned pointer (tainted pointer), and maintains an object table to track spatial, temporal, and debugging information for user-allocated heap objects throughout program execution. It is important to note that the terms taint and tainted pointer are used in this thesis in the context of runtime verification [106, 107], and must not be confused with their interpretation in security research.

AMon consists of two main components : a compile-time analysis and transformation module

and a runtime library. The compile-time transformation performed by AMon demonstrates how static analysis can detect heap-derived pointers and enforce safe memory access patterns without modifying program semantics. By combining value-flow analysis to identify derived pointers, bitmasking to ensure safe dereferencing, and selective instrumentation for runtime validation, the pass addresses both spatial and temporal memory errors. The runtime library of AMon supports dynamic monitoring and analysis by constructing and managing the object table, intercepting standard C memory allocation and deallocation functions, and defining the program analysis mechanisms employed by AMon. We describe the details of the two components (the compile-time transformation and the runtime library) in Sections 5.6 and 5.7, respectively.

5.6 AMon : Compile-time transformation

The compile-time analysis and transformation module of AMon is designed to detect heap accesses in user code. The module consists of three main analysis phases. We have implemented a prototype of the compile-time transformation module of AMon as an LLVM pass ; however, the proposed ideas are not compiler-specific.

5.6.1 Identification of heap pointers

This phase (described by Algorithm 2) aims to identify all heap pointers within each analyzed function using a safe over-approximation. Within each function, the module begins by constructing an initial set of original heap pointers, including those returned from standard heap allocation functions. Additionally, function arguments of pointer type, pointers returned from function calls, and module-level global variables storing pointers are conservatively included. This is to ensure that any potential tainted pointers originating from these external sources are properly accounted for in the analysis.

Once the initial set is established, the module performs a static value-flow analysis to identify all derived pointers as well. The analysis tracks how heap pointers propagate through operations such as pointer arithmetic, type casting, and control-flow merges. Furthermore, it captures the more complex case of indirect propagation through memory accesses, where a tainted pointer is stored in memory and later retrieved.

To achieve such comprehensive pointer tracking, the analysis employs a worklist-based algorithm that iteratively processes all uses of heap-derived pointers. The worklist is initially populated with the original heap pointers identified at the beginning of this phase. Each derived pointer is recursively examined, and new pointers inferred through propagation are

added to the worklist. This iterative process continues until a fixed point is reached, ensuring that all pointers potentially referencing heap-allocated regions are accounted for, even in the presence of complex transformations that obscure their origins.

5.6.2 Replacement of Tainted Pointers with Their Untainted Variants

The pass traverses each instruction in a function to detect memory access operations. If a pointer operand used in these operations is part of the identified tainted set detected in the previous phase, the pass generates an untainted variant by masking out the taint bits. The original instruction is then modified to use the untainted pointer instead to avoid a segmentation fault at runtime. If the tainted pointer has already been untainted earlier within the same function, the previously generated untainted variant is reused to minimize redundant computations and optimize performance. The replacement operation is carefully placed to preserve SSA form, ensuring that no PHI nodes are directly modified. In cases where the tainted pointer originates from a defining instruction, the untainting operation is inserted immediately after that instruction. If the pointer involves complex control flow merges, the pass applies additional strategies to determine a safe insertion point while maintaining correctness.

5.6.3 Instrumentation of Memory Accesses for Runtime Verification

To enable runtime validation, the pass determines which memory accesses require monitoring and inserts instrumentation logic accordingly (Algorithm 3). For each monitored access, it captures the tainted pointer and access size and provides this information to the runtime verification logic. The instrumentation is only applied when necessary, avoiding redundant checks and reducing performance overhead. The pass distinguishes between read and write operations to invoke the appropriate runtime validation function. The size of the accessed memory is extracted using the data layout, ensuring accurate verification. When possible, multiple memory accesses within the same basic block are optimized to minimize redundant runtime checks while maintaining detection accuracy. This optimization is specifically safe to apply when there is no call instruction between two memory-related operations in a basic block accessing the same heap pointer.

5.7 AMon : Runtime library

The runtime library of AMon provides dynamic monitoring and analysis support. It builds and maintains the object table, provides interceptors for standard C memory allocation and

deallocation functions, and defines the program analysis mechanisms applied by AMon.

5.7.1 Object table

The object table is a data structure designed to maintain spatial, temporal, and debugging information on user-allocated heap objects throughout program execution. Each entry in this table represents an object allocated through our pointer tainting framework, with the taint value serving as the index of the corresponding object in the table, which accelerates queries. When a tainted pointer is dereferenced, the runtime analysis mechanisms verify the temporal and spatial safety of the memory access by retrieving relevant details from the object table, as explained in detail in 5.7.3. Specifically, AMon maintains the following information for each allocated object :

1. Base address : It stores the base address of the allocated memory region.
2. Size : It represents the size of the allocated memory region. In combination with the base address, it determines the intended boundaries of the allocated object, which is used to verify the spatial safety of a memory access, as discussed in 5.7.3.
3. Temporal state : It encodes the latest temporal state of the object (active, freed, realloc_freed). AMon actively updates the state of an object upon related program events.
4. Call stack : the call stack at the point of the latest temporal state update. This information will be used by the error report generation component of AMon.

The object table is the only source of data memory overhead imposed by the detection logic of AMon, and this overhead is very minimal. On our 64-bit test system with 16 unused address bits, the entire object table consumes approximately 3 megabytes of memory. The internal structure of the object table utilized by AMon is illustrated in Fig 5.1. In this figure, a hypothetical memory allocation of size 4, with a base address of 0x00005b49425332a0, results in the creation of a new entry in the table. The unique taint value assigned to this newly allocated heap object is 0x0001.

5.7.2 Intercepting standard C library functions

Part of the runtime library is dedicated to implementing interceptors for standard C library functions. The proposed interceptors can be categorized into two main groups. The first group are interceptors for the standard C heap allocation functions. The interceptors of this family are designed to add a unique taint value to the unused bits of the pointer they return, in our environment, the two most significant bytes of the pointer. The second group are other

object table

index	base address	size	temporal status	stack trace
0x0001	0x00005b49425332a0	4	ALLOCATED	#0 foo+0x19

FIGURE 5.1 Internal layout of the object table used by AMon to record metadata for allocated heap objects

standard C functions that may receive a tainted pointer. For these functions, the interceptors untaint the memory operand(s), check the underlying memory access for spatial or temporal safety violations, and then perform the original functionality of the intercepted function. In a Linux-based ecosystem, the runtime library can be pre-loaded using the `LD_PRELOAD` environment variable.

5.7.3 Program analysis mechanisms

Part of the runtime library implements various on-the-fly program analysis mechanisms employed by AMon to detect violations of temporal and spatial memory access safety, as well as a broader range of unsafe memory-related programming practices. These analyses are invoked through a combination of code instrumentation and specific monitoring functionalities embedded in our wrapper implementations for standard C library functions. In what follows, we provide a detailed discussion of each analysis mechanism, illustrating how AMon detects different types of memory issues in dedicated sections.

Out-of-bound memory accesses

AMon detects both heap underflow and overflow spatial access violations. Specifically, it monitors accesses to user-allocated heap objects in two contexts : within user code itself and within calls to standard C library functions that access heap memory via a pointer passed as an argument. To verify the spatial safety of a memory access, AMon requires two pieces of information pertaining to it : the base address and the size of the accessed memory region. For user code, this information is obtained through compile-time instrumentation by analyzing instructions that perform memory access, inserting specialized code to extract the base address and size of the access, and adding a call to the runtime library to deliver this information to the detection logic. For calls to standard C functions, it is collected as part of the wrapper implementations and provided to the runtime library, as explained in 5.7.2. With this information available for each heap access to be monitored, AMon extracts the

taint value embedded in the accessed base address, uses it to retrieve the intended bounds of the corresponding object from the object table, and verifies whether the accessed memory region lies entirely within the intended boundaries of the allocated object. If the accessed memory region is not fully contained within the intended boundaries, AMon generates a comprehensive report on the out-of-bound memory access. This report includes the exact boundaries of both the accessed region and the corresponding allocated object, the call stack at the point of the violating access, and the call stack at the point where the corresponding object was allocated.

Use-after-frees

AMon is able to identify use-after-free violations on user-allocated heap objects, detecting whether a dynamically allocated memory region is accessed after it has been deallocated, either in user code or as a result of a call to a standard C library function. To perform this analysis, AMon processes the accessed base address, which is obtained and delivered to the runtime detection logic using the same mechanisms shared with the out-of-bound memory access detection logic, namely, specialized code inserted by compile-time instrumentation for user code and code embedded in the wrapper implementations for standard C library functions. AMon extracts the taint value embedded in the accessed base address and uses it to query the current state of the corresponding object from the object table. If the pointer is marked as freed, AMon identifies a use-after-free violation and generates a detailed report. This report includes the involved pointer, the call stack at the point of the violating access, and the call stack at the point where the corresponding object was initially freed.

Specifically, in conjunction with its wrapper implementation of the `realloc` function, AMon can detect complex use-after-free cases potentially arising from memory reallocation. One such scenario occurs when a `realloc` call allocates a new pointer and frees the original one, such as when reallocating to a larger memory size and the existing memory block cannot be expanded in place. In the GNU C Library (glibc) implementation, another case arises when the requested reallocation size is zero, and the input pointer is not `NULL`, making the call equivalent to freeing the input pointer. Beyond detecting these cases, AMon explicitly reports the specific context of the deallocation event, distinguishing whether it occurred through a direct `free` call, during memory relocation caused by a `realloc` call, or more specifically, as a result of a zero-size `realloc` operation. This capability is achieved by AMon through defining three distinct event types to accurately represent these pointer states within its object table, rather than relying on a single generic type to represent a freed pointer. This design choice enhances the precision of error reporting by offering detailed insights into the

nature of deallocation events, thereby simplifying debugging and reducing the user effort required to identify the cause of pointer deallocations.

Double-frees

AMon incorporates built-in double-free analysis within its wrapper implementation of the `free` function. When an attempt is made to free a tainted pointer, its current temporal state is retrieved from the object table. If the pointer is already marked as freed, the tool reports a double-free violation. For each detected instance, the tool provides a comprehensive report detailing the pointer involved in the violation, the call stack at the point of the current free attempt, and the call stack recorded at the point where the pointer was deallocated. This methodology extends to addressing edge cases specifically associated with memory reallocation, utilizing the same principles described for handling related use-after-free issues. For instance, AMon can detect situations where a pointer, deallocated as part of a `realloc` operation, is subsequently attempted to be freed via a direct `free` call.

Memory leaks

AMon is capable of detecting all heap objects that remain allocated by the time the program terminates, including both unreachable and reachable objects. Unreachable objects contribute to memory leaks, as they represent memory that has not been freed and can no longer be accessed after a certain point during program execution due to the loss of references. Reachable objects, although still accessible, highlight potential inefficient memory management practices that can primarily result in memory exhaustion, a critical concern in systems with limited resources, such as embedded systems. Additionally, such cases can degrade performance by increasing the need for paging, which slows down memory access [80]. Therefore, it is preferable for a memory leak detector to be able to identify both of these causes, ensuring a complete evaluation of dynamic memory mismanagement. The memory leak detection logic of AMon is invoked at the end of the process via the deconstructor of the runtime library. It iterates over each entry in the object table, examines the state of the allocated objects, and detects those that are still active, which indicates that the programmer failed to free the object either because it became unreachable or due to not adhering to recommended memory management practices. For each identified instance, AMon generates a detailed report that includes the base address of the leaked memory region, the size of the allocated memory, and the call stack at the point of memory allocation.

Other unsafe practices

In addition to detecting spatial and temporal heap access violations, AMon can also identify two other common unsafe programming practices related to heap access, which are passing a non-base pointer to `realloc` or `free`, and potential use-after-free conditions stemming from memory reallocation.

Use of a non-base pointer The pointer argument passed to a `realloc` function or a `free` function must be a base pointer previously returned by a function from the `malloc` family, unless it is a `NULL` pointer. When the pointer argument is tainted, AMon verifies this condition by checking whether the untainted version of the pointer matches the base address associated with its taint value in the object table. By employing this verification mechanism, AMon is able to explicitly detect such violations and, therefore, provide a precise and clear report that directly pinpoints the actual issue in user code in the event of a violation. In contrast, ASan identifies calls to `realloc` with non-base pointers indirectly, as a side effect arising from its interceptor implementation. Specifically, the `realloc` interceptor in ASan allocates a new memory region of the requested size, copies the contents from the region referenced by the pointer argument to the newly allocated one, and then frees the pointer. If the pointer argument is a non-base pointer, ASan raises an error during the free operation, reporting it as an attempt to free a non-malloced pointer. However, this report hides the true source of the issue, making it non-intuitive for users to diagnose the underlying problem. Consequently, users must manually examine the reported call stack to trace the issue back to a `realloc` call, and even then, understanding the root cause requires prior knowledge of how ASan intercepts `realloc`.

Potential use-after-frees In 5.7.3, we discussed how AMon identifies use-after-free scenarios resulting from memory reallocation when a new pointer is returned. However, the case where the reallocation returns the same pointer also requires careful analysis while this may not seem as evident. In this case, although the old pointer is not freed during reallocation in the observed execution, its continued use after reallocation may still represent a potential use-after-free condition, as it likely indicates insufficient verification by the user to update all variables storing the old pointer, which can result in an actual use-after-free violation in future executions. ASan detects this condition through its wrapper implementation of `realloc`, which is designed to consistently allocate a new memory chunk and free the old pointer, as described earlier. Consequently, ASan reports a use-after-free error on any dereference of the old pointer following the reallocation. However, this error report may appear

counterintuitive to the user in the scenario described above, as the old pointer might appear valid in native executions where ASan is not used. In contrast, while AMon also reports this case, it generates a distinct report type compared to cases where the old pointer is freed during reallocation. In this dedicated report, AMon notifies the user that although the dereferenced pointer has not been freed, it has undergone a reallocation process, and therefore, its subsequent dereferences might expose the program to a use-after-free violation in future executions. This functionality is achieved through specific internal checks integrated into its wrapper implementation of `realloc`, where, even if the reallocation process returns the same pointer, AMon assigns a new taint value to the returned pointer and updates the state of the old pointer in the object table to a dedicated value (`REALLOC_SAME`) specifically representing this event. Through this analysis, AMon enables users to detect such potential use-after-free cases within the code under test, even if these violations do not manifest during the observed execution.

5.8 Evaluation

5.8.1 Performance analysis

In the first stage of our evaluation study, we assess the runtime efficiency of AMon using a set of five SPEC CPU 2017 benchmarks, comparing its performance with ASan, a widely used memory error detector, and EffectiveSan, a state-of-the-art academic tool based on pointer metadata. All evaluations are conducted on a desktop system, equipped with a 12th Gen Intel i7-12700 CPU and 16 GB of RAM, running Ubuntu 24.04 LTS, with Linux kernel version 6.8.0. All benchmarks are executed with the reference input data provided by the suite. The findings of this assessment are summarized in Table 5.1. In this table, the “Native” column represents the execution time and memory consumption - measured in terms of maximum resident set size (MRSS) - of each benchmark when compiled without any instrumentation. The execution time and memory consumption of running each benchmark, with ASan, EffectiveSan, and AMon enabled, are reported in distinct columns. Each benchmark was executed ten times, and the results presented in the table reflect the average measurements collected over these runs.

Compared to ASan, AMon exhibits notably lower memory overheads but generally imposes a slightly larger execution time overhead. As expected, AMon incurs minimal memory overhead as the only source of data memory overhead in its implementation is the object table. In contrast, ASan causes a considerable memory overhead, which stems from its use of shadow memory and redzones, utilized to encode the accessibility of program memory and the

boundaries of allocated objects, respectively.

AMon preserves memory usage close to native in most cases, whereas ASan typically doubles or even triples memory use. In embedded environments, memory resources are often extremely constrained, making the large overhead from ASan unfeasible. By contrast, AMon is particularly well-suited for deployment in these resource-limited environments.

Compared to EffectiveSan, AMon consumes slightly more memory in some benchmarks but generally achieves faster execution times. While EffectiveSan instrumentation can introduce longer slowdowns, AMon's overhead remains comparatively moderate. This balance makes AMon more suitable for environments where both memory availability and runtime performance are critical. It is also important to note that AMon can detect a wider spectrum of memory violation types than EffectiveSan, particularly encompassing a broader range of temporal safety violations.

TABLE 5.1 Performance Evaluation on SPEC CPU 2017 Benchmarks

Benchmark	**Native**		**ASan**		**EffectiveSan**		**AMon**	
	Time (s)	MRSS (MB)	Time (s)	MRSS (MB)	Time (s)	MRSS (MB)	Time (s)	MRSS (MB)
505.mcf	232.4	623.7	285.3 (22.8%)	919.8 (47.5%)	508.3 (118.8%)	624.8 (0.2%)	370.6 (59.4%)	660.0 (5.8%)
519.lbm	123.5	420.3	128.3 (3.9%)	476.8 (13.4%)	198.1 (60.4%)	420.8 (0.1%)	221.0 (78.9%)	420.8 (0.1%)
544.nab	247.6	149.7	290.4 (17.3%)	540.8 (261.2%)	332.3 (34.2%)	163.4 (9.2%)	257.5 (3.9%)	164.4 (9.8%)
538.imagick	256.9	293.0	423.8 (64.9%)	823.8 (181.2%)	561.9 (118.7%)	296.9 (1.3%)	535.2 (108.3%)	296.5 (1.2%)
557.xz	245.6	794.1	327.3 (33.3%)	1051.0 (32.4%)	384.3 (56.5%)	794.3 (≈0%)	368.8 (50.1%)	794.7 (≈0%)
Average Overhead			28.5%	107.2%	77.8%	2.2%	60.1%	3.4%

Each benchmark was executed ten times, and the presented results reflect the average measurements collected over these runs.

5.8.2 Accuracy analysis

To evaluate the detection accuracy of AMon, we tested it on a variety of micro benchmarks. Specifically, we conducted experiments on a set of benchmarks that we implemented, as well as 20 benchmarks selected from the Juliet test suite. ASan was chosen as the baseline for comparison because it is a state-of-the-art runtime verification tool known for its high detection accuracy. Our results indicate that, for the tested benchmarks, AMon achieves the same level of detection as ASan. Specifically, for the selected Juliet benchmarks, neither AMon nor ASan produced false positives or false negatives, demonstrating the reliability of both tools in these cases.

5.8.3 Integration of AMon for a special-purpose industry use case

Ericsson attempted to use AMon in their internal, proprietary Baseband processing system, comprised of specialized multicore CPUs and hardware acceleration engines. This system

is programmable in C, and can suffer from memory management faults that are detectable from a tool such as AMon. Implementing this type of instrumentation is challenging in these embedded system environments because of specialized memory systems, small memory size and limited memory in certain contexts such as startup and event handling memory conditions.

The operating environment for these Baseband processing systems (custom hardware and software) has been around and evolved over at least 25 years. Specialized hardware has been developed and integrated to accelerate as many operations as possible, and hardware and software are tightly integrated. Memory is specialized for this unique systems design, and memory addresses are used by both hardware accelerators and specialized processors without modifications for simplicity of design, development and maintenance. There exist many control and data paths within this design that need to be considered when integrating a tool such as AMon. Available address bits for the "taint" portion of the address are limited, and vary from system to system. This was an important consideration for integration into Ericsson systems.

Integration problems included the following :

— Limited memory conditions in startup code and in event handling routines. This can be mitigated by disabling instrumentation in those contexts using compiler attributes. Another way to mitigate this problem is check stack size at instrumentation entry points, and avoid instrumentation operations dynamically under low memory conditions.

— "Tainted" addresses can be used to program hardware directly since the compiler instrumentation pass has no way to know in a systematic way if hardware is being programmed or not. This can be mitigated in a number of ways including : 1) If hardware programming APIs are well designed, instrumentation can be used to mask addresses passed to those APIs. Unfortunately this is not true for a system that's been in use with many changes over the years. 2) The kernel can be modified to mask addresses before programing hardware. This is possible, but long and labor intensive. 3) Hardware could be modified to be tolerant of these operations, but that requires careful design consideration and time to redesign and fabrication of a new system.

— Specialized hardware can drive allocations and frees of memory, then the compiler is not able to instrument those cases. A mitigation is to write custom instrumentation for those operations.

— The Specialized software consists of both C language and assembly code, whereas the compiler instrumentation is for C code only. A mitigation would be to add hand coded

instrumentation and control through build flags.

Some interesting findings from Ericsson's integration work include :

— Initially we found allocated memory in test cases being used without initialization. This AMon approach is an efficient and easy way to accomplish this.
— In those cases, the check was initially repeatedly reporting on the same allocated memory block. This led to an overwhelming number of error reports that was due to the same root cause. An easy improvement is to detect and report unique findings per memory block so the user is not overwhelmed with findings.
— Some very narrow corner cases of kernel code made use of intimate knowledge of the allocators' inner workings, and accessed memory just out of allocation bounds (because this was known to be available). This is by definition a true positive, but some specialized management of these types of cases would help users trust the tool.

Ericsson has made several improvements to AMon in the integration process, including : 1) Clang front end option for invoking AMon. 2) Attributes for disabling AMon for specific functions. 3) Brought the AMon modifications up to latest tip of tree LLVM/Clang version.

5.9 Conclusion

Although the explicit control over memory in the C programming language is fundamental to its adoption in performance-critical software development, it poses significant verification challenges. The absence of built-in mechanisms to verify the temporal and spatial safety of memory accesses necessitates developer-enforced validation, which is a recognized source of critical errors and security vulnerabilities in C programs. While dedicated compiler extensions have significantly improved stack protection, enforcing heap safety remains a challenge, driving interest in tools that detect heap access violations. Dynamic verification techniques offer higher detection accuracy and scalability than static approaches but often introduce excessive overhead, making them impractical for resource-constrained environments such as embedded systems. Additionally, many existing tools have a limited detection scope. To address these issues, this paper presents AddressMonitor (AMon), a low-overhead runtime verification tool for detecting a wide range of heap access violations in C programs. AMon combines pointer tainting with compile-time analysis and transformations. It identifies heap pointer operands, generates untainted variants, and replaces the original pointers accordingly. At runtime, AMon assigns a unique taint value to each allocated object, embeds it into the pointer, and maintains an object table to track spatial, temporal, and debugging information throughout execution. Our experimental results show that AMon detects a broad range

of memory access violations without false positives or false negatives, while maintaining a more balanced performance overhead compared to state-of-the-art baselines such as ASan and EffectiveSan.

Algorithm 2 Heap Pointer Derivation Analysis

Require: LLVM Function F
Ensure: Set of all derived heap pointers *DerivedPtrs*

```
1: function ComputeDerivedPointers(F)
2:     Initialize
3:     HeapPtrs ← {globals, allocs, args}                    ▷ Phase 1
4:     Worklist ← HeapPtrs
5:     DerivedPtrs ← HeapPtrs
6:     TaintedMem ← ∅
7:     while Worklist ≠ ∅ do                                 ▷ Phase 2
8:         v ← Worklist.pop()
9:         for all users u of v in F do
10:            if u is store with value v then
11:                addr ← store address operand
12:                if addr ∉ TaintedMem then
13:                    TaintedMem ← addr
14:                    Add loads of addr to Worklist
15:                end if
16:            else if u manipulates pointer semantics then
17:                Mark u as derived if :
   — Pointer arithmetic (GEP)
   — Type casts (bitcast/ptrtoint)
   — Integer operations with derived operands
   — PHI/Select with derived inputs
18:                if u ∉ DerivedPtrs then
19:                    DerivedPtrs ← u
20:                    Worklist ← u
21:                end if
22:            end if
23:        end for
24:    end while
25:    return DerivedPtrs
26: end function
```

Algorithm 3 Untainting Tainted Pointers and Instrumenting Their Uses

Require: I : the LLVM `Instruction` to transform
Require: *Ptr* : the pointer operand of I
Require: *DL* : the LLVM `DataLayout` for type information
Require: *UntaintedPtrMap* : a map storing previously created untainted pointers
Require: *AMonRead*, *AMonWrite* : instrumentation functions for memory accesses
Ensure: I is updated to use an untainted pointer, and optionally has instrumentation checks inserted

1: **function** UNTAINT(I, Ptr, DL)
2: **if** SHOULDSKIP(I) **then return**
▷ (A) Extract upper bits of pointer (the taint index)
3: *PtrAsInt* ← Ptr cast to 64-bit integer
4: *Index* ← *PtrAsInt* ≫ 48 ▷ shift to retrieve upper 16 bits
▷ (B) If *Index* is statically zero, do nothing
5: **if** *Index* is a compile-time constant AND zero **then**
6: **return**
7: **end if**
▷ (C) Emit a runtime check : if upper bits are zero, skip instrumentation
8: *IsZero* ← (*Index* = 0)
9: *ContBB* ← splitBasicBlock(*I*) ▷ moves I to a new block
10: *CheckStatusBB* ← createNewBlock()
11: replaceTerminator(*CurrentBB*, CondBr(*IsZero*, *ContBB*, *CheckStatusBB*))
▷ (D) In *CheckStatusBB*, check whether pointer is `ACTIVE`
12: *ObjTblPtr* ← loadObjectTable(*CheckStatusBB*)
13: *Status* ← ObjTblPtr[*Index*].*status*
14: *IsActive* ← (*Status* = `ACTIVE`)
▷ (E) If not active, call *AMonRead*/*AMonWrite* in `AMonReportBB`
15: *AmonReportBB* ← createNewBlock()
16: insertCallIfNeeded(*AMonReportBB*, *AMonRead*/*AMonWrite*, *Ptr*)
17: replaceTerminator(*CheckStatusBB*, CondBr(*IsActive*, *ContBB*, *AMonReportBB*))
▷ (F) Finally, ensure I uses an untainted pointer
18: **if** *Ptr* ∈ *UntaintedPtrMap* **then**
19: *I.operand* ← *UntaintedPtrMap*[*Ptr*]
20: **else**
21: *UntaintedPtr* ← ptrmask(*Ptr*, *UntagMask*) ▷ intrinsic to clear high bits
22: *UntaintedPtrMap*[*Ptr*] ← *UntaintedPtr*
23: *I.operand* ← *UntaintedPtr*
24: **end if**
25: **end function**

CHAPITRE 6 ARTICLE 3 : IDENTIFYING AND MITIGATING IMPLEMENTATION-INDUCED DATA RACE DETECTION BLIND SPOTS IN THREADSANITIZER V3

Auteurs : Farzam Dorostkar, Michel Dagenais, Heng Li, Vladislav Aranov et Ankush Tyagi.

Soumis à Concurrency and Computation : Practice and Experience

Date : 21 Octobre 2025

6.1 Abstract

Detecting data races remains a major challenge in debugging multithreaded programs. While much of the existing work focuses on improving the theoretical aspects of race detection algorithms, such as happens-before and lockset models, the correctness of practical implementations has received comparatively little attention. In this paper, we examine the latest version of ThreadSanitizer, a production-grade race detector integrated into both LLVM and GCC, and uncover two previously undocumented detection blind spots stemming from implementation-level decisions. The first blind spot arises when all shadow slots are occupied and a new access overwrites an existing entry through a random replacement policy, risking the loss of critical access information. To address this, we introduce three mitigation strategies guided by different heuristics : (i) an access-size-aware policy that prioritizes overwriting entries representing narrower memory accesses, (ii) a thread-diversity-aware policy that evicts entries from threads occupying multiple slots, and (iii) an access-recency-aware policy that prioritizes overwriting entries corresponding to older accesses, applying lightweight epoch tracking. The second blind spot results from the non-atomic nature of the end-to-end race checking and shadow update process, potentially allowing interleaved execution to obscure data races. We formally characterize the condition under which this blind spot occurs. Further, to mitigate it, we introduce a mutex-based synchronization mechanism that enforces atomic execution of the whole check-and-update procedure. Our evaluation, combining synthetic stress tests with benchmarks from the PARSEC suite, demonstrates that all three shadow overwriting strategies improve detection coverage relative to the default policy, with the size-aware heuristic yielding the largest benefits, while the mutex-based mitigation further improves race detection accuracy at the cost of increased runtime overhead.

6.2 Introduction

Although shared memory facilitates efficient thread communication, it also makes multithreaded programs vulnerable to *data races* [108]. A data race occurs when two or more threads access the same memory region in the absence of synchronization or mutual exclusion, with at least one of the accesses being a write operation [109, 110].

Identifying data races is one of the most challenging aspects of debugging multithreaded programs due to the non-deterministic interleavings and the need for high tracking precision [111, 112]. Over the years, numerous dynamic detection algorithms have been proposed, such as happens-before-based approaches (e.g., FastTrack [112], iFT [113]) and lockset-based approaches (e.g., Eraser [114, 115]). Hybrid detectors combining both logics such as Hybrid [109] and ThreadSanitizer [35] have demonstrated the practical benefits of these algorithmic innovations.

However, prior research has predominantly focused on algorithmic improvements—reducing vector-clock size, refining check policies, or employing predictive analysis [113,115–120]. These efforts improved theoretical guarantees and performance, yet they do not consider behavioral effects emerging from real-world implementations. Therefore, the impact of low-level implementation choices on detection accuracy is not covered in previous studies.

In this paper, we present a detailed implementation-level analysis of two previously not studied detection blind spots in ThreadSanitizer v3—a widely used, production-grade data race detector embedded in both LLVM and GCC. The first blind spot relates to shadow value eviction in the event of all slots being occupied. We show that the default random replacement policy applied by ThreadSanitizer can cause critical race information to be overwritten prematurely. To address this, we propose three alternative overwrite strategies guided by different heuristics : a size-aware policy that prioritizes evicting entries corresponding to narrower memory accesses, a thread-diversity-aware policy that selects entries from threads already dominating multiple slots, and a recency-aware policy that favors overwriting older accesses identified through lightweight epoch tracking. Our evaluation using both synthetic stress tests and PARSEC benchmarks shows that all three approaches reduce coverage loss compared to the default policy, with the size-aware and recency-aware policies providing the most substantial gains.

The second blind spot arises from the non-atomicity of the complete check-and-update sequence. Although individual shadow reads and writes are atomic, interleaving during the composite operation can lead to missed race reports. We formalize the condition under which such interleaving may lead to suppressed race detection. We also propose a mutex-based

mechanism that ensures atomic execution of the entire check-and-update procedure as a mitigation, preventing interleaved race checks by multiple threads on the same shadow region.

Together, our findings emphasize that ensuring the correctness of race detectors requires assessing both the algorithmic and implementation details. The remainder of this paper is organized as follows. In Section 6.3, we present a technical review of the key implementation aspects of ThreadSanitizer across its three major versions, with particular emphasis on the most recent version of the tool. In Section 6.4, we describe the first identified detection blind spot in ThreadSanitizer and introduce a mitigation strategy that improves its overwrite policy. In Section 6.5, we describe and formally characterize a second blind spot arising from the non-atomicity of the overall race checking procedure. In Section 6.6, we evaluate the effectiveness of the proposed mitigations using both PARSEC benchmarks and synthetic stress tests, highlighting improvements in detection coverage. Section 6.7 concludes this paper, summarizing our most important findings.

6.3 ThreadSanitizer

ThreadSanitizer (TSan) is a state-of-the-art open-source dynamic data race detector that currently supports multiple 64-bit architectures. The tool consists of two primary components : an instrumentation module and a runtime library. The instrumentation module annotates memory accesses within user code with calls to the runtime library, where each call provides runtime information about the instrumented memory access and initiates a check for potential data races involving that access. In addition to implementing race detection logic, the runtime library intercepts standard library functions that fall into two main categories : those associated with managing threads, coordinating mutual exclusion, and enforcing synchronization ; and those involved in performing memory access operations. The former enables the tool to capture timing constraints among threads, an aspect essential for reasoning about data races. The latter allows it to monitor shared memory accesses not explicitly present in user code but occurring within standard library functions. To date, TSan has evolved through three major versions, marked by notable differences in their handling of code instrumentation and/or data race analysis framework.

The first version of TSan (TSan v1) employed run-time instrumentation, with an initial implementation based on PIN [33] followed by a transition to Valgrind [34] that offered improved performance [35]. In terms of the underlying race detection algorithm, TSan v1 offered two analysis modes : a default hybrid mode combining happens-before and lockset techniques, and an alternative pure happens-before mode. While reporting fewer false positives, pure happens-before race analysis is generally less predictable, as its ability to detect certain real

data races can depend on the specific execution order of conflicting accesses determined non-deterministically by the scheduler [35, 36].

TSan v1 suffered from substantial runtime overhead, with reported slowdowns ranging from 20× to 300× [37]. This overhead was largely attributed to its reliance on Valgrind for instrumentation, which also limited portability to platforms supported by Valgrind [38]. To address these limitations, the second version of the tool (TSan v2) adopted compile-time instrumentation integrated into the build process [38]. TSan v2 initially retained the race detection logic of its predecessor, but later transitioned to a pure happens-before mode as its sole analysis approach, in part due to the tendency of lockset analysis to produce false positives in lock-free code [39]. In its re-designed runtime library, TSan v2 employed a vector clock mechanism to assess the presence of a happens-before relation between conflicting memory accesses. It relied on shadow memory to retain information about prior memory accesses, where each eight-byte block of application memory was directly mapped to a group of typically four *shadow cells*. Each shadow cell, eight bytes in size, encoded a memory access and consisted of four components : 16 bits storing the thread ID of the accessing thread, 42 bits capturing a snapshot of its vector clock at the time of access, 5 bits encoding the position of the accessed bytes, and 1 bit indicating whether the access was a write operation [37].

6.3.1 ThreadSanitizer v3

The third and most recent version of TSan (TSan v3) preserves the compile-time instrumentation strategy introduced in TSan v2 and is actively maintained within both the LLVM and GCC compilers [41, 42]. Its instrumentation module is implemented as dedicated compiler passes, operating on LLVM IR and GIMPLE, respectively—the architecture-independent intermediate representations specific to each compiler.

Similar to its predecessor, TSan v3 employs a pure happens-before analysis as its only race detection logic. However, it introduces a major redesign of the runtime library, primarily aimed at reducing the runtime overhead and memory footprint of the tool [43]. Among the modifications, TSan v3 reduces the size of each shadow value by half compared to TSan v2, yielding a twofold decrease in the shadow-to-application memory ratio. Figure 6.1 illustrates the default mapping from application memory to shadow memory on `x86_64`, along with the internal layout of a shadow value in TSan v3. As depicted, each 8-byte aligned region of application memory is directly mapped to four shadow values by default, where each shadow value occupies four bytes and encodes five components that characterize a memory access : an 8-bit mask indicating which bytes in the associated application region were accessed, 8 bits recording the identifier of the accessing thread, 14 bits storing its own logical clock at the

time of access, 1 bit specifying whether the access was a write, and 1 bit indicating whether it was an atomic operation.

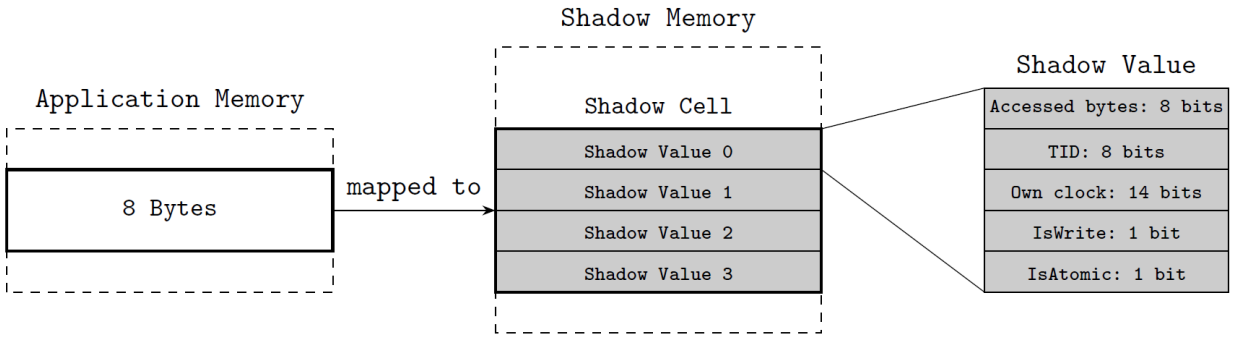

FIGURE 6.1 Default shadow memory mapping on `x86_64`, and the internal layout of a shadow value in ThreadSanitizer v3

Algorithm 4, extracted directly from the implementation of the TSan v3 runtime library in LLVM 21, outlines the core procedure used to verify a memory access against shadow memory for a potential data race utilizing the compact shadow value structure described above. The algorithm iterates over the shadow values associated with the accessed memory region to determine whether the current access constitutes a data race with any previously recorded access to that region. For each entry, it applies a series of conditions to quickly rule out non-conflicting cases. These include disjoint byte-level access ranges, identical thread identifiers, and cases where both operations are reads or are marked as atomic. If the previous access was performed by the same thread and matches the current access in type and granularity, no conflict is reported, and the algorithm proceeds to the next entry.

When the prior access was issued by a different thread and the byte ranges overlap, the algorithm checks whether the happens-before relation is violated by comparing the logical clock of the current thread against the epoch stored in the shadow value. If the recorded access is not observed by the current thread, a data race is reported. Otherwise, the algorithm continues scanning the remaining entries. If no race is detected after examining all relevant shadow values, the access is deemed safe with respect to previously recorded accesses.

6.4 First Blind Spot : Overwriting Shadow Values

While Algorithm 4 is designed to maintain a bounded representation of memory access history using a fixed number of shadow values per memory region, the act of overwriting an existing shadow value introduces a potential source of imprecision that can lead to data race detection

blind spots. Specifically, the algorithm writes the current memory access into shadow memory in three distinct locations, each with different implications for correctness and precision.

The first case occurs at line 6, where an empty slot is encountered during the initial scan. If the current access is not marked as check-only and a vacant entry is available, the algorithm stores the access directly. This action is safe and race-preserving, as no information is lost—no previously recorded access is being replaced. It simply extends the tracking capacity to accommodate a new access.

The second write occurs at line 15, in a special-case optimization where the current access and an existing one originate from the same thread, have the same accessed byte range, and the current access is of equal or stronger access type (e.g., a write following a read). In this case, the shadow value is overwritten to reflect the more recent access. Since data races are only defined across threads, and both accesses are issued by the same thread, replacing the old value with the current one does not affect the soundness of the algorithm. It may, however, lose temporal information about earlier accesses from the same thread, but this is irrelevant to race detection under the happens-before model.

The third and critical write occurs at line 33, where none of the earlier entries have been deemed conflicting or available for update, and no race has been reported. If the current access has not been stored up to this point, the algorithm randomly selects an existing shadow value and replaces it with the current access. This strategy is essential for bounding memory usage, as it maintains a fixed number of slots per memory region. However, it introduces a blind spot : the randomly replaced shadow value may contain metadata for a prior access that would have been relevant for detecting a race with a future access. Once this metadata is lost, the algorithm can no longer verify whether a later access is racing with the evicted one. In such cases, the race remains undetected, not because the logic is incorrect, but because the prior access is no longer represented in shadow memory.

This limitation highlights a core trade-off in the design of TSan v3 : the use of compact shadow memory enables scalable instrumentation with bounded overhead, but it necessarily limits the temporal tracking window of prior accesses. The replacement strategy at line 33 is particularly susceptible to missing races that involve accesses overwritten due to the lack of available slots. While this design choice is practical for performance, it influences the completeness of race detection.

6.4.1 Mitigation : better overwriting policy

To address the blind spot introduced by overwriting shadow values—particularly at line 33 of Algorithm 4—we propose three alternatives to the shadow value replacement strategy, aimed at minimizing the risk of evicting metadata that is critical for detecting future races. In the current design of TSan v3, when none of the existing shadow values are empty or eligible for update, the algorithm selects one of the slots randomly and replaces its contents with the current access. While this ensures predictable memory overhead, it disregards the relative importance of the recorded accesses. As a result, the algorithm may inadvertently discard metadata for accesses that are more likely to conflict with future operations, thereby introducing a potential blind spot in the race detection logic.

To mitigate this issue, we introduce three prioritization-based overwrite strategies that replace the randomness of the existing design with heuristically guided selection mechanisms.

Access-size-aware eviction policy

The key insight behind this approach is to prefer evicting the shadow value corresponding to the access involving the fewest number of bytes. This decision is based on the observation that smaller accesses have a lower probability of overlapping with future accesses to the same memory region, and therefore are less likely to be involved in future data races.

Formally, let each shadow value track the byte-level mask corresponding to its access footprint. When a replacement is necessary, we inspect all current shadow values in the cell and select the one with the smallest number of bits set in its byte mask—that is, the access with the narrowest span. In the case of a tie, a secondary strategy (e.g., random selection among equally minimal accesses) can be applied. This selection criterion effectively reduces the likelihood that the overwritten metadata would have matched the byte range of a subsequent conflicting access.

By introducing this size-aware overwrite policy, the modified algorithm enhances the retention of shadow values that are more likely to be relevant for future race detection, particularly those representing larger or full-region accesses. This simple yet effective change improves the temporal robustness of the tracking mechanism without increasing memory usage. As a result, the proposed mitigation reduces the probability of missing races that would otherwise go undetected due to indiscriminate eviction, thereby strengthening the completeness of the detector under bounded memory constraints.

Thread-diversity-aware eviction policy

As a complementary enhancement to the size-aware approach, we propose a second strategy that incorporates thread-level diversity into the replacement decision. The primary intuition behind this policy is to maximize the variety of threads represented in the shadow memory, thereby preserving information that is most useful for identifying conflicting inter-thread accesses. In particular, we seek to avoid evicting shadow values belonging to threads that are underrepresented in the cell and instead target redundant entries from already-dominant threads.

Formally, let each shadow cell contain metadata about recent accesses to a given memory region, including the thread identifier responsible for each access. When all slots are occupied and a new access arrives, the algorithm examines the set of resident thread IDs in the shadow cell. If a single thread appears multiple times, one of its entries becomes a candidate for replacement. Among such candidates, we again apply a size-based heuristic—favoring eviction of the smallest access footprint to minimize the likelihood of discarding critical metadata. In the event that all thread IDs are unique (i.e., maximal diversity), the policy defaults to the size-aware heuristic alone.

Preserving accesses from distinct threads increases the ability of a shadow cell in capturing data races, which by definition occur between different threads. Redundant entries from a single thread often represent spatially close operations and are less likely to introduce new information relevant to a potential race. By eliminating such redundancy, the policy makes room for new accesses while maintaining the diversity necessary for effective inter-thread conflict detection.

Crucially, this strategy incurs no significant memory overhead, as it operates solely on the thread identifiers already stored in the shadow values. The computational cost of evaluating thread diversity is minimal, given the small fixed number of slots per shadow cell (typically four). In return, the detector gains an increased chance of retaining metadata that will participate in future racy interactions, especially in regions of memory accessed by many concurrent threads.

Access-recency-aware eviction policy

The third replacement policy leverages temporal information already embedded in the shadow metadata to guide eviction decisions. Inspired by the principles of memory hierarchy design—where recently accessed data is more likely to be reused—this strategy favors the retention of more recently observed accesses.

As previously explained, each shadow value in TSan v3 stores the thread identifier and the thread-local epoch at the time of the access. Meanwhile, each thread maintains a vector clock tracking the most recent epochs it has observed for all threads. These structures collectively provide a mechanism to reason about the relative recency of memory accesses across threads.

When a shadow replacement is required, the current thread evaluates the existing shadow values to identify the least recently observed one. Let VC_{cur} denote the vector clock of the current thread. For each candidate shadow slot associated with thread T_i and epoch E_i, the thread computes the age as $VC_{cur}[T_i] - E_i$. A larger value suggests that the access occurred further in the past relative to the timeline of the current thread.

The shadow value with the largest computed age is selected for eviction. In case of ties, a secondary heuristic—such as favoring smaller access sizes or randomly choosing among equally old entries—may be applied. This policy thereby implements a recency-based eviction scheme, but does so by reusing thread-local logical timestamps that are already maintained by the underlying race detection logic.

By prioritizing the retention of recent memory accesses, this policy improves the likelihood of preserving shadow values that will be relevant for upcoming conflicting accesses. It is particularly beneficial in scenarios involving frequent thread interactions or fine-grained interleaving, where the temporal proximity of accesses increases the chance of a data race.

6.5 Second Blind Spot : Non-Atomicity of the Race Checking Procedure

While each load and store operation on shadow memory in TSan v3 is performed atomically—ensuring that individual shadow values are read and written in a consistent and complete form—the race detection logic as a whole is not atomic. That is, the full sequence implemented in Algorithm 4, which encompasses reading shadow values, checking for conflicts, and potentially writing a new shadow value, does not execute as a single atomic transaction.

This lack of end-to-end atomicity introduces another race detection blind spot. Consider two threads simultaneously accessing the same memory location : one thread may finish reading and validating the shadow memory state, and determine that no conflict exists. However, before it updates the shadow memory to reflect its own access, the second thread may interleave, perform the same check, and then complete its update first. As a result, both threads may miss the race condition because they evaluated the shadow memory against an outdated view—one that did not yet reflect the access performed by the other thread.

We propose to formulate this blind spot more precisely using the following condition :

$$\texttt{update}_i^{T_1} > \texttt{load}_i^{T_2} \quad \wedge \quad \texttt{update}_j^{T_2} > \texttt{load}_j^{T_1} \tag{6.1}$$

Here, $\texttt{load}_k^T$ and $\texttt{update}_k^T$ respectively denote the load and update operations performed by thread T on the k-th shadow value during the execution of Algorithm 4. This condition captures the scenario where thread T_1 ends up writing to the i-th shadow slot, and thread T_2 to the j-th slot, but each performs its shadow memory check *before* the update performed by the other thread occurs.

This form of interleaving is not prevented by atomic shadow value operations alone. It arises because the correctness of the detection relies on a sequence of operations that, in practice, are not executed atomically. Consequently, even though each individual shadow value remains internally consistent, the overall reasoning about data races can be invalidated by concurrent updates, resulting in missed data races.

6.5.1 Mitigation : enforcing atomic race checking through mutual exclusion

A possible mitigation for this blind spot is to enforce mutual exclusion around the combined race checking and shadow update procedure. In this scheme, each thread acquires a mutex before performing the race detection routine and releases it only after the corresponding shadow entries have been safely updated. This guarantees atomicity between detection and update, preventing interleaved executions that could conceal true data races. Although such serialization may introduce contention in highly parallel workloads, it provides a correctness-preserving safeguard against the loss of races caused by concurrent race checking.

6.6 Evaluation

To evaluate the effectiveness of our mitigation approaches for the two identified blind spots, we extended the TSan v3 runtime library with the proposed algorithmic improvements. For the first blind spot, we implemented three heuristic-based shadow replacement strategies that replace the default random policy of TSan. Our strategies prioritize shadow value eviction based on informed heuristics :

— Our first strategy prioritizes overwriting the shadow value corresponding to the access with the smallest byte width.
— Our second strategy prefers evicting shadow values belonging to the most dominant thread (i.e., one that already occupies multiple slots), aiming to increase thread di-

versity.

— Our third strategy uses an age-based heuristic that approximates least-recently-used semantics through thread-local epoch comparisons.

For the second blind spot, we introduced a mutex-based mechanism that enforces atomicity between race checking and shadow metadata updates within the same runtime environment.

We conducted three complementary evaluation studies to assess these extensions. The first employs a synthetic stress-test benchmark explicitly designed to expose race detection blind spots caused by shadow value eviction. The second evaluates the impact of our shadow overwriting strategies on applications from the PARSEC 3.0 benchmark suite. The third extends the stress test to include the mutex-based mitigation, evaluating its effect on detection coverage and runtime performance.

6.6.1 Stress-test shadow value eviction

To investigate the relative effectiveness of the proposed shadow replacement heuristics, we developed a synthetic microbenchmark specifically designed to stress-test shadow value eviction. The benchmark spawns six threads, where the first five carry out non-overlapping writes that collectively cover all eight bytes of a shared, 8-byte aligned memory region. A sixth thread then performs a 1-byte write to a randomly selected offset in the same region, without any synchronization. This setup ensures that each execution contains exactly one data race, between the sixth thread and one of the preceding accesses.

The first five access sizes are selected from the valid sets {4, 1, 1, 1, 1} and {2, 2, 2, 1, 1}, which together yield 15 distinct permutations that fully span the 8-byte region without overlap. For each execution, one of these permutations was randomly chosen to define the layout of the first five accesses, introducing variability in access arrangement.

Since TSan maps every 8-byte application memory region to four shadow slots, the arrival of the fifth access forces an eviction, potentially discarding metadata needed for detecting the race with the sixth access. We executed this benchmark for 100,000 iterations under each of the four strategies. Using TSan with its default random replacement approach, we observed that 26% of races were missed. The size-aware policy reduced the miss rate to 16%, demonstrating clear improvements. The access-recency-aware strategy achieved a miss rate of 20%, while the thread-diversity-aware strategy performed slightly less effectively with a 22% miss rate.

It is important to note that while this benchmark provides a direct and verifiable setup for evaluating the size-aware approach, it is less suitable for capturing the effects of the access-

recency-aware and thread-diversity-aware heuristics. The influence of recency depends on timing, and diversity cannot be assessed in a single-access-per-thread design. Therefore, real workloads such as PARSEC are better suited for evaluating these heuristics as discussed in 6.6.2, as these heuristics are inspired by broader principles such as temporal locality and thread interaction patterns that cannot be intuitively modeled in a synthetic benchmark.

6.6.2 PARSEC 3.0 benchmark suite

In our second evaluation study, we applied all four variants of the TSan runtime (original plus three alternatives) to a subset of multithreaded applications from the PARSEC 3.0 benchmark suite, namely `blackscholes`, `canneal`, `fluidanimate`, and `streamcluster`. We chose PARSEC as it is a widely recognized multithreaded benchmark suite, offering realistic and diverse workloads. However, since it does not contain known data races, we introduce three heuristic metrics that estimate the likelihood of retaining information relevant to race detection if races were present. Each metric is formally defined based on the semantics of a data race and the corresponding detection procedure employed in happens-before race detection. We measure :

— **Inter-thread Access Overlap (%)** : Measures the increase in the number of memory regions accessed by multiple threads with overlapping byte ranges. This metric is directly associated with the fundamental requirement for data race detection : the ability to detect overlapping memory accesses from different threads. Therefore, a higher rate of inter-thread overlaps indicates that the detector retains more relevant access information, increasing the opportunity to recognize potential races. The metric reflects the intuition that a tool better preserving information about overlapping accesses is more likely to detect races if they were to occur.

— **Vector Clock Checks (%)** : Indicates the rise in executed vector clock comparisons due to more detected overlaps. While inter-thread overlapping accesses are a necessary condition for data races, not all such overlaps indicate actual race. For example, cases involving only read operations or atomic accesses are excluded (as indicated in line 20 of Algorithm 4). This metric captures the ability of a tool to proceed to the final verification phase, where it evaluates the happens-before relationship by comparing vector clocks (line 23 in Algorithm 4). The rationale is that a higher number of such comparisons implies more candidate conflicts were preserved and analyzed, thereby increasing the likelihood that true data races would be detected if present.

— **Coverage Loss Due to Shadow Overwrite (%)** : Quantifies the reduction in cases where a shadow value was overwritten before a race could be detected. While inter-

thread memory access overlap is a necessary condition for a data race, this metric captures how much of that potentially useful information is lost due to shadow value eviction. It reflects the ability of each tool to retain overlapping access records long enough to enable race detection, thus indicating the extent to which the overwriting policy affects detection coverage.

TABLE 6.1 Impact of the access-size-aware overwrite strategy across selected PARSEC benchmarks

Benchmark	Overlap Increase (%)	VC Checks (%)	Coverage Loss Decrease (%)
`blackscholes`	5.1	6.2	7.4
`canneal`	6.3	8.7	9.2
`fluidanimate`	9.8	10.2	11.6
`streamcluster`	7.2	9.1	8.8

TABLE 6.2 Impact of the thread-diversity-aware overwrite strategy across selected PARSEC benchmarks

Benchmark	Overlap Increase (%)	VC Checks (%)	Coverage Loss Decrease (%)
`blackscholes`	3.2	4.1	5.3
`canneal`	4.5	5.6	6.1
`fluidanimate`	7.1	7.8	8.6
`streamcluster`	5.2	6.3	6.0

TABLE 6.3 Impact of the access-recency-aware overwrite strategy across selected PARSEC benchmarks

Benchmark	Overlap Increase (%)	VC Checks (%)	Coverage Loss Decrease (%)
`blackscholes`	4.9	6.4	6.5
`canneal`	5.8	7.9	8.8
`fluidanimate`	9.2	10.4	11.1
`streamcluster`	6.7	8.5	8.1

Tables 6.1–3 summarize the effects of the three overwrite strategies across selected PARSEC benchmarks. All three approaches consistently improve overlap detection and reduce coverage loss compared to the baseline random replacement policy, though the magnitude of improvement varies both by strategy and by benchmark characteristics.

The access-size-aware strategy (Table 6.1) yields the most pronounced improvements overall. In particular, `fluidanimate`, which is highly memory intensive and exhibits substantial inter-thread sharing, shows increases of 9.8% in overlap detection, 10.2% in vector clock checks, and 11.6% in coverage loss reduction. Even in comparatively less demanding workloads such

as `blackscholes`, the strategy still provides a 7.4% reduction in coverage loss. These results support the intuition behind this policy : retaining larger-span accesses in shadow memory increases the likelihood of preserving metadata that will participate in future races.

The thread-diversity-aware strategy (Table 6.2) shows more modest improvements. The policy is effective in balancing representation among threads, but because it does not directly account for access footprint or temporal relevance, it lags behind the other two strategies in overall gains. For instance, `fluidanimate` still benefits from an 8.6% reduction in coverage loss, but this remains several percentage points below what the size- and recency-aware strategies achieve.

The access-recency-aware strategy (Table 6.3) performs very closely to the access-size-aware policy, particularly in memory-intensive workloads. Again, `fluidanimate` stands out with an 11.1% reduction in coverage loss, only slightly below the size-aware strategy. By favoring eviction of stale shadow entries, this policy preserves metadata for accesses more likely to be involved in imminent conflicts. The benefit is especially visible in benchmarks with sustained interleaving of accesses, where maintaining temporal proximity of shadow values aligns well with the underlying race patterns.

Overall, the results indicate that while all three strategies reduce the likelihood of missing races due to shadow overwriting, the access-size-aware and access-recency-aware approaches consistently provide the greatest improvements. The thread-diversity-aware approach, though beneficial, achieves comparatively smaller gains. These findings suggest that heuristics tied to access footprint and temporal locality are more effective predictors for preserving race-relevant shadow metadata than thread balance alone.

6.6.3 Stress-test with mutex-based race checking

This evaluation extends the stress test benchmark introduced earlier, which was used to assess the four shadow replacement strategies—namely the original random replacement policy of TSan, and the proposed access-size-aware, access-recency-aware, and thread-diversity-aware approaches. In this new experiment, each of these four variants is further combined with the proposed mutex-based mitigation for the second blind spot, enforcing mutual exclusion during the race checking and shadow metadata update process. By introducing a global lock, this configuration ensures that no two threads perform race checking simultaneously, thus eliminating interleaving effects that could obscure existing data races.

The results show that detection accuracy improved across all configurations, though to varying degrees. Specifically, the data race miss rate decreased from 26% to 24% for the original

random strategy, from 16% to 13% for the access-size-aware policy, from 20% to 18% for the access-recency-aware policy, and from 22% to 19% for the thread-diversity-aware policy. These improvements were, however, accompanied by an average runtime overhead of approximately 120% due to the serialization of race checking under the global lock.

Overall, this study shows that integrating the mutex-based mitigation effectively reduces the likelihood of missed races across all shadow replacement strategies. Nonetheless, the high runtime cost implies that its practical adoption depends on the analysis requirements. In scenarios where the added overhead is acceptable in exchange for a moderate increase in detection precision, this approach can serve as a practical mitigation to strengthen analysis reliability.

6.7 Conclusion

In this paper, we identified and analyzed two previously unreported implementation-induced blind spots in ThreadSanitizer v3, a widely adopted data race detector integrated into both LLVM and GCC. While prior research on race detection tools has predominantly focused on their algorithmic correctness and theoretical guarantees, our work highlights impactful inaccuracies rooted in the actual implementation of the detection logic.

The first blind spot stems from the policy used to overwrite shadow values in the event of all shadow slots being occupied. The default random replacement strategy can evict important access information, potentially suppressing race reports. To address this, we proposed a set of targeted mitigations, each guided by a different heuristic. The first strategy prioritizes overwriting shadow values associated with narrower memory accesses, thereby improving the likelihood of preserving critical race-relevant information. The second strategy seeks to increase thread diversity by evicting shadow values from threads that already dominate multiple slots. The third strategy introduces a recency-based heuristic that leverages lightweight epoch tracking. Our evaluation confirms the efficacy of these approaches, with all three reducing coverage loss compared to the default policy across both synthetic stress tests and PARSEC benchmarks. Taken together, the results indicate that while all proposed strategies enhance race detection coverage, the access-size-aware and access-recency-aware policies consistently yield the largest improvements, with the size-aware policy emerging as the most effective overall, and the thread-diversity heuristic providing more modest gains.

The second blind spot arises from the non-atomicity of the overall race checking and shadow updating procedure. Although individual shadow memory reads and writes are performed atomically, interleaving between threads during the full check-update cycle can result in lost

race reports under specific interleaved execution patterns. To make this behavior explicit, we formally formulated the condition under which such race suppression may occur. We further evaluated a mutex-based mitigation designed to address this issue, and observed that it successfully reduced missed races across all shadow replacement strategies, though at the cost of a notable runtime overhead.

Collectively, our findings underscore the importance of assessing not only the theoretical soundness of race detectors, but also the implementation-level details that can compromise their effectiveness. These insights contribute to improving the reliability and robustness of race detection in production-grade tools.

Algorithm 4 Race Checking and Shadow Update Procedure in TSan v3

Require: Count of shadow values in a shadow cell *ShadowCnt*, Bitmask encoding the type of the current memory access *typ*, current thread state *thr*, description of the current access *cur*
Require: Shadow cell corresponding to the accessed memory region *shadow_cell*
Ensure: *true* if data race detected, *false* otherwise

1: $stored \leftarrow false$
2: **for** $idx \leftarrow 0$ **to** $ShadowCnt - 1$ **do**
3: $old \leftarrow shadow\_cell[idx]$ ▷ atomic load
4: **if** *old* is empty **then** ▷ Found an empty slot ; store current access if required
5: **if** $\neg(typ \wedge \textsc{AccessCheckOnly}) \wedge \neg stored$ **then**
6: $shadow\_mem[idx] \leftarrow cur$ ▷ atomic store
7: $stored \leftarrow true$
8: **end if**
9: **return** *false*
10: **end if**
11: **if** $(cur.access \wedge old.access) = 0$ **then** ▷ No data race if accessed bytes do not overlap
12: **continue**
13: **end if**
14: **if** $cur.sid = old.sid$ **then** ▷ No data race if both accesses are from the same thread
15: **if** $\neg(typ \wedge \textsc{AccessCheckOnly}) \wedge (cur.access = old.access) \wedge old.\textsc{IsRWWeakerOrEqual}(typ)$ **then**
16: $shadow\_mem[idx] \leftarrow cur$
17: $stored \leftarrow true$
18: **end if**
19: **continue**
20: **end if**
21: **if** $old.\textsc{IsBothReadsOrAtomic}(typ)$ **then** ▷ No data race if both accesses are read or atomic
22: **continue**
23: **end if**
24: **if** $thr.clock[old.sid] \geq old.epoch$ **then** ▷ Check clocks
25: **continue**
26: **end if**
27: $\textsc{DoReportRace}(thr, shadow\_mem, cur, old, typ)$
28: **return** *true*
29: **end for** ▷ After checking all slots, store in a random slot if not already stored
30: **if** *stored* **then**
31: **return** *false*
32: **end if**
33: $index \leftarrow \textsc{Random}(0, kShadowCnt - 1)$
34: $shadow\_mem[index] \leftarrow cur$
35: **return** *false*

CHAPITRE 7 CONCLUSION

Ce chapitre conclut cette thèse en revenant sur ses trois questions de recherche centrales et en résumant la manière dont chacune a été abordée à travers l'axe de recherche correspondant. La section suivante examine ensuite les principales limitations associées à chaque contribution. Enfin, le chapitre propose des pistes potentielles pour des travaux futurs s'appuyant sur les résultats présentés dans cette thèse.

7.1 Synthèse des travaux

Le premier axe de recherche a exploré la possibilité de tirer parti des avancées matérielles récentes en matière de traçage d'exécution pour concevoir un détecteur de conditions de course (data races) à faible surcoût, adapté aux environnements contraints en ressources, tels que les systèmes embarqués, tout en préservant les capacités de détection offertes par les outils existants à fort surcoût. Dans ce but, nous avons conçu et implémenté ThreadMonitor (TMon), un détecteur post-mortem de conditions de course destiné aux programmes C et C++ multithreadés utilisant la bibliothèque Pthread. TMon a été spécifiquement conçu pour offrir la même analyse exhaustive que ThreadSanitizer (TSan), tout en réduisant significativement le surcoût d'exécution. Pour ce faire, TMon repose sur deux phases principales : la phase de traçage et la phase d'analyse post-mortem. Lors de la phase de traçage, TMon utilise les paquets `ptwrite` d'Intel, une fonctionnalité récemment généralisée de Intel Processor Trace (Intel PT), caractérisée par un très faible surcoût, afin de tracer les mêmes événements que ceux surveillés par TSan. Grâce à ces paquets, TMon enregistre, pour chaque événement, exactement les mêmes informations d'exécution que celles capturées par TSan à des fins d'analyse. Par la suite, l'analyseur post-mortem de TMon reconstruit la séquence des événements à partir des données de trace pour déterminer si l'exécution tracée présente des conditions de course. TMon n'introduit aucun surcoût direct sur la mémoire de données, entraîne un surcoût minime sur la mémoire d'instructions et provoque un ralentissement très faible. Nos expérimentations montrent qu'en moyenne, TMon impose un surcoût d'exécution 2 à 6 fois inférieur à celui de TSan, même en tenant compte du coût d'écriture des traces sur disque.

Le deuxième axe de recherche s'est attaché à déterminer si une détection exhaustive des violations temporelles et spatiales d'accès à la mémoire de tas dans les programmes C pouvait être réalisée en combinant la propagation dynamique de métadonnées associées aux pointeurs avec une analyse à la compilation, sans engendrer un surcoût important. À cette fin, nous

avons développé AddressMonitor (AMon), un outil de vérification à l'exécution qui intègre le marquage des pointeurs avec une analyse et des transformations effectuées à la compilation. Plus précisément, AMon détecte les accès hors limites (out-of-bound accesses), les utilisations après libération (use-after-free), les doubles libérations (double-free) et les fuites de mémoire (memory leaks). De plus, il permet d'identifier deux autres pratiques de programmation dangereuses liées à l'accès mémoire : le passage d'un pointeur non-base aux fonctions standard de réallocation ou de libération de mémoire, ainsi que les utilisations potentielles après libération causées par une réallocation. AMon combine le marquage des pointeurs avec une analyse et des transformations à la compilation. À la compilation, il identifie les opérandes pointeurs issus du tas, génère des variantes non marquées, et remplace les pointeurs originaux en conséquence. À l'exécution, AMon attribue une valeur de marquage unique à chaque objet alloué, l'insère dans le pointeur, et maintient une table des objets afin de suivre les informations spatiales, temporelles et de débogage tout au long de l'exécution. Nos résultats expérimentaux montrent qu'AMon offre un compromis plus avantageux entre surcoût en temps d'exécution et surcoût mémoire que les outils de pointe existants. L'efficacité et la pertinence pratique d'AMon ont été démontrées par son intégration réussie dans un système informatique embarqué propriétaire déployé par nos partenaires industriels chez Ericsson.

Le troisième axe de recherche a examiné comment les compromis d'implémentation effectués pour réduire le surcoût dans les détecteurs dynamiques de conditions de course peuvent introduire des angles morts de détection non documentés, et si de telles limitations peuvent être corrigées sans accroître le temps d'exécution ni la consommation mémoire. Nous nous sommes intéressés à TSan, un détecteur de conditions de course de pointe intégré aux chaînes de compilation Clang et GCC. Nous avons étudié son sous-système de mémoire d'ombre — chargé d'enregistrer et de vérifier l'historique des accès mémoire — et identifié deux angles morts jusqu'alors non documentés : l'un causé par l'éviction aléatoire lors du remplacement des emplacements dans la mémoire d'ombre, et l'autre résultant du caractère non atomique de la vérification des courses et de la mise à jour des valeurs d'ombre. Nous avons formalisé les conditions dans lesquelles ces angles morts apparaissent et proposé des stratégies d'atténuation permettant d'améliorer la couverture de détection sans affecter le surcoût initial de l'outil.

7.2 Limitations de la solution proposée

Pour TMon, le volume des données de trace générées peut devenir important. En pratique, la taille de ces données dépend principalement du nombre d'accès mémoire effectués par l'application testée. Bien que le décodage des données de trace et l'analyse post-mortem

puissent être réalisés sur une machine distincte de celle où l'exécution du programme a eu lieu, la gestion du stockage et du transfert de fichiers de trace volumineux demeure un défi lorsque la quantité de données générées est considérable.

AMon exploite le marquage des pointeurs, en intégrant directement la valeur de marquage dans la représentation du pointeur. Par conséquent, le nombre maximal d'objets marqués *actifs* pouvant être suivis simultanément est limité par le nombre de bits d'adresse inutilisés dans l'architecture sous-jacente et exploités par AMon pour le stockage du marquage. Par exemple, dans les architectures 64 bits, les deux octets les plus significatifs sont généralement inutilisés et peuvent être attribués au stockage des valeurs de marquage. Bien que cette conception impose une limite théorique, nos expériences n'ont révélé aucun cas pratique où celle-ci serait atteinte. Sur notre système de test 64 bits, qui offrait deux octets inutilisés pour le stockage du marquage, AMon — grâce à l'utilisation de nouvelles techniques de recyclage de marquages — a atteint une couverture complète des applications réelles du banc d'essai SPEC CPU 2017, même lors de leur exécution avec les jeux de données de référence, qui constituent les entrées les plus volumineuses de la suite, sans jamais atteindre le nombre maximal d'objets de tas actifs autorisés par l'espace de marquage. Néanmoins, l'existence de cette limite théorique demeure une caractéristique inhérente à l'approche.

Bien que le troisième axe de recherche ait apporté une contribution significative en mettant en évidence des angles morts de détection induits par l'implémentation dans un détecteur de conditions de course à la fine pointe et en proposant des stratégies d'atténuation améliorant la couverture sans accroître le surcoût d'exécution ou mémoire, cela n'exclut pas l'existence d'autres angles morts ailleurs dans l'outil. Notre étude s'est spécifiquement concentrée sur le sous-système de mémoire fantôme — à savoir les mécanismes responsables du stockage des valeurs fantômes et de la vérification des accès mémoire à l'exécution — car ce composant joue un rôle central dans la conception de l'outil et contribue de manière substantielle à son surcoût. Néanmoins, d'autres parties de l'implémentation, qui dépassaient le cadre de cette étude, peuvent également introduire des limitations de détection qui méritent une investigation plus approfondie.

7.3 Améliorations futures

La technique novatrice proposée par TMon est extensible à toute autre architecture ou tout autre mécanisme de traçage, à condition que les informations requises puissent toujours être tracées avec un surcoût d'exécution raisonnablement faible. L'évolution de cette méthodologie au fil du temps offre également la possibilité d'identifier des conditions de course issues d'accès mémoire effectués par des périphériques de systèmes embarqués via l'accès direct à la

mémoire (Direct Memory Access – DMA). Dans de tels environnements, des blocs matériels peuvent accéder à la mémoire d'une manière analogue à des conditions de course, mais qui diffère des accès mémoire classiques initiés uniquement par l'exécution d'instructions du processeur. Cela rend nécessaire la mise en œuvre d'une analyse basée sur le traçage des événements DMA. Bien que cette approche soit exigeante, elle demeure réalisable et peut considérablement améliorer la détection des conditions de course dans les systèmes embarqués complexes. En outre, le développement futur de cette approche pourrait mener à une mise en œuvre matérielle de la détection des conditions de course, accélérant ainsi ce processus et permettant l'identification en temps réel de ces conditions lors de l'exécution du programme.

Une autre piste de travaux futurs visant à réduire encore le surcoût de TMon consisterait à étendre l'analyse de redondance effectuée à la compilation au-delà des cas actuellement couverts, afin de détecter un plus grand nombre d'accès mémoire redondants dans le contexte de la détection des conditions de course. Une telle amélioration permettrait de réduire davantage le surcoût d'instrumentation inutile, de diminuer le volume des données de trace générées et, en fin de compte, d'optimiser encore le processus global de vérification.

Une piste possible pour de futurs travaux visant à améliorer AMon consisterait à réduire la dépendance à l'identification conservatrice des pointeurs en y intégrant une analyse plus avancée effectuée à la compilation. Dans la conception actuelle, tous les arguments de fonction de type pointeur, les pointeurs retournés par des appels de fonction et les variables globales au niveau du module stockant des pointeurs sont inclus de manière conservatrice afin de couvrir l'ensemble des pointeurs potentiellement marqués (tainted) provenant de sources externes. Bien que cette surestimation prudente garantisse la solidité de l'approche, elle peut également entraîner une instrumentation inutile et un surcoût d'exécution plus élevé. Améliorer la phase d'analyse statique afin d'identifier plus précisément les pointeurs de tas pourrait réduire la nécessité de recourir à des hypothèses conservatrices, diminuant ainsi l'instrumentation superflue et améliorant l'efficacité globale.

Dans une perspective plus large, notre troisième axe de recherche a mis en évidence l'importance d'aller au-delà des fondements algorithmiques des techniques de vérification dynamique pour examiner de manière critique leurs implémentations pratiques. Dans de nombreux cas, les compromis introduits pour maîtriser le surcoût en temps d'exécution et en mémoire peuvent conduire à des limitations de détection substantielles, plaçant les capacités réelles des outils bien en deçà des objectifs théoriques au niveau algorithmique souvent avancés dans leurs présentations et masquant leur véritable efficacité de détection. Des recherches futures dans cette direction pourraient permettre de mieux comprendre les capacités réelles et les contraintes des outils existants, offrant ainsi à la fois au milieu académique et à l'industrie

la possibilité d'identifier les besoins réels et de définir des priorités plus éclairées pour faire progresser l'état de l'art en matière de vérification dynamique.

RÉFÉRENCES

[1] D. Bakhvalov. Enhance performance analysis with Intel Processor Trace. [En ligne]. Disponible : https://easyperf.net/blog/2019/08/23/Intel-Processor-Trace

[2] B. Swain *et al.*, "Openrace : An open source framework for statically detecting data races," ser. 2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness), 2021, p. 25–32.

[3] V. Vojdani et V. Vene, "Goblint : Path-sensitive data race analysis," ser. Annales Univ Sci Budapest, Sect Comp, 2009, p. 141–155.

[4] D. Engler et K. Ashcraft, "Racerx : Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, nº. 5, p. 237–252, 2003.

[5] P. Pratikakis, J. S. Foster et M. Hicks, "Locksmith : Practical static race detection for c," *ACM Trans Program Lang Syst*, vol. 33, nº. 1, p. 1–55, 2011.

[6] V. Vojdani *et al.*, "Static race detection for device drivers : the goblint approach," ser. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016, p. 391–402.

[7] E. Saillard *et al.*, "Static local concurrency errors detection in mpi-rma programs," ser. 2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness), 2022, p. 18–26.

[8] M. Naik, A. Aiken et J. Whaley, "Effective static race detection for java," ser. Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, p. 308–319.

[9] M. Abadi, C. Flanagan et S. N. Freund, "Types for safe locking : Static race detection for java," *ACM Trans Program Lang Syst*, vol. 28, nº. 2, p. 207–255, 2006.

[10] "Clang 19.0.0git documentation : Thread safety analysis," accessed April 19, 2024. https://clang.llvm.org/docs/ThreadSafetyAnalysis.html.

[11] S. Blackshear *et al.*, "Racerd : Compositional static race detection," *Proceedings of the ACM on Programming Languages*, vol. 2, nº. OOPSLA, p. 1–28, 2018.

[12] J. W. Voung, R. Jhala et S. Lerner, "Relay : Static race detection on millions of lines of code," ser. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2007, p. 205–214.

[13] S. Cui *et al.*, “Smallrace : Static race detection for dynamic languages-a case on smalltalk,” ser. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, p. 1136–1147.

[14] D. Marino, M. Musuvathi et S. Narayanasamy, “Literace : Effective sampling for lightweight data-race detection,” *ACM SIGPLAN Notices*, vol. 44, n°. 6, p. 134–143, 2009.

[15] T. Sheng *et al.*, “Racez : A lightweight and non-invasive race detection tool for production applications,” ser. Proceedings of the 33rd International Conference on Software Engineering, 2011, p. 401–410.

[16] M. D. Bond, K. E. Coons et K. S. McKinley, “Pacer : Proportional detection of data races,” *ACM SIGPLAN Notices*, vol. 45, n°. 6, p. 255–268, 2010.

[17] R. H. B. Netzer et B. P. Miller, “What are race conditions ? some issues and formalizations,” *ACM Lett Program Lang Syst*, vol. 1, n°. 1, p. 74–88, 1992.

[18] Intel® 64 and ia-32 architectures software developer manuals - vol. 3c. [En ligne]. Disponible : https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[19] Coresight components technical reference manual. [En ligne]. Disponible : https://developer.arm.com/documentation/ddi0314/h/

[20] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” dans *Concurrency : the Works of Leslie Lamport*, 2019, p. 179–196.

[21] S. Savage *et al.*, “Eraser : A dynamic data race detector for multithreaded programs,” *ACM Transactions on Computer Systems (TOCS)*, vol. 15, n°. 4, p. 391–411, 1997.

[22] Valgrind user manual : Helgrind. [En ligne]. Disponible : https://valgrind.org/docs/manual/hg-manual.html

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

[24] M. Sulzmann et K. Stadtmüller, “Efficient, near complete, and often sound hybrid dynamic data race prediction,” dans *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*, 2020, p. 30–51.

[25] U. Mathur, D. Kini et M. Viswanathan, “What happens-after the first race ? enhancing the predictive power of happens-before based dynamic race detection,” *Proceedings of the ACM on Programming Languages*, vol. 2, n°. OOPSLA, p. 1–29, 2018.

[26] Q.-L. Chen *et al.*, “Detecting data races caused by inconsistent lock protection in device drivers,” dans *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, p. 366–376.

[27] P. Zhou, R. Teodorescu et Y. Zhou, “Hard : Hardware-assisted lockset-based race detection,” dans *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, p. 121–132.

[28] E. Pozniansky et A. Schuster, “Multirace : efficient on-the-fly data race detection in multithreaded c++ programs,” *Concurrency and Computation : Practice and Experience*, vol. 19, n°. 3, p. 327–340, 2007.

[29] R. O’callahan et J.-D. Choi, “Hybrid dynamic data race detection,” dans *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2003, p. 167–178.

[30] S. Mincheva, “Sound thread local analysis for lockset-based dynamic data race detection,” 2017.

[31] X. Xie, J. Xue et J. Zhang, “Acculock : Accurate and efficient detection of data races,” *Software : Practice and Experience*, vol. 43, n°. 5, p. 543–576, 2013.

[32] T. Elmas, S. Qadeer et S. Tasiran, “Goldilocks : a race and transaction-aware java runtime,” *Acm Sigplan Notices*, vol. 42, n°. 6, p. 245–255, 2007.

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

[34] N. Nethercote et J. Seward, “Valgrind : a framework for heavyweight dynamic binary instrumentation,” dans *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA : Association for Computing Machinery, 2007, p. 89–100.

[35] K. Serebryany et T. Iskhodzhanov, “Threadsanitizer : Data race detection in practice,” ser. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009, p. 62–71.

[36] “Threadsanitizer v1 algorithm wiki,” accessed : 2025-05-31. https://code.google.com/archive/p/data-race-test/wikis/ThreadSanitizerAlgorithm.wiki. [En ligne]. Disponible : https://code.google.com/archive/p/data-race-test/wikis/ThreadSanitizerAlgorithm.wiki

[37] “Threadsanitizer, memorysanitizer – 2012 llvm developers’ meeting,” accessed : 2025-05-31. https://llvm.org/devmtg/2012-11/Serebryany_TSan-MSan.pdf. [En ligne]. Disponible : https://llvm.org/devmtg/2012-11/Serebryany_TSan-MSan.pdf

[38] K. Serebryany *et al.*, “Dynamic race detection with llvm compiler,” dans *RV 2011. Lecture Notes in Computer Science*, S. Khurshid et K. Sen, édit., vol. 7186. Heidelberg : Springer, 2012, p. 110–114.

[39] “D. vyukov on threadsanitizer soundness,” accessed : 2025-05-31. https://groups.google.com/g/thread-sanitizer/c/UuOzVuYWUA4. [En ligne]. Disponible : https://groups.google.com/g/thread-sanitizer/c/UuOzVuYWUA4

[40] C. Sadowski et J. Yi, “How developers use data race detection tools,” ser. Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, 2014, p. 43–51.

[41] “Llvm threadsanitizer documentation,” accessed : 2025-05-31. https://clang.llvm.org/docs/ThreadSanitizer.html. [En ligne]. Disponible : https://clang.llvm.org/docs/ThreadSanitizer.html

[42] “Gcc program instrumentation options,” accessed : 2025-05-31. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html. [En ligne]. Disponible : https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[43] “Tsan : New runtime (v3),” accessed : 2025-05-31. https://github.com/llvm/llvm-project/commit/b332134921b42796c6b46453eaf2affdc09e3154. [En ligne]. Disponible : https://github.com/llvm/llvm-project/commit/b332134921b42796c6b46453eaf2affdc09e3154

[44] Q.-L. Chen *et al.*, “Detecting data races caused by inconsistent lock protection in device drivers,” ser. 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2019, p. 366–376.

[45] A. TehraniJamsaz *et al.*, “Deeprace : A learning-based data race detector,” ser. 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2021, p. 226–233.

[46] C. Cowan *et al.*, “Stackguard : Automatic adaptive detection and prevention of buffer-overflow attacks,” ser. USENIX security symposium, 1998, p. 63–78.

[47] H. Etoh et K. Yoda, “Propolice : Protecting from stack-smashing attacks,” *Technical Report, IBM Research Division, Tokyo Research Laboratory*, 2000.

[48] J. Edge, “"strong" stack protection for gcc,” linux Weekly News. Accessed : 2024-11-28. https://lwn.net/Articles/584225/. [En ligne]. Disponible : https://lwn.net/Articles/584225/

[49] MITRE, “2023 cwe top 10 kev weaknesses,” accessed : 2024-07-17. https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html. [En ligne]. Disponible : https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html

[50] Red Hat, “2023 red hat product security risk report,” accessed : 2024-07-17. https://www.redhat.com/en/resources/product-security-risk-report-2023. [En ligne]. Disponible : https://www.redhat.com/en/resources/product-security-risk-report-2023

[51] Y. Younan *et al.*, “Improving memory management security for c and c++,” *International Journal of Secure Software Engineering (IJSSE)*, vol. 1, nº. 2, p. 57–82, 2010.

[52] G. Novark et E. D. Berger, “Dieharder : Securing the heap,” dans *Proceedings of the 17th ACM conference on Computer and communications security*, ser. Proceedings of the 17th ACM conference on Computer and communications security, 2010, p. 573–584.

[53] S. Silvestro *et al.*, “Freeguard : A faster secure heap allocator,” dans *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, p. 2389–2403.

[54] B. Liu, P. Olivier et B. Ravindran, “Slimguard : A secure and memory-efficient heap allocator,” dans *Proceedings of the 20th International Middleware Conference*, ser. Proceedings of the 20th International Middleware Conference, 2019, p. 1–13.

[55] B. Wickman *et al.*, “Preventing use-after-free attacks with fast forward allocation,” dans *30th USENIX Security Symposium (USENIX Security 21)*, ser. 30th USENIX Security Symposium (USENIX Security 21), 2021, p. 2453–2470.

[56] K. Hohentanner, P. Zieris et J. Horsch, “Cryptsan : Leveraging arm pointer authentication for memory safety in c/c++,” dans *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, ser. Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, 2023, p. 1530–1539.

[57] F. Gorter *et al.*, “Sticky tags : Efficient and deterministic spatial memory error mitigation using persistent memory tags,” dans *2024 IEEE Symposium on Security and Privacy (SP)*, ser. 2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2024, p. 217–217.

[58] O. Oleksenko *et al.*, “Intel mpx explained : A cross-layer analysis of the intel mpx system stack,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, nº. 2, p. 1–30, 2018.

[59] T. Zhang, D. Lee et C. Jung, “Bogo : Buy spatial memory safety, get temporal memory safety (almost) free,” dans *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, p. 631–644.

[60] W. Huang *et al.*, “Lmp : light-weighted memory protection with hardware assistance,” dans *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016, p. 460–470.

[61] "Clang 20 documentation : Addresssanitizer," accessed : 2024-12-11. https://clang.llvm.org/docs/AddressSanitizer.html. [En ligne]. Disponible : https://clang.llvm.org/docs/AddressSanitizer.html

[62] T. Kroes *et al.*, "Delta pointers : Buffer overflow checks without the checks," dans *Proceedings of the Thirteenth EuroSys Conference*, ser. Proceedings of the Thirteenth EuroSys Conference, 2018, p. 1–14.

[63] K. Zhu, Y. Lu et H. Huang, "Scalable static detection of use-after-free vulnerabilities in binary code," *IEEE Access*, vol. 8, p. 78 713–78 725, 2020.

[64] "cwe_checker," accessed : 2024-11-26. https://github.com/fkie-cad/cwe_checker. [En ligne]. Disponible : https://github.com/fkie-cad/cwe_checker

[65] H. Shahriar et M. Zulkernine, "Classification of static analysis-based buffer overflow detectors," dans *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, ser. 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion, 2010, p. 94–101.

[66] P. Luo *et al.*, "Static detection of real-world buffer overflow induced by loop," *Computers Security*, vol. 89, p. 101616, 2020.

[67] J. D. Pereira, N. Ivaki et M. Vieira, "Characterizing buffer overflow vulnerabilities in large c/c++ projects," *IEEE Access*, vol. 9, p. 142 879–142 892, 2021.

[68] S. Shiraishi, V. Mohan et H. Marimuthu, "Test suites for benchmarks of static analysis tools," ser. 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2015, p. 12–15.

[69] A. Arusoaie *et al.*, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," ser. 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2017, p. 161–168.

[70] F. Kirchner *et al.*, "Frama-c : A software analysis perspective," *Formal Aspects of Computing*, vol. 27, n°. 3, p. 573–609, 2015.

[71] V. D'Silva, M. Payer et D. Song, "The correctness-security gap in compiler optimization," dans *2015 IEEE Security and Privacy Workshops*, ser. 2015 IEEE Security and Privacy Workshops, 2015, p. 73–87.

[72] M. J. Hohnka *et al.*, "Evaluation of compiler-induced vulnerabilities," *Journal of Aerospace Information Systems*, vol. 16, n°. 10, p. 409–426, 2019.

[73] "Module cwe_checker_lib : :checkers : :cwe_119," accessed : 2024-11-26. https://docs.cwe-checker.io/cwe_checker_lib/checkers/cwe_119/. [En ligne]. Disponible : https://docs.cwe-checker.io/cwe_checker_lib/checkers/cwe_119/

[74] T. Marjanov, I. Pashchenko et F. Massacci, “Machine learning for source code vulnerability detection : What works and what isn’t there yet,” *IEEE Security & Privacy*, vol. 20, nº. 5, p. 60–76, 2022.

[75] T. Chappelly *et al.*, “Machine learning for finding bugs : An initial report,” ser. 2017 IEEE workshop on machine learning techniques for software quality evaluation (MaLTeSQuE), 2017, p. 21–26.

[76] Y. Zhao *et al.*, “Buffer overflow detection for c programs is hard to learn,” ser. Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. Association for Computing Machinery, 2018, p. 8–9. [En ligne]. Disponible : https://doi.org/10.1145/3236454.3236455

[77] P. Krishnan *et al.*, “Why is static application security testing hard to learn ?” *IEEE Security & Privacy*, vol. 21, nº. 5, p. 68–72, 2023.

[78] S. Nagarakatte *et al.*, “Cets : compiler enforced temporal safety for c,” dans *Proceedings of the 2010 international symposium on Memory management*, ser. Proceedings of the 2010 international symposium on Memory management, 2010, p. 31–40.

[79] H. Cho *et al.*, “Vik : practical mitigation of temporal memory safety violations through object id inspection,” ser. Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, p. 271–284.

[80] V. Azhari *et al.*, “Efficient heap monitoring tool for memory leak detection and root-cause analysis,” dans *2021 IEEE International Conference on Big Data (Big Data)*, ser. 2021 IEEE International Conference on Big Data (Big Data), 2021, p. 3020–3030.

[81] “Leaksanitizer : false negative when functions stack frames overlay,” accessed : 2024-12-14. https://github.com/google/sanitizers/issues/937. [En ligne]. Disponible : https://github.com/google/sanitizers/issues/937

[82] M. L. Soffa, K. R. Walcott et J. Mars, “Exploiting hardware advances for software testing and debugging (nier track),” dans *Proceedings of the 33rd International Conference on Software Engineering*, 2011, p. 888–891.

[83] C. Pedersen et J. Acampora, “Intel code execution trace resources,” *Intel Technology Journal*, vol. 16, nº. 1, p. 130–136, 2012.

[84] A. Vasudevan, N. Qu et A. Perrig, “Xtrec : Secure real-time execution trace recording on commodity platforms,” dans *2011 44th Hawaii International Conference on System Sciences*. IEEE, 2011, p. 1–10.

[85] S. D. Sharma et M. Dagenais, “Hardware-assisted instruction profiling and latency detection,” *The Journal of Engineering*, vol. 2016, nº. 10, p. 367–376, 2016.

[86] Linux kernel documentation for intel pt. [En ligne]. Disponible : https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/perf-intel-pt.txt

[87] J. Mellor-Crummey, "Compile-time support for efficient data race detection in shared-memory parallel programs," *ACM SIGPLAN Notices*, vol. 28, n°. 12, p. 129–139, 1993.

[88] P. A. Buhr et A. S. Harji, "Concurrent urban legends," *Concurr Comput Pract Exp*, vol. 17, n°. 9, p. 1133–1172, 2005.

[89] P. A. Buhr, *Understanding Control Flow*. Springer, 2016.

[90] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun ACM*, vol. 21, n°. 7, p. 558–565, 1978.

[91] S. Savage *et al.*, "Eraser : A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, n°. 4, p. 391–411, 1997.

[92] R. O'Callahan et J.-D. Choi, "Hybrid dynamic data race detection," *ACM SIGPLAN Notices*, vol. 38, n°. 10, p. 167–178, 2003.

[93] K. Serebryany *et al.*, "Dynamic race detection with llvm compiler : Compile-time instrumentation for threadsanitizer," ser. International Conference on Runtime Verification, 2011, p. 110–114.

[94] D. Vyukov, "Address/thread/memorysanitizer : Slaughtering c++ bugs," accessed April 19, 2024. https://www.slideshare.net/sermp/sanitizer-cppcon-russia.

[95] ——, "tsan : new runtime (v3)," accessed April 19, 2024. https://github.com/llvm/llvm-project/commit/b332134921b42796c6b46453eaf2affdc09e3154.

[96] Llnl - intel inspector. [En ligne]. Disponible : https://hpc.llnl.gov/software/development-environment-software/intel-inspector

[97] "Clang 19.0.0git documentation : Threadsanitizer," accessed April 19, 2024. https://clang.llvm.org/docs/ThreadSanitizer.html.

[98] "Fft-parallelized-with-pthread-api," accessed April 19, 2024. https://github.com/EstellaPaula/FFT-parallelized-with-PThread-API/tree/master.

[99] P. C. van Oorschot, "Memory errors and memory safety : C as a case study," *IEEE Security & Privacy*, vol. 21, n°. 2, p. 70–76, 2023.

[100] M. Payer, *How memory safety violations enable exploitation of programs*, ser. The Continuing Arms Race : Code-Reuse Attacks and Defenses. Association for Computing Machinery and Morgan & Claypool, 2018. [En ligne]. Disponible : https://doi.org/10.1145/3129743.3129745

[101] S. Nagarakatte, "Full spatial and temporal memory safety for c," *IEEE Security & Privacy*, vol. 22, n°. 4, p. 30–39, 2024.

[102] V. Van der Veen *et al.*, “Memory errors : The past, the present, and the future,” dans *Research in Attacks, Intrusions, and Defenses : 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*, ser. 2012 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID), 2012, p. 86–106.

[103] L. Szekeres *et al.*, “Sok : Eternal war in memory,” dans *2013 IEEE Symposium on Security and Privacy*, ser. 2013 IEEE Symposium on Security and Privacy, 2013, p. 48–62.

[104] M. Abadi *et al.*, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, n°. 1, p. 1–40, 2009.

[105] D. Jeffrey, V. Nagarajan et R. Gupta, “Execution suppression : An automated iterative technique for locating memory errors,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, n°. 5, p. 1–36, 2008.

[106] F. Lu *et al.*, “A survey of detection methods for software use-after-free vulnerability,” dans *International Conference of Pioneering Computer Scientists, Engineers and Educators*. Springer, 2022, p. 272–297.

[107] M. Unterguggenberger *et al.*, “Multi-tag : A hardware-software co-design for memory safety based on multi-granular memory tagging,” dans *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, p. 177–189.

[108] H.-J. Boehm et S. V. Adve, “You don’t know jack about shared variables or memory models,” *Communications of the ACM*, vol. 55, n°. 2, p. 48–54, 2012.

[109] R. O’Callahan et J.-D. Choi, “Hybrid dynamic data race detection,” dans *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, California, USA : Association for Computing Machinery, 2003, p. 167–178.

[110] P. A. Buhr, *Understanding control flow : concurrent programming using $\mu C++$*, 1er éd. Switzerland : Springer Cham, 2016.

[111] S. Roy *et al.*, “Raptor : An online dynamic analysis for sound predictive data race detection,” The Ohio State University, Rapport technique, 2020.

[112] C. Flanagan et S. N. Freund, “Fasttrack : Efficient and precise dynamic race detection,” dans *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Dublin, Ireland : ACM, 2009, p. 121–133.

[113] U. Mathur, S. Qadeer et E. Yahav, “The complexity of dynamic data race prediction,” dans *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer*

Science (LICS). Association for Computing Machinery, 2020, p. 1–12.

[114] S. Savage *et al.*, "Eraser : A dynamic data race detector for multithreaded programs," dans *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France : ACM, 1997, p. 27–37.

[115] D. Entezari *et al.*, "A comparative study of dynamic data race detection techniques," dans *Proceedings of the International Conference on Software Engineering (ICSE)*, 2022, p. 404–416.

[116] M. D. Bond, M. Vilayannur et C. Flanagan, "Smarttrack : Efficient predictive race detection," dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020, p. 294–310.

[117] A. Tehrani *et al.*, "Deeprace : Finding data race bugs via deep learning," dans *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2019, p. 115–126.

[118] J. Doe et A. Smith, "Dynamic race detection with $o(1)$ samples," *arXiv preprint arXiv :2506.20127*, 2025.

[119] J. R. Wilcox, "Shrinkwrap : Efficient dynamic race detection," Washington University in St. Louis, Rapport technique, 2013.

[120] M. Amrani *et al.*, "Dynamic data-race detection through the fine-grained lens," dans *CONCUR, LIPIcs*, 2021.