Security enhancement in embedded hard real-time systems

Nicolas Bellec

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Nicolas BELLEC

Security enhancement in embedded hard real-time systems

Thèse présentée et soutenue à Rennes, le 23/05/2023
Unité de recherche : IRISA, Equipe PACAP

Direction de thèse :

Dirigée par :	Isabelle PUAUT	Professeur des Universités, IRISA Rennes
Encadrée par :	Guillaume HIET	Professeur, CentraleSupélec, Rennes
	Frédéric TRONEL	Professeur, CentraleSupélec, Rennes
	Simon ROKICKI	Maître de Conférence, ENS Rennes

Rapporteurs avant soutenance :

Vincent NICOMETTE	Professeur des Universités, INSA Toulouse
Karine HEYDEMANN	Maître de Conférence HDR, Sorbonne Université & Senior Expert Architecte, Thales

Composition du Jury :

Président :	Gildas AVOINE	Professeur des Universités, INSA Rennes
Examinateurs :	Vincent NICOMETTE	Professeur des Universités, INSA Toulouse
	Karine HEYDEMANN	Maître de Conférence HDR, Sorbonne Université & Senior Expert Architecte, Thales
	Thomas CARLE	Maître de Conférence, Université Toulouse III
Dir. de thèse :	Isabelle PUAUT	Professeur des Universités, IRISA Rennes
Encadrants de thèse :	Guillaume HIET	Professeur, CentraleSupélec, Rennes

Remerciements

Écrire cette thèse ne fut pas chose aisée. Je tiens donc à remercier en préambule toutes les personnes qui m'ont aidée tout au long de cette thèse.

Les premières personnes que je souhaite remercier sont mes encadrants de thèse, Isabelle Puaut, Frédéric Tronel, Guillaume Hiet et Simon Rokicki, qui m'ont épaulé durant ces (presque) quatre années de thèse. Un grand merci pour votre soutien et votre grande humanité, en particulier durant la période de rédaction.

Je remercie également Karine Heydemann et Vincent Nicomette pour avoir accepté de rapporter cette thèse ainsi que les autres membres du jury, Thomas Carle et Gildas Avoine.

Un remerciement particulier va à Gildas Avoine pour avoir accepté de présider mon jury de thèse et de m'avoir prodigué d'excellents conseils avec Hai Nam Tran durant le comité de suivi individuel. Merci aussi à Zoltan pour avoir pris le temps de me tutorer, ses conseils autour d'une pause-café à l'IRISA ont été très utiles.

J'ai eu la chance d'être très bien entouré durant ma thèse. J'ai pu en effet effectuer mes trois premières années de thèse dans un cadre accueillant qu'est l'IRISA au sein de l'équipe PACAP. Merci beaucoup aux membres de cette équipe pour leur accueil chaleureux, merci aux permanents, Erven, Pierre M., Isabelle, Pierre-Yves, André, Caroline et Damien, et aux non-permanents, Nassim, Lily, Camille, Antoine, Sara, Anis, Aurore, Hugo R., Nicolas, Pierre, Stefanos, Daniel et Niloofar, la "Team Gouter", qui ont animés les pauses-café avec une bonne humeur indétrônable. J'ai fini ma thèse avec un A.T.E.R à CentraleSupelec, dans l'équipe CIDRE, qui m'a très gentiment accueilli et je tiens à les remercier pour cet accueil des plus chaleureux, alors que je terminais mon manuscrit. Merci à l'équipe pédagogique et aux permanents, Jean-François, Frédéric, Guillaume, Pierre, Valérie, Pierre-François, Kévin, Michel, Emmanuelle, Ludovic, Yufei, Christophe, Gilles, et les non-permanents que j'ai pu rencontrer : Jérémy, Romain, Pierre-Victor, Nathan, Hélène, Racim, Maxime, Lionel, Jean-Marie, Lucas et Manuel. Un grand merci à Virginie, Lydie et Myriam, pour leur aide incroyable qu'elles m'ont apportée lors d'obscures démarches administratives.

Enfin, cette thèse n'aurait pas été possible sans le soutien d'Absint qui nous a fourni

gracieusement AiT, un logiciel propriétaire, qui m'a permis de faire tourner mes expériences. Pour cela, je les remercie.

Il y a aussi les membres des autres équipes de l'IRISA qui m'ont fait découvrir des sujets très intéressants. Pour cela, j'ai une pensée pour Benoit, merci pour tous ces cafés pris ensemble. Merci également à Nicolas et Mathias pour m'avoir invité à leurs séminaires "ELSE" de non-permanents, et plus généralement aux doctorants et post-doctorants de Sumo et Hycomes, Léo, Abdul, Bastien, Emily, Nicolas W., Hugo B., Aline, Adrian, et Joan, pour les pauses-café et fous rires partagés. Un grand merci à François Schwarzentruber et Sophie Pinchinat pour leur dynamisme et bienveillance.

Faire une thèse n'aurait pas été possible sans certains professeurs que j'ai pu rencontrer tout au long de ma scolarité, qui m'ont poussé à faire des sciences. En particulier, je remercie Mme Billerai qui m'a poussé à être plus rigoureux dans mes raisonnements, Mme Aulanier, pour m'avoir fait découvrir la recherche grâce à son atelier scientifique, et enfin à David Pichardie et Martin Quinson pour m'avoir donné ma chance en m'accueillant à l'ENS Rennes en informatique.

Merci à Nicolas L., Charles, Vincent, Nathan, Corentin, Pierre-Malo, Pierre-Louis, Jean-Pierre et Diego, mes amis de longue date, pour m'avoir fait découvrir les jeux de rôles et pour réussir à maintenir des séances hebdomadaires malgré la distance qui nous séparait durant la thèse. Merci pour cette bouffée d'oxygène.

Merci aussi à Clément, Rémi, Santiago, Léo, Solène, Hugo B., Hugo R., Lily et Joan pour toutes les soirées jeux autour d'une tasse de thé, et à Riwan pour tous ces moments de détente apportés.

Merci également aux amis avec qui je partage ma passion pour la danse. Merci à Filam, Guillaume De Longuemar, Léa, Guillaume Lefeuvre (encore un), Guillaume Bury (encore un !), Camille, Marie-Morgane et Marie-Audrey. Merci pour toutes les soirées de danse.

Merci à Pierre et Hadrien pour être d'aussi bons amis depuis aussi longtemps, j'ai toujours grand plaisir à vous revoir et à discuter avec vous.

Je remercie toutes les personnes de ma famille pour m'avoir supporté et soutenu pendant mes (longues) études, en particulier sur la dernière ligne droite qu'est cette thèse. J'ai une pensée émue pour mon père, avec une tristesse qu'il n'ait pas pu me voir commencer cette thèse. Merci de m'avoir transmis sa passion pour la technique dès mon enfance, d'avoir toujours été fier de moi. Merci également à ma mère d'être là pour moi et de m'avoir encouragé durant ma thèse.

Un autre type de soutien, plus original, m'a beaucoup apporté durant ma thèse : mon

chien, Plume. Merci de me forcer à sortir de nez de la recherche, et en fait de l'appartement, en particulier pendant les périodes de confinement.

Merci également à ma belle-famille, Sylvie, Frédéric, Mathilde, Barbara et Adrien, pour m'avoir accueilli aussi chaleureusement. Une pensée affectueuse aux deux derniers arrivants, Adélie et Orion.

Enfin, et c'est la plus importante, je remercie Emily, avec qui je partage ma vie. Elle a toujours été là, dans les meilleurs moments comme dans les pires, et pour cela, je la remercie du plus profond de mon cœur.

Table of Contents

Résumé en français

De nombreux systèmes doivent réagir à des événements avant une date limite stricte afin de garantir leurs bon fonctionnement. Les avions et les voitures sont des exemples classiques de systèmes répondant à ces exigences. Ces systèmes sont aujourd'hui équipés de nombreux capteurs et doivent être capables de répondre suffisamment rapidement aux commandes qu'ils reçoivent afin d'assurer la sûreté de leurs utilisateurs ainsi que des personnes qui les entourent. Ces systèmes sont appelés **Systèmes temps réel** (*Real Time Systems* ou RTS en anglais), car ils doivent être conçus de manière à réagir assez rapidement aux événements à venir en toutes circonstances.

Historiquement, les systèmes temps réel ont utilisé des canaux de communication isolés (typiquement un bus de donnée CAN) car ils sont plus prévisibles. Cependant, on assiste à l'émergence de nouveaux systèmes temps réel utilisant des communications sans fil pour recevoir des commandes, transmettre des données et communiquer avec d'autres systèmes. Les drones volants sont un bon exemple de ce nouveaux type de systèmes. Un drone volant est intrinsèquement un système temps réel car il doit toujours adapter sa propulsion en fonction de son environnement. Il peut être contrôlé à distance, utilisé pour capturer des images qui doivent être envoyées à l'utilisateur et enfin, il y a une tendance à l'utilisation de flottes de drones capables de s'organiser pour accomplir une mission. Dans ces situations, la communication sans fil est essentielle pour communiquer avec l'utilisateur (soit pour recevoir des commandes, soit pour envoyer des données) et entre les drones.

Ces nouveaux canaux de communication sont également une excellente occasion d'attaquer ces systèmes. Avec les systèmes temps réel classiques, comme les communications ne se faisaient que dans un environnement fermé et contrôlé, attaquer un système signifiait déjà pouvoir communiquer avec lui, ce qui était suffisamment difficile pour empêcher de nombreuses attaques. Avec ces nouveaux systèmes, il suffit à un attaquant d'être suffisamment proche (typiquement à portée du wifi ou du bluetooth du système) pour lancer une communication et potentiellement attaquer le système. Ainsi, la sécurité des systèmes temps réel devient de plus en plus importante au fur et à mesure du déploiement de ces systèmes.

Caractéristiques des systèmes en temps réel

Tout d'abord, décrivons certaines caractéristiques des systèmes temps réel qui ont un impact sur leur sécurité. Ces systèmes sont souvent écrits dans des **langages bas niveau** (typiquement du C) qui obligent les développeurs à gérer manuellement la mémoire. Ces langages sont sujets à des erreurs de gestion de la mémoire qui peuvent être exploitées pour attaquer les programmes. En utilisant ces erreurs, un attaquant peut briser les garanties du système en modifiant des parties de la mémoire qu'il ne devrait pas pouvoir changer. Cela peut conduire à un large éventail de problèmes affectant le système. Par exemple, un attaquant pourrait empêcher le système de répondre ou y exécuter son propre code, empêchant le système de fonctionner correctement.

Les systèmes temps réel sont également souvent **difficiles à mettre à jour**, car la modification du système nécessite de s'assurer à nouveau que toutes les échéances seront respectées par le système. De plus, les correctifs sont souvent plus difficiles à transmettre au système car ils ne disposent pas d'une connexion internet et doivent être en pause pour éviter de manquer des dates limites. Cela signifie que le délai entre la découverte d'une vulnérabilité et l'application d'un éventuel correctif peut être très long et que le système doit rester sûr pendant ce temps.

Enfin, les systèmes temps réel doivent être **prévisibles** afin de garantir les délais du système. Cela signifie que toute défense déployée pour protéger le système doit également être prévisible, en particulier dans l'estimation de son temps d'exécution, afin de s'assurer que le système peut toujours respecter ces garanties. En particulier, cela empêche le déploiement de nombreuses défenses modernes qui utilisent des sources d'aléatoire pour rendre plus difficile la tâche de l'attaquant qui doit trouver des informations cruciales pour monter son attaque[1].

Modèle de l'attaquant

De nombreuses attaques existent à différents niveaux d'un système. Certaines attaques se concentrent sur le réseau et les protocoles pour extraire des informations confidentielles, contourner les authentifications ou rendre le système indisponible. À l'inverse, certaines attaques se concentrent sur le matériel utilisé par le système, faisant fuir des informations par des canaux auxiliaires (utilisant des différences temporelles ou les émissions électro-

[1] tel que les protections *PIE* et *ASLR*

magnétiques) ou même en injectant des fautes pour modifier le comportement du système.

Dans cette thèse, nous concentrons notre attention sur les attaques logicielles qui utilisent une vulnérabilité sur la gestion de la mémoire. Nous supposons que l'attaquant a trouvé une erreur logicielle lui permettant de corrompre la mémoire, c'est-à-dire d'écrire dans une partie inattendue de cette dernière. Cette erreur peut alors être utilisée par l'attaquant pour modifier un point arbitraire de la mémoire. À titre d'exemple, l'attaquant pourrait avoir trouvé un dépassement de tampon qui lui permet d'écrire en dépassant les limites d'un tableau et de modifier d'autres variables sur la pile. Il pourrait alors utiliser cette vulnérabilité pour changer des variables importantes dans la pile et modifier le comportement du programme.

Nous nous concentrons sur ce modèle d'attaquant pour deux raisons. Premièrement, ces vulnérabilités sont courantes dans le code des langages bas niveau. Rien qu'en 2022, le MITRE a répertorié 331 enregistrements CVE[2] basés sur des corruptions de la mémoire. La deuxième raison est due à la connectivité accrue des RTS. La gestion de nouvelles interfaces réseau nécessite des piles réseau complexes, également écrites dans des langages bas niveau, qui peuvent aussi être vulnérables aux corruptions de mémoire[3]. Comme ces piles sont utilisées par de nombreux systèmes, la moindre vulnérabilité dans l'une d'entre elles pourrait mettre en péril de nombreux systèmes, ce qui en fait des cibles très intéressantes pour un attaquant.

Thèse

Dans cette thèse, nous présentons nos travaux concernant la sécurité des systèmes temps réel. Plus précisément, nous avons étudié une protection appelée Intégrité du flux de donnée (*Data-Flow Integrity* ou DFI en anglais) qui existait déjà pour les programmes génériques et nous l'avons adaptée pour prendre en compte les contraintes et caractéristiques des systèmes temps réel. Cette thèse est composée de cinq chapitres : un chapitre qui introduit le contexte, trois chapitres contenant nos contributions, et une conclusion. Le chapitre 2 présente une vue d'ensemble sur les systèmes temps réel, les attaques et les protections contre les corruptions de mémoire ainsi qu'un état de l'art sur la sécurité des systèmes temps réel. Le chapitre 3 présente notre implémentation et l'analyse du surcoût

[2]Common Vulnerability Exposure, une base de données qui répertorie les vulnérabilités présentes dans les programmes couramment utilisés

[3]comme la série de vulnérabilités URGENT/11 sur la pile TCP/IP de VxWorks, un système d'exploitation temps réel

de la protection DFI. Le chapitre 4 présente notre première contribution qui optimise certaines parties de la protection DFI pour réduire son surcoût le long du Chemin de Pire Temps d'Exécution. Le chapitre 5 présente une deuxième contribution qui optimise également le DFI en supprimant des redondances dans les instrumentations de la protection. Enfin, le chapitre 6 conclut cette thèse en résumant nos contributions et en proposant des travaux futurs pour poursuivre ces recherches.

PUBLICATIONS

This thesis is based on the following paper:

Nicolas Bellec, Guillaume Hiet, Simon Rokicki, Frédéric Tronel, Isabelle Puaut. *RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems*, in ECRTS 2022 - 34th Euromicro Conference on Real-Time Systems.

During the beginning of this thesis, we also published another paper based on a work done during the master internship. This paper is not discussed in the rest of this thesis:

Nicolas Bellec, Simon Rokicki, Isabelle Puaut. *Attack Detection Through Monitoring of Timing Deviations in Embedded Real-Time Systems*, in ECRTS 2020 - 32th Euromicro Conference on Real-Time Systems.

Chapter 1

Introduction

Many systems need to respond to specific events before a strict deadline to ensure their correct behavior. Planes and cars are examples of systems with these requisites. These systems are now equipped with many sensors and must be able to respond fast enough to the commands they receive in order to ensure the safety of their users as well as other people around them. Such systems are labeled **Real-Time Systems** (RTS) as they must be designed to ensure that they react before their deadlines to the coming events in all circumstances. To guarantee that no deadline can be missed by the system, RTS are validated before being deployed.

To ease this validation, RTS have historically used isolated communication channels (typically a Controlled Area Network bus) as it is easier to analyze the latencies of such channels, making them more predictable. However, there is an emergence of new RTS using wireless communications to receive commands, transmit data and communicate with other systems. A good representative of these new systems are flying drones. A flying drone is intrinsically a RTS as it must always adapt its propulsion in function of its environment. It can be controlled at a distance, used to capture images that must be sent back to the user and finally, there is a trend in using fleets of drones capable of organizing themselves to complete a mission. In these situations, wireless communications are essential to communicate with the user (either to receive commands or send data) and between the drones.

These new communication channels are also a great opportunity to attack these systems. With classical RTS, as the communications only happen in a closed and controlled environment, attacking a system already meant being able to communicate with it, which was hard enough to prevent many attacks. With these new RTS, an attacker just has to be close enough (e.g. at the range of the Wi-Fi or Bluetooth of the system) to start a communication and potentially attack the system. Thus, the security of RTS is becoming more and more important as these systems are massively being deployed.

1.1 Real-Time System characteristics impacting the security

Let's describe some characteristics of RTS that impacts their security. First, RTS are often written in **low-level languages** (typically C) that require the developers to manually manage the memory. These languages are prone to memory errors that can be exploited to attack the programs. By using memory errors, an attacker can break the guarantees of the system by modifying parts of the memory he/she should not be able to. This can lead to a wide range of issues impacting the system. For example, an attacker could crash the system or execute its own code on it, preventing the system from functioning correctly.

RTS are also often **hard to update** as modifying the system requires to re-validate that no deadline can be missed by the system. Furthermore, patches are often harder to deploy onto the system as RTS can operate for long periods of time during which they can not be updated. This means that the window of time between the discovery of a vulnerability and the application of an eventual patch can be very long. The system must remain safe during that period of time.

Finally, RTS requires to be **predictable** in order to guarantee the deadlines of the system. This means that any defense deployed to protect the system must also be predictable to ensure that we can still provide these guarantees. In particular, it prevents the deployment of many modern defenses that use randomness to prevent the attacker from finding crucial information to mount his/her attack[1].

1.2 Attacker model

The manual management of the memory combined with the predictability of RTS makes them very vulnerable against a common class of vulnerabilities called **memory corruption vulnerabilities**. The vulnerabilities in this class uses an error the memory management to writes and read at unexpected places in memory, modifying the behavior of the program or corrupting important data. Such vulnerabilities are already very common as shown by MITRE that listed 331 CVE records[2] based on memory corruption in 2022

[1] Such as the Position Independent Executable (PIE) combined with Address Space Layout Randomization (ASLR) that requires a high entropy to effectively protect a program [SPP+04], which makes cache analyses very pessimistic [FGG18]

[2] Common Vulnerability Exposure, a database that lists vulnerabilities found in commonly used programs

alone. Furthermore, the increased connectivity of RTS requires complex network stacks that are also written in low-level languages and can also be vulnerable to memory corruption[3]. As these stacks are used by many systems, any vulnerability in one of these could endanger many RTS, making them very interesting targets for an attacker.

In this thesis, we wish to improve the security of real-time systems against memory corruption vulnerabilities. We focus our work on memory corruption based on a software attack (i.e. without the attacker physically attacking the system) as these attacks are the most likely to affect many RTS. Thus, we suppose that the attacker found a software error allowing him to corrupt the memory, i.e. that he/she is able to write in an unexpected part of the memory. The attacker can then use this error to modify an arbitrary point in memory.

As an example, the attacker could have found a buffer overflow that allows him to write paste the bounds of a buffer and modify other variables on the stack. The attacker could then use this vulnerability to modify important variables in the stack and modify the program behavior.

1.3 Thesis

In this thesis, we present our work regarding the security of real-time systems. More specifically, we studied a protection called Data-Flow Integrity (DFI) that already existed for general programs [CCH06] and we adapted it for real-time systems.

We first analyze the overhead of this protection on the Worst-Case Execution Time (WCET) to establish where we should focus our optimizations. This analysis shows that three parts of the DFI are responsible for most of the overhead of this protection. The first responsible part verifies that the data loaded by the program is not corrupted at runtime. The second part computes addresses to a special table containing metadata used by the first part. Finally, the third part ensures that this table and the program code can not be modified by the protected program.

Knowing which part of the DFI are the most important sources of overhead, we first focus our effort to reduce the overhead of the first part on the WCET. To do so, we combine an Integer Linear Programming solver with data retrieved when estimating the WCET, reducing the overhead of DFI by a mean 7.6% in our experiments. This contribution was

[3]Such as the URGENT/11 set of vulnerabilities on the real-time operating system VxWorks' TCP/IP stack

published in the 34th Euromicro Conference on Real-Time Systems.

We then tackle the overhead of the two other parts by detecting redundant computations in consecutive uses of these parts. We present a method that reduces these redundancies at the basic-block level using available registers. This method presents a mean 14.4% improvement in our experiments.

This thesis is composed of five chapters: one chapter that introduces some background, three chapters containing our contributions, and a conclusion. Chapter 2 presents an overview regarding real-time systems, memory corruptions attacks and defenses as well as a state-of-the-art about security of real-time systems. Chapter 3 presents our analysis of the overhead of Data-Flow Integrity. Chapter 4 presents our first contribution that optimizes one part of the DFI to reduce its overhead along the Worst-Case Execution Path. Chapter 5 presents a second contribution that also optimizes the DFI by removing redundancies in the protection instrumentation. Finally, Chapter 6 concludes this thesis by summarizing our contributions and providing future works to continue this research.

Chapter 2

Background & State of the Art

Chapter overview

In this chapter, we present some background on real-time systems and security against memory corruption attacks. We first present real-time systems and how their properties are guaranteed. We then present memory corruption attacks and the protections developed against these attacks. Finally, we present a state of the art of the security for real-time systems.

2.1 Real-Time Systems

Real-time systems (RTS) are systems composed of multiple **tasks** with constraints on the execution time of each task. A **deadline** is associated to each task and represents how much time the task has to finish. Such systems are present in many parts of our daily life, from a simple controller on a washing machine to the system controlling a satellite or a power plant. Ensuring that every task in the system respects its deadline is thus very important as any deadline miss could compromise the safety of the system (e.g. a satellite missing a deadline on its navigation task could end up colliding with debris or another satellite). To guarantee the respect of every deadline, RTS are subject to a temporal evaluation. First, the maximum execution time, called the **Worst-Case Execution Time** (WCET), of each task is estimated in isolation. The WCETs are then used to verify that, given a specific scheduling policy, the system will meet all its deadlines.

In this section, we first present the different methods to establish the WCET of the tasks in a system in Subsection 2.1.1. We then dive deeper into the static analysis method in Subsection 2.1.2, as this is the one we use in the rest of this thesis. Finally, we present how the WCETs are used to verify the schedulability of the system in Subsection 2.1.3.

2.1.1 WCET

To ensure that a task has enough time to finish before its deadline, its WCET is estimated first. The WCET of a task is the maximum time it takes for that task to execute on the system (in particular, on the system's hardware) when running in isolation (i.e. without the interference of other tasks).

Knowing the exact WCET of a given task is often impossible as it requires to analyze all the possible executions of the program on the system's hardware. As the number of states of the program and of the hardware increases exponentially with the size of the program, such analysis is almost never possible, especially with the extremely complex hardware architectures that we use nowadays. Thus, the WCET analysis aims at estimating an upper bound of the WCET that is as tight as possible to the real WCET (that we do not know). We search for an upper bound of the WCET to ensure that a task cannot run more than its estimated WCET in any circumstance. This property ensures that we can use the estimated WCET to verify the schedulability of the system.

In Figure 2.1, we show an example of the distribution of the execution time of a task. A first point of observation is that the execution time may vary greatly between two executions. We also see that the distribution is bounded by an upper value, 11s in this example. This value is the real WCET. However, as we do not know this distribution perfectly when analyzing a task, we have to estimate this value. In our example, we show three possible estimations. The first one (9s) is not an upper bound of the real WCET and is thus an unsafe estimation of the WCET. Such estimation, can lead to false guarantees that the system meets all its deadlines. The second (12s) and third (15s) values are upper bounds of the real WCET and are thus safe estimations of the WCET. These values can both be used to ensure that the system meets its deadlines. However, the 15s value is more pessimistic than the 12s value, which can lead to pessimism in the schedulability analysis. In particular, the analysis may respond that the system is not schedulable with the 15s value while it would be with the 12s value. Thus, we also want the WCET analysis to return an as tight as possible estimation of the WCET (i.e. an estimation that is close to the real value).

Numerous tools have been proposed to estimate the WCET using different methods. The survey [WEE+08] presents many such tools. We can split the methods in three categories: measure based estimation, static analysis based estimation and hybrid estimation.

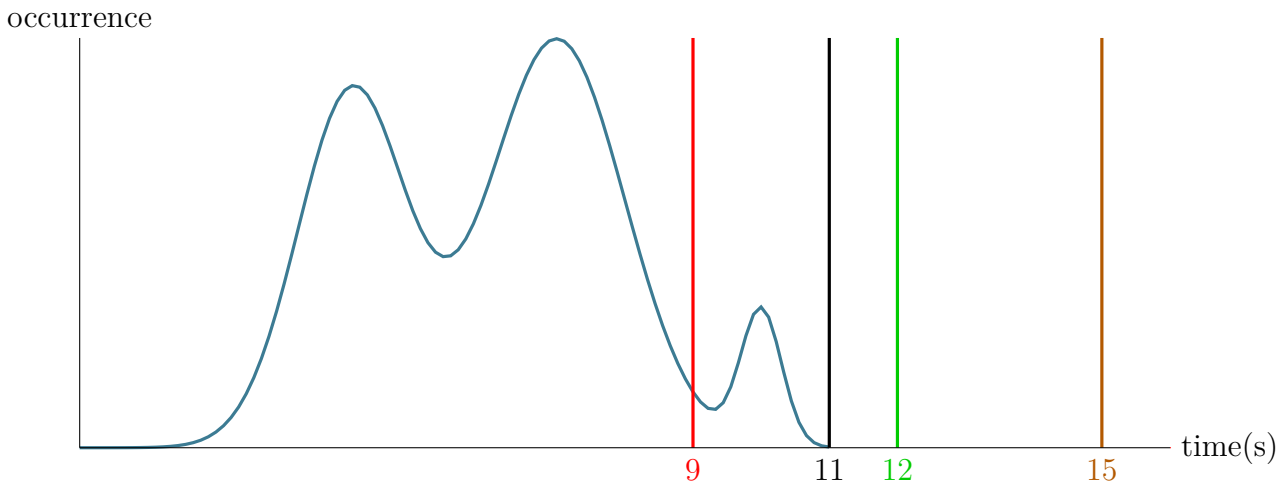


Figure 2.1: Illustration of the real and three estimated WCETs for a given execution time distribution of a task

2.1.1.1 Measure based estimation

Measure based estimation uses measurement of the execution time of a task on the system to estimate the WCET of the task. By measuring numerous execution times (with many distinct inputs), these methods try to retrieve the execution time distribution of the task. In particular, such methods do not need to be given a hardware model of the system to estimate the WCET of the task. However, as these methods only have access to an estimation of the distribution, they are not sure to capture the real WCET in their measures. A common way to compensate this risk of under-approximation is to use Extreme Value Theory (EVT). This theory estimates the risk that a rare event happens based on previous events. For example, EVT can be used to estimate the risk of an earthquake of magnitude higher than a given threshold happening in a region, knowing the history of seismic activity of the region. This theory is based on a given number of hypotheses that must be verified to use it. Reghenzani et al. [RMF+19] worked on verifying that the hypotheses of EVT are experimentally valid. The tightness of the estimated WCET is also important to guarantee the schedulability of the systems. To this regard, Vilardell et al. [VSM+22] presented a method using Markov's inequality and power-of-k functions (functions of the form x^k) to reduce the pessimism of methods using EVT.

2.1.1.2 Static analysis based estimation

To the opposite of measure based estimation, static analysis based estimation uses a model of the system's hardware and a static analysis of the task to estimate an upper-bound of the WCET. The main downside of these techniques is the necessity to provide a model of the system's hardware, which can be very complex for modern hardware architectures. This can lead to a more pessimistic WCET estimate which can later hurt the schedulability analysis. On the other hand, if the hardware model and the analysis are correct and provide an over-estimation of the behavior of the real system, static analysis ensures that the estimated WCET is an upper-bound of the real WCET. This is in particular very important for critical systems (such as in power plants or in avionics) in which failing to meet a deadline could cause a catastrophe.

Many tools have been developed to statically estimate the WCET. AiT [FH04] is an industrial close-source estimator with many hardware models. However, as the tool is closed-source, researchers cannot implement new methods or hardware models for it. To test new analyzes, other estimators such as Heptane [HRP17] and OTAWA [CS06] exist. These estimators are open-source and with a particular focus on some specific analyzes used to estimate the WCET.

2.1.1.3 Hybrid estimation

Hybrid estimation associates static analysis based estimation and measure based estimation to estimate the WCET. The main idea is to measure many program executions and to derive the probability distribution of the execution time for blocks of code. These data are then fed to a static analysis method that searches for the path with the highest execution time. This provides better guarantees than measure-only estimation while not requiring to provide a hardware model.

Bernat et al. [BCP03] proposed a probabilistic framework for hybrid estimation based on measurement of the execution time of the basic-blocks[1]. This led to multiple works that have been summarized in a survey by Cazorla et al. [CKM$^+$19].

Bonenfant et al. [BCD$^+$17] proposed to use machine learning to derive an early estimation of the WCET of a piece C source code. Machine learning have also been used by Amalou et al. [APM21] to estimate the WCET of basic-blocks at the binary level and

[1]Basic-blocks are sequences of instructions such that the only way to branch in the sequence is at the entry and the only way to branch out of the sequence is at the exit.

then to combine these estimations into a WCET estimation of the whole task using static techniques.

2.1.2 Static estimation of WCET

In the rest of this thesis, we focus on static estimation of the WCET as this method also provides very useful information about the task under estimation. These data can typically be used to optimize the task in order to reduce its WCET. In this part, we present the different analyses that are often used when statically estimating the WCET. By abuse of notation, as we almost never have access to the real WCET, we use the term WCET to refer to the estimated WCET in the rest of this document.

We present in Figure 2.2 a common sequence of analyzes performed by a WCET estimator. The WCET estimation is performed on the binary of a program rather than its source code to ensure that we know precisely which instructions are executed as well as the memory layout of the program under analysis. This is particularly useful to obtain a tight result as the WCET analysis depends on the micro-architecture.

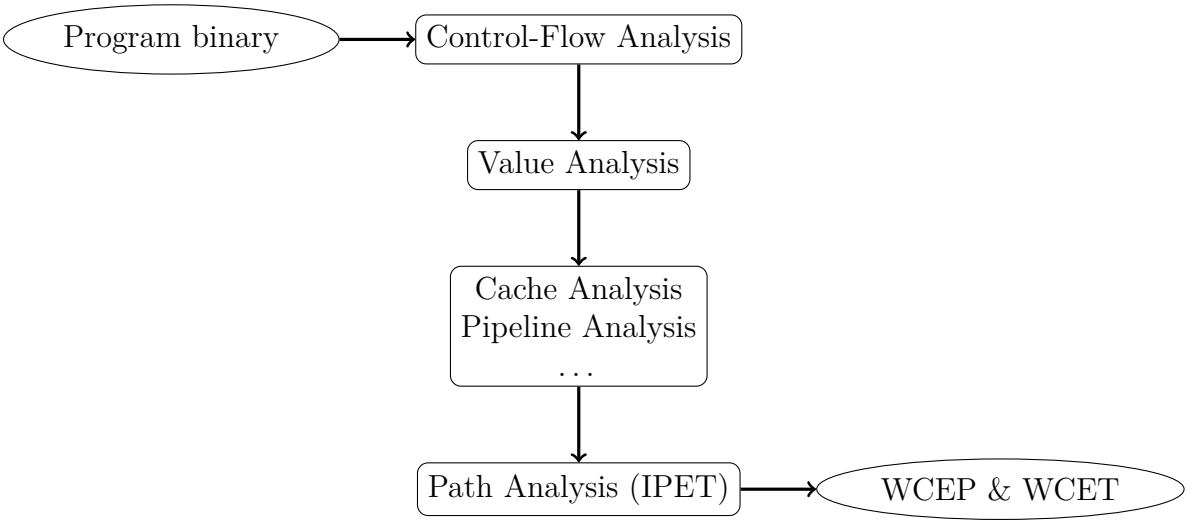


Figure 2.2: Pipeline of analyzes for WCET estimation

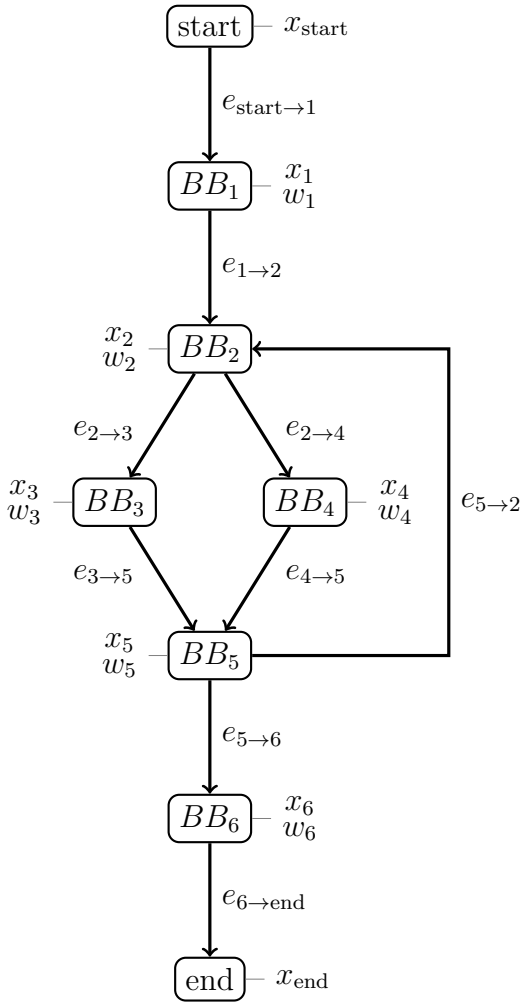
A common approach for static WCET estimation is to estimate the WCET of each basic-block and then find the longest path in the program with the WCET of each basic-block as weight. This requires to first perform a **Control-Flow analysis**. This analysis retrieves the structure of the program from its binary representation and determines a

superset of the possible paths in the program. In particular, this analysis retrieves the functions, the basic-blocks and the edges between the basic-blocks as well as the function calls and loops.

Another important analysis to estimate the WCET and whose results can be used to optimize the task is the **Value analysis**. This analysis estimates the value in each memory cell and each register used by the task. This is a particularly important analysis to detect infeasible paths in the program, loop bounds and is further used by other analyzes such as cache analysis. For example, if the value analysis is able to detect that a given register is equal to five for a conditional jump and the jump is only taken if the value of the register is less than three, then we can ensure that the jump cannot be taken, removing a path for the other analyzes (in particular, the search of the longest path). To improve their precision, value analyzes often provide multiple results on the same part of the code in different contexts when this part can be reached multiple times [TSH+03]. For example, in a loop, there might be a context for the first iteration of the loop and another for subsequent iterations, allowing the analysis to have more precise results on the first iteration and collapsing the results of all other iterations in the second context results. This is particularly useful to obtain a good trade-off between the analysis cost (in terms of time and memory) and its precision.

Finally, another family of analyzes that is crucial for the static estimation of the WCET is the micro-architectural analysis family. By using abstract timing models of the system's architecture, the analyses of this family deduce the possible set of states of the architecture when executing the basic-blocks of the program, to deduce the WCET of each basic-block in the different contexts. This family typically regroups the **Cache Analysis** and the **Pipeline Analysis**. The cache analysis aims at determining the state of the caches and, in particular, at detecting the cache misses and cache hits that could occur and impact the execution time of the task [FW99]. The pipeline analysis [LTH02] tries to establish the pipeline behavior. In particular, we are interested in knowing the number of cycles required to execute a basic-block, taking into account the latencies induced by memory accesses or conflicts of resources inside the processor. Other analyses can be needed with complex processors containing features such as branch prediction. In this case, specific analyses that estimate the impact of such features on the estimated WCET of a basic-block are required to obtain a tight upper-bound on the WCET for each basic-block.

Once the WCET of each basic-block is estimated, these estimations are combined to estimate the overall WCET of the task. This last analysis, called **Path Analysis**, searches



(a) CFG for the IPET

$G = \max_{x_i \in \mathbb{N}} \sum w_i x_i$	Goal
$x_{\text{start}} = 1$	Structural constraints
$x_{\text{end}} = 1$	
$e_{start\to 1} = x_{\text{start}}$	
$x_1 = e_{\text{start}\to 1}$	
$e_{1\to 2} = x_1$	
$x_2 = e_{1\to 2} + e_{5\to 2}$	
$e_{2\to 3} + e_{2\to 4} = x_2$	
$x_3 = e_{2\to 3}$	
$x_4 = e_{2\to 4}$	
$e_{3\to 5} = x_3$	
$e_{4\to 5} = x_4$	
$x_5 = e_{3\to 5} + e_{4\to 5}$	
$e_{5\to 2} + e_{5\to 6} = x_5$	
$x_6 = e_{5\to 6}$	
$e_{6\to \text{end}} = x_6$	
$x_{\text{end}} = e_{6\to \text{end}}$	
$1x_1 \leq x_2$	Functional constraints
$x_2 \leq 10x_1$	

(b) IPET formulation

Figure 2.3: IPET example based on [LM95]

a path in the **Control-Flow Graph** (CFG) of the task with the highest estimated execution time, the so-called **Worst-Case Execution Path** (WCEP). The classical method to do so is with the **Implicit Path Enumeration Technique** (IPET). The IPET encodes the problem of finding the path with the highest execution time as an **Integer Linear Programming** (ILP) problem [LM95]. As an example of this method, we show in Figure 2.3 the CFG (2.3a) of a simple program and the resulting ILP (2.3b). For each basic-block BB_i, we note w_i its WCET and x_i an ILP variable that represents the number of time this basic-block is executed in the WCEP. For each edge between BB_i and BB_j, we note $e_{i\to j}$ an ILP variable that represents the number of time the edge is taken in the WCEP. The goal of the ILP is to maximize $\sum w_i x_i$, that represents the maximal execution time of the path selected by the ILP.

A first set of **structural constraints** are used to encode in the ILP the limits due to the structure of the program. In particular, we encode that each basic-block is executed as many times as an edge enters it in the path and that each execution of a basic-block leads to an execution of one of its successor in the CFG. This is expressed by the following formula: $\sum_{i\in \text{pred}(BB_j)} e_{i\to j} = x_j = \sum_{k\in \text{succ}(BB_j)} e_{j\to k}$. In this set of constraints, we also find limits such as $x_{\text{start}} = 1$ and $x_{\text{end}} = 1$ that express that the task only start and end once (to estimate its WCET).

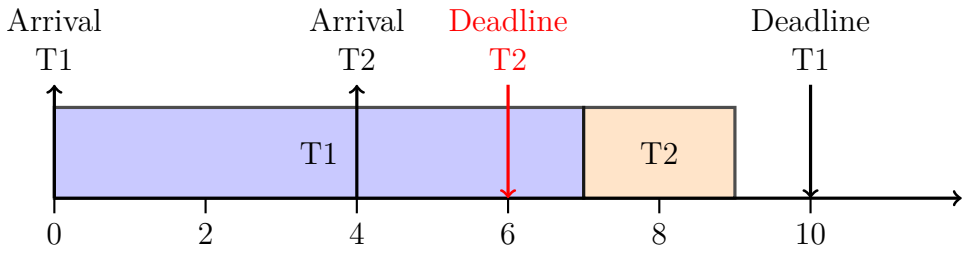
A second set of constraints, called **functional constraints**, encodes information about the functionality of the program. In the example, we can find constraints about the loop bounds of the loop that can only have between 1 and 10 iterations. Encoding loop bounds typically prevents an unbounded ILP problem. This is in this set that we can also find constraints that encode program specific behaviors which can improve the precision of the WCET estimation. For example, if BB_4 could only be executed once in the loop, a constraint $x_4 <= 1$ would be added, potentially reducing the estimated WCET if $w_4 > w_3$.

2.1.3 Scheduling

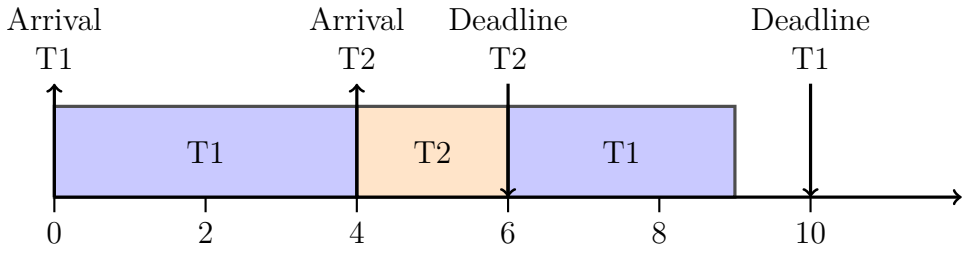
Verifying that the system always respects its timing constraints requires the estimation of the WCET of each task in the system. However, WCETs alone are not enough to ensure that multiple tasks running in parallel won't miss their deadlines. Interferences between the tasks may prevent a task from running, generating a missed deadline. For example, in Figure 2.4, we present two scenarios with the same set of tasks (T1 and T2) but with two different scheduling policies. We suppose that T1 requires seven units of time to execute

while T2 only requires two units of time. With the first policy (Figure 2.4a), the system never preempts a running task, which leads to a deadline miss for the task T2 as this task arrives while T1 is already running. On the other hand, with the second policy (Figure 2.4b), the system preempts a running task when another task has less time to complete, preventing the deadline miss in our example.

These considerations led to many researches on the impact of the schedule and how to ensure the absence of deadline miss for different systems and schedulers. In this section, we only provide a light overview of the existing works as a context for the thesis.



(a) Example of a scheduling generating a deadline miss



(b) Example of a correct scheduling

Figure 2.4: Example of scheduling policies with two tasks T1 (WCET=7) and T2 (WCET=2)

Depending on the characteristics of the system and of the set of tasks, many methods have been proposed to ensure the schedulability of the system. A first class of methods is to find sufficient schedulability conditions that, when verified, ensure that the system is schedulable with a given scheduling policy. Jie et al. [JRZ10] provide a survey on many such conditions, in particular for uniprocessor scheduling.

For example, let $S = \{\tau_1, \tau_2, ..., \tau_n\}$ be a task set with (T_i, W_i) the couple (period,WCET) of τ_i. We suppose that the deadline of a task, i.e. the maximum time allowed for the task to execute, is equal to its period. The **Earliest Deadline First** (EDF) scheduling policy

always executes the task with the earliest deadline. It supposes that the system can preempt running tasks and that the system uses a uniprocessor. In the absence of inversion of priority, typically caused by a task of higher priority (according to EDF) accessing a shared variable reserved by a task of lower priority, the EDF scheduling policy has the following sufficient condition: if $\sum_{i=1}^{n} (W_i/T_i) \leq 1$, then S is schedulable.

Another method to ensure the schedulability of the system is to perform a **Response Time Analysis** (RTA) [JP86]. The RTA consists in estimating the response time of every task in the system, i.e. the time between the arrival of the task (the moment the task is ready to be scheduled) and the time the task finishes its execution, considering the interferences of the other tasks. If, for every task in the system, its response time is lower than its deadline, then the system is schedulable. However, obtaining a safe and tight upper bound of the interferences between the tasks can become very complicated, especially with multiple, heterogeneous cores. This can make the analysis too pessimistic from some schedulable systems.

Finally, the schedulability of the system can be estimated by exploring the space of possible schedules to ensures the absence of deadline misses. This is in particular useful when dealing with multiprocessor systems where the previous methods fail. Guan et al. [GGD+07] proposed to use timed automata to search the schedule space of sets of tasks with static-priority on a multiprocessor system. By transforming the schedulability problem into a reachability problem in a timed automaton, Guan et al. can use well known reachability algorithms to exhaustively search the schedule space and ensure that the system is schedulable. Yalcinkaya et al. [YNB19] extended this idea to non-preemptive self-suspending sets of tasks.

The main problem with these methods are the resources required to execute them (in terms of time and memory). Reachability algorithms for timed automata are known to be PSPACE, which often makes them intractable on large schedulability problems.

Thus, a new method has emerged since 2017 [NB17] to search the scheduling space without relying on timed automata. This method uses **Schedule Abstract Graphs** (SAG) that abstract all the possible execution orders of the tasks. By using such a graph, one can prove (or disprove) that the system may not cause a deadline miss. The main advantage of this method is its computation time. By abstracting and merging some scheduler states (e.g., when the order of scheduling of a subset of tasks does not modify the arriving state), evaluating the schedulability with this method can be much faster than with timed automata, even if the problem remains exponentially complex [RNN22].

2.2 Security of the memory

The apparition in 1988 of the Morris' worm [ano21] has shown the importance of protecting the memory against corruption attacks. These considerations have also started appearing for RTS as memory corruption can break the guarantees on the system. In this section, we present the evolution of software-based memory corruption attacks and protections without considering the real-time properties of a system.

2.2.1 Basic memory corruption attacks and first protections

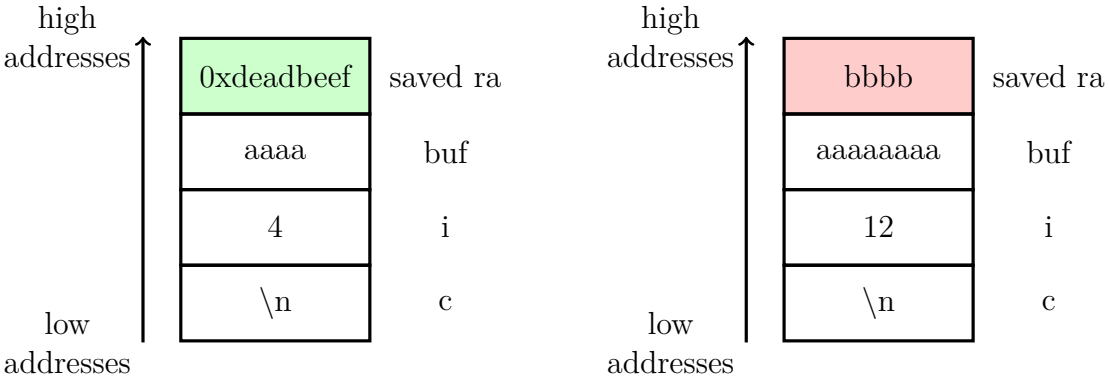
The most classic memory corruption attack consists in writing data past the bounds of a buffer. This attack, called **buffer overflow**, has been described in [One96]. In Figure 2.5, we present an example of a program vulnerable to this attack. This program receives a password and stores it in a buffer of eight characters. In a normal use-case, the memory of the program is presented in Figure 2.5b. The password ("aaaa") is contained in the variable **buf** and the other parts of the stack are not affected. However, when an attacker provides a password longer than 8 characters, the program continues to write past the buffer limits into other variables on the stack. In Figure 2.5c, we present the case of the input "aaaaaaaabbbb" (eight 'a' followed by four 'b'). In this case, we can see in memory that the value of the **saved ra** has been written over by the attacker. As the **saved pc** value is the address to which the program returns when finishing the execution of the function, this means that the attacker can control which instructions the program will execute after this function.

The first method to exploit such a vulnerability is to write the binary code of a small program (typically called a **shellcode** as it is often used to launch a shell) into the buffer. Then, by writing past the buffer, we overwrite the saved instruction pointer (i.e. **saved ra**) with the address of the buffer. When the function returns, the saved instruction pointer is restored and the program jumps to the written code and executes it.

To protect against such attacks, a first protection that appeared was to prevent writable parts of a program from being executable. This protection is often referred as **W⊕X** (write xor execute) and prevents directly writing the code you want to execute into the buffer before jumping on it. Another common protection is to randomize where parts of the program are loaded. This protection, called **Address Space Layout Randomization** (ASLR), was first used to randomize the start address of the stack and the libraries and thus prevent the attacker from finding where he/she should return to execute its

```
void get_password() {
 char buf[8];
 char c = recv();
 for (int i = 0; c != '\n'; ++i ) {
  buf[i] = c;
  c = recv();
 }
 ...
}
```

(a) C code with buffer overflow



(b) Memory state with normal input ("aaaa") after the loop

(c) Memory state with invalid input ("aaaaaaaabbbb") after the loop

Figure 2.5: Example of a buffer overflow vulnerability

shellcode. However, other techniques appeared to bypass these protections.

2.2.2 Code reuse attacks and protections

Instead of writing the code to execute into the memory and then execute it, we can use code that already exists in the program. In practice, most programs loads a standard library (typically the *libc*) which contains functions to launch new processus (e.g. *execve*). By jumping to one of these functions, we can easily obtain an arbitrary execution of code (by launching a shell for example). A generalization of this method is to use small snippets of code that finish by a return instruction (called **gadgets**) and to link them together to execute arbitrary code. As there are many such gadgets in most binaries, finding the right gadgets to execute an arbitrary sequence of instructions is often possible.

This generalization is called **Return-Oriented Programming** (ROP) and remains one of the main methods to use a memory corruption vulnerability inside a program [CW14] [vdVAS+17].

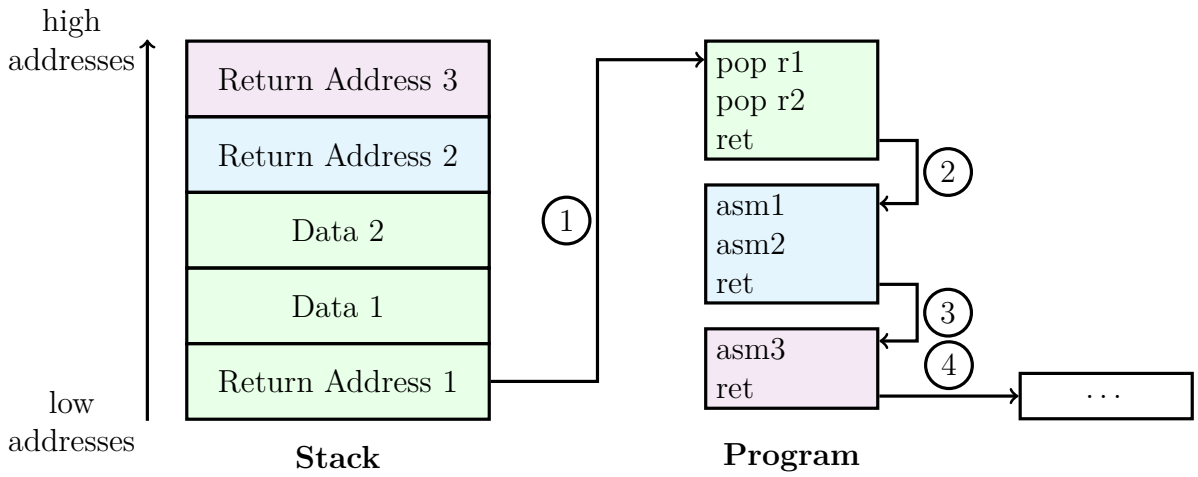


Figure 2.6: Example of Return-Oriented Programming

In Figure 2.6, we present an example of ROP. The attacker overwrites the stack such that a first return address ("Return Address 1" in our example) overwrites the return address of the function with the rest of the stack setup by the attacker as in the figure. When the program returns, it first jumps to the return address placed on the stack to a first gadget. In the example, this gadget loads some data placed by the attacker onto the stack into registers. The program then returns again, this time jumping to the gadget pointed by "Return Address 2". It then executes a few instructions and returns again, this time to the address pointed by "Return Address 3". As we see here, the attacker can ultimately link as many gadgets as he/she needs to create its own program inside the program.

A first protection against such attacks is to prevent the attacker from knowing the addresses of the gadgets in memory. This protection, called **Position Independent Code** (PIE), extends the ASLR protection to the program code and not just the libraries. However, it requires to re-compile the program such that the whole program can run whatever the address at which it is loaded. Each time the program is launched, this protection loads the program in a randomized part of the virtual memory such that the attacker can not predict the addresses of the gadgets. The main problem of this protection is that any data leak about the address of a function (inside the code) can allow the attacker to retrieve the address at which the program has been loaded, which breaks the

protection (note that ASLR has the same issue).

To further increase the difficulty for the attacker to find gadgets, finer grain randomization have been proposed such as **Selfrando**[2]. This protection randomizes the addresses of the functions in the program at each execution. As with PIE and ASLR, this prevents the attacker from predicting the addresses of gadgets. However, as each function address is randomized separately, the attacker needs to leak much more addresses to find enough gadgets to mount its attack.

Another protection against memory corruption, especially buffer overflows, is the use of a **stack canary**. This protection inserts a value, randomized at each program execution, at each function call between the stack frame (containing the local variables) and the saved registers. Just before the function returns, this value is checked to ensure it has not changed. If the value has changed, then the value of the stored registers cannot be trusted and an exception is raised. However, an attacker can easily bypass this protection either by leaking the value of the canary to overwrite the canary with the correct value, or by overwriting directly the registers without modifying the canary (with a well modified pointer for example).

To provide a stronger protection against ROP, Abadi et al. [ABE+05] presented the Control-Flow Integrity (CFI) and the shadow stack protections. The shadow stack protection stores the return addresses of called functions in a so-called *shadow stack*. When the function returns, the return address in the stack is compared with the return address in the shadow stack to verify that they are the same. If it is not the case, a memory corruption is detected and an exception triggered. This protects against ROP but still allows attacks that modify function pointers (e.g. to call *execve*). On the other hand, CFI ensures that when calling a function, the program does not deviate from a control-flow graph that have been extracted at compile time. In particular, this means that an attacker that modifies a function pointer cannot redirect the control-flow wherever he wants but is forced to choose among a restricted set of functions. By using both CFI and a shadow stack, the goal is to prevent an attacker from exploiting a memory corruption vulnerability in the program by preventing any illegal jump.

CFI has two main weaknesses that can be exploited to bypass it. First, as Evans et al. showed [ELO+15], the precision of the control-flow graph is extremely important to obtain a good protection. The control-flow graph often contains imprecision, containing paths that do not exist in the real program (especially due to the precision of pointer/alias

[2] https://github.com/runsafesecurity/selfrando

```
char buffer[12];
char c = 255;
int height = getHeight();
for(int i = 0; c != 0; i++) {
 c = recv();
 buffer[i] = c;
}
if (height >= 10000)
 crash_plane()
```

Figure 2.7: Example of program vulnerable to a Data-only attack

analyzes). An attacker can use these paths to bend the control-flow and carry its attack while remaining undetected by the CFI. The second weakness of CFI is the apparition of new attacks that exploit memory corruption vulnerabilities without modifying the control-flow [CXS+05]. These attacks target the data-flow of the program instead of the control-flow and can thus remain undetected even with the best CFI.

2.2.3 Data-flow attacks and protections

Data-flow attacks modify values in the memory of the program that are later used by the program to take a decision. For example, the attacker could modify a loop counter such that the loop iterates more time than it is supposed to. This can then lead the program to read a cryptographic key that the attacker can later use to break into the system.

A more concrete example is present in Figure 2.7. In this example, supposedly placed in an airplane, the program receives a number of characters that are placed on a buffer of size 12. As the size of the received data is not checked, it is easy for an attacker to overflow past the buffer and to overwrite the saved instruction pointer. However, if a protection such as CFI protects the program, overwriting the instruction pointer would be easily detected. On the other hand, the attacker could also just modify the **height** variable with its buffer overflow, remaining undetected by the protection while forcing the program into the nefarious function **crash_plane**.

To protect the systems against data-flow attacks, new protections that ensure the integrity of the data-flow have been proposed. Castro et al. [CCH06] proposed the **Data-Flow Integrity** (DFI) protection that instruments store and load instructions in the program to verify at each load instruction that the loaded data has been written by a

store instruction that is supposed to write it (based on a statically analyzed data-flow graph of the program). In the rest of this document, we use the term *DFI* instead of *DFI protection* for simplification purpose. Akritidis et al. [ACR+08] proposed **Write Integrity Testing** (WIT). This protection computes for each store instruction in the program a set of objects that the instruction can write into. The protection then instruments the program to ensure that each store instruction writes in one of the object in its set and adds guards between the objects to detect overflows from one object to another. Compared to DFI, WIT trades off the protection for better performances. Indeed, while DFI can detect a large class of attacks including data-only attacks and some information leaks, WIT can only defend against data-only attacks. However, WIT has a time overhead up to 25% while DFI can go up to 150% on some benchmarks. Furthermore, as they are both based on a static analysis of the program to produce their data-flow graph, they can both have an imprecise data-flow graph which could let an attack slip through [DS16] even if this is far more complicated than with CFI.

Other protections have also been developed to obtain memory safety (i.e. a protection that stops all memory corruption exploits) such as CCured [NCH+05], Cyclone [JMG+02] or Softbound [NZM+09]. These protections use *fat-pointers* which are pointers with metadata attached to them to store the bounds of the pointers. However, such pointers tend to prevent binary compatibility and to have a large overhead [SPW+13]. Another notable work in this domain has been the development of CHERI [WWN+15] that aims at using hardware implemented fat-pointers to improve the security of programs without too much overhead.

2.3 Security of real-time systems

With the increase in complexity and the addition of new communication channels (e.g. Bluetooth, Wi-Fi), RTS have become the target of new attacks. Such attacks negatively impact the safety of these systems as each attack potentially breaks the timing and safety properties of the system. As these systems are more and more integrated in our daily life, the risk of attacks on these systems impacts the safety of more and more people. As an example, Millers et al. [MV15] were able to take the control of the brakes of an unmodified car using wireless communication while Carsten et al. [CAY+15] presented multiple vulnerabilities in car systems due to the complexity of these systems. This led to an increased focus on the security of these systems. In this section, we present security

issues and proposed solutions specific to real-time systems.

2.3.1 Real-time system specific vulnerabilities

Real-time systems require to be predictable to obtain a tight estimation of the WCET and to perform the schedulability analysis. However, this increased predictability can be exploited by the attackers to trigger new vulnerabilities.

A first security issue for RTS is the predictability of the scheduling. As the set of tasks running on a RTS is easier to predict than on a general system, it is also easier for an attacker to detect the arrival of specific tasks that he/she wants to target. Chen et al. [CMP+19] presented such an attack as a side-channel in preemptive fixed-priority schedulers that can reveal the arrival time of future jobs. Such a side-channel can be used to launch advanced attacks such as cache side-channel attacks to recover a cryptokey by targeting a specific task. Kwak et al. [KL18] presented a covert timing channel that allows two conspiring tasks in a uniprocessor real-time system to communicate using timing variations. For example, this could be used to extract data (e.g. a cryptokey) from a specific task into another task, although these tasks are isolated in terms of memory.

Many RTS also use sensors to estimate the state of their environment and decide the next step of operation. For example, an autonomous car uses cameras and radars to detect obstacles, signalization and other cars in order to safely move in its environment. An attacker could disrupt the sensors of one such system in order to feed false-data to the system. For example, Cao et al. [CXC+19] presented an attack on cars with lidar-based perception where they could modify the car perception and make the car crash.

Finally, real-time systems are often written in low-level languages such as C or C++ that are known to be vulnerable to memory corruption attacks. The tendency of RTS to not use non-deterministic hardware mechanisms (e.g. virtual memory and MMU) and protections (e.g. ASLR) to improve the predictability of the system tends to ease the work of attackers. For example, the absence of ASLR in a system often means for an attacker that he can brute-force the address space to find interesting addresses that he can use.

2.3.2 Protecting the schedule

To protect real-time systems against attacks using predictable schedule, a first method has been to randomize the schedule. However, as the RTA of the system depends on the schedule, the randomization of the schedule must still ensure the schedulability of the system.

Yoon et al. [YMC+16] presented a protection, called *TaskShuffler*, that randomizes the scheduler while maintaining the schedulability of the system. To ensure the schedulability, the randomizer bounds, for each task τ_i, the maximum time other lower priority tasks can execute instead of τ_i. Vreman et al. [VPK+19] presented an upper-bound on the number of random schedules that can be obtained for a system while maintaining its schedulability. This upper-bounds can be used to evaluate whether randomizing the schedule improves the security or if the configurations can be deduced by the attacker. However, Nasri et al. [NCB+19] showed that schedule randomization can sometime make the system more vulnerable to attacks rather than improving the security by transforming an inherently safe schedule into a schedule where sensitive tasks could appear in a vulnerable order.

Another method, presented by Völp et al. [VHH08], is to modify the scheduler to prevent timing side-channels. By estimating if a blocking thread could leak information and using the idle thread (a thread that does nothing), this protection is able to prevent the leakage of information using timing side-channels. However, it requires a higher budget for the scheduling and is limited to fixed-priority schedulers.

2.3.3 Adapting existing protection to real-time systems

Real-time systems must be predictable to guarantee their timing constraints. This leads to many constraints that prevent the use of general protections against memory corruption attacks as many of these protections either worsen the predictability of the system (such as ASLR or CFI) or require specific hardware (e.g. a MMU). Furthermore, real-time OSes tend to be simplified to ease their timing verification and improve their predictability. In particular, that means that protections often provided by the kernel on general computers are not present in a real-time OS, which further ease the exploitation by the attackers.

Thus, there is a gap between existing protections for general computers and protections fit for RTS. A first method to bridge this gap is to adapt the analyses used to estimate the WCET. Fellmuth et al. [FGG18] proposed to modify the instruction cache analysis to reduce the pessimism of this analysis in presence of artificially diversified software at different levels. Their study focused on three levels of artificial diversity: segment-level (typically ASLR), function-level (the order of the functions in the executable is randomized at runtime, similar to *Selfrando*) and block-level (each basic-block of each function is randomized). By modifying the instruction cache analysis to handle artificial diversity, they try to obtain a more precise analysis while providing an important protection for the program.

Another method to use general purpose protections in RTS is to adapt them to the system at hand. For example, Abad et al. [AvdWL+13] proposed a hardware-implemented CFI with a predictable overhead such that it can be used by RTS while Walls et al. [WBB+19] proposed RECFISH, a CFI specialized for ARM Cortex-R devices that does not rely on process isolation provided by the operating system. Both these approaches modify existing protections (in both these cases: CFI) and adapt them to real-time systems that have different features and requirements. Mishra et al. [MCG21] presented a survey on CFI methods specialized for real-time systems. In this survey, in additions to the CFI techniques already presented, we can also find new methods that rely on using specific characteristics of RTS. For example, Wolf et al. [WFU+12] and Bellec et al. [BRP20] use information on the WCET coupled with the CFG to detect anomalies in the control-flow at runtime via timing anomalies.

Fellmuth et al. [FHP+17] proposed to use information on the WCEP to guide the artificial diversification of real-time tasks. By knowing the WCEP, the protection can diversify the program on this path up to a given budget such that the maximum overhead on the WCET is fixed by the developer. Furthermore, the program can also diversify the basic-blocks outside the WCEP without impacting the WCET by computing a metric called **criticality** [BHJ12] that acts as a timing distance between a basic-block and the WCEP.

Finally, Yoon et al. [YMC+17] and Kadar et al. [KTF19] used the predictability of RTS to protect them. They both learned predictable patterns of how system calls are used by the system. Then, when the system is deployed, they monitor the system calls. If they detect an abnormal pattern, they consider it indicates the presence of an attack.

In conclusion, real-time systems' security is still in its infancy. Almost all the works presented in this section are still very recent (almost all have been presented after 2017) and new vulnerabilities are discovered every year in these systems. It is thus important to provide new protections adapted to RTS that are able to resist against attackers that use more and more sophisticated attacks. This is precisely what we do in this thesis by adapting DFI to real-time systems and by optimizing it to reduce its impact on the WCET.

Chapter 3

Analyzing Data-Flow Integrity costs on Real-Time Systems

Chapter overview

The goal of this chapter is to analyze the impact of **Data-Flow Integrity** (DFI) on the Worst-Case Execution Time in order to target our optimizations presented in the next chapters. We first present how DFI works and which optimizations already exist for it in the literature. We then present our implementation of DFI as no reference implementation exists and we focus our work on real-time systems. Using our implementation, we empirically analyze the cost of this protection on the Worst-Case Execution Time, and we determine which parts of DFI are the most likely to drive this cost. We show that the timing overhead of DFI is mainly concentrated in 3 parts which are the tag check at each load, the computation of the addresses in the **Runtime Definition Table** (RDT) and preventing store instructions from modifying the RDT.

Data-Flow Integrity (DFI) is a protection against memory corruption attacks introduced in 2006 by Castro et al. [CCH06]. DFI attempts to detect when a loaded value has been written by an instruction that should not have written it. Before the protected program loads a value from memory, DFI first loads a metadata (called a **tag**) associated to that value. That metadata represents which instruction stored that value in memory. DFI then checks that this metadata belongs to the set of valid tags for that load. If it does, the value can be loaded by the program. If not, a memory corruption is detected and an exception is raised.

As DFI protects the whole memory, it can tackle advanced attack schemes such as Non-Control Data Attacks [DS17] that use corrupted data without affecting the program's control-flow. This means that DFI can detect a wide range of memory corruption attacks, including the most recent software-only attacks, and help the system react ac-

cordingly to survive these attacks. Furthermore, software-implemented DFI only adds a few instructions before each load and store instruction in the program and does not require help from the system itself. As these instructions are simple instructions (mostly arithmetic instructions and fixed target jumps), a WCET estimator should be able to analyze a program protected with DFI and estimate a coherent WCET without the need for manually provided information. Thus, the main issue to use DFI for real-time systems is its memory and time overhead. Indeed, the time overhead of DFI on the average runtime is known to be important (up to 150% in some experiments [CCH06]). It would not be surprising to have a similar or higher overhead on the estimated WCET which would inevitably prevent the use of DFI.

To establish the overhead of DFI on the estimated WCET and find real-time specific optimizations, we first need an implementation of DFI. However, there is no off-the-shelf implementation available. Thus, we developed our own version based on the description presented in [CCH06].

In this chapter, we start by presenting how DFI works in Section 3.1. In Section 3.2, we explain the implementation choices we made when developing our version of DFI. Finally, we present a cost analysis of our DFI implementation in Section 3.3 where we detect three main sources of overhead for DFI. We use this cost analysis to motivate our contributions presented in the next two chapters.

3.1 Principle of software-implemented DFI

To protect a program, DFI first analyzes and instruments the program at compile time. The instrumentation of the program then dynamically checks that the program behaves correctly based on the performed analysis. If at any time the program does not behave correctly, the instrumentation raises an exception to be handled by the system on which the program is executing. For example, to ensure that the system still respects its schedule, one way to handle such an exception could be to have a deteriorated state where the system only ensures the most critical tasks.

In this section, we present how DFI protects a program and the optimizations proposed in the original paper [CCH06]. We first introduce the static analysis part of DFI in Subsection 3.1.1. We then present the instrumentation part DFI that handles the protection at runtime in Subsection 3.1.2. Finally, we present in Subsection 3.1.3 the existing optimizations for DFI as proposed by Castro et al. [CCH06].

3.1.1 DFI static analysis

Remark. In this thesis, we often use **load** and **load instruction** indifferently as well as **store** and **store instruction** indifferently.

The static analysis used by DFI is based on determining, for each load instruction in the program, which store instructions may have legitimately written the loaded data. To do this, we construct, at compile time, a **Data-Flow Graph** (DFG) which is a bipartite graph where nodes are load/store instructions and an edge between a store and a load represents the fact that this load can read data written by this store. A program example is shown in Figure 3.1 with its DFG presented in Figure 3.2. This example contains the C code of the program (Figure 3.1a), to understand the goal of this code snippet, and a RISCV assembly version of the program (Figure 3.1b) on which the DFG is constructed. In this example, we have some variables and a simple loop that writes received characters into a buffer. In the DFG, we identify each instruction by its type (store or load) and its line number. We also add the variable name on which the instruction operates.

Once the DGF is constructed, we associate to each load instruction the set of store instructions that may have written the loaded data. For example, the variable **c** read in line 7 may be written by two distinct stores : line 2 and line 10.

The example program contains a buffer overflow vulnerability at line 6 of the C code (and line 15 in the assembly code). Indeed, if an attacker sends more characters than the buffer can hold (i.e., 12 characters), then the attacker can write beyond the **buffer** in memory and overwrite other variables such as **height**. However, such a data-flow is not present in the DFG because the **buffer** and **height** variables are distinct and do not alias in the program. This is an example of illegal data-flow that DFI aims to detect.

Since the DFI bases its definition of legal and illegal data-flow on the computed DFG, the DFG must be an over-approximation (i.e., it must contain all) of the program's legitimate data-flow. If this is not the case, then the DFI may raise an exception for a legal data-flow, which would prevent the program from executing normally. However, if the computed DFG over-approximates too much the real program's data-flow (in the worst case, all the loads are connected to all the stores), then illegal data-flows can no longer be detected and the DFI cannot protect the program. An important problem when computing a precise over-approximation of the DFG is the presence of pointers in the program. Since a pointer can point to multiple variables, any store to the pointed address could potentially write any of these variables, over-approximating the real data-flow of the program. To provide a more precise over-approximation, we use a field-insensitive version

```c
char buffer[12];
char c = 255;
int height = getHeight();
for(int i = 0; c != 0; i++) {
  c = recv();
  buffer[i] = c;
}
if (height >= 10000)
  crash_plane()
```

(a) Program C code

```
addi t2, x0, 255
sb t2, 12(sp) # c
call getHeight
sw a0, 16(sp) # height
sw x0, 20(sp) # i = 0
forLoop:
lb t2, 12(sp) # c
beq t2, x0, endFor # c != 0
call recv
sb a0, 12(sp) # c = recv
addi t4, sp, 0
lw t2, 20(sp) # i
add t4, t4, t2
lb a0, 12(sp) # c
sb a0, 0(t4) # buffer[i] = c
lw t2, 20(sp) # i
addi t2, t2, 1 # i = i + 1
sw t2, 20(sp) # i
jal forLoop
endFor:
addi t3, x0, 10000
lw a0, 16(sp) # height
blt a0, t3, else
call crash_plane
else:
```

(b) Program RISCV code

Figure 3.1: Example program

of the Andersen's alias analysis [And94] to reduce the set of potential targets for each memory instruction.

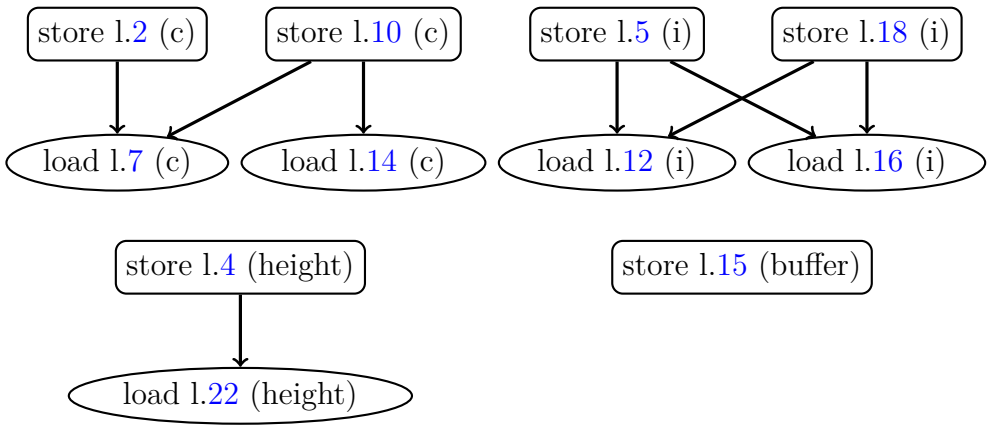


Figure 3.2: Data-Flow Graph of the program in Figure 3.1

3.1.2 DFI instrumentation

Each store instruction is assigned a unique identifier (called a **tag**) and each load instruction is assigned a **valid tag set**, which is the set containing the tag of each store connected to the load in the DFG. The principle of DFI is then to ensure at runtime that each time a load reads a data in memory, the tag of the store that wrote the loaded data belongs to the valid tag set of this load. To protect the program at runtime, the DFI instruments the program using a special table called **Runtime Definition Table** (RDT) combined with a few instructions before each store and load in the program.

The RDT associates fixed-sized areas of memory addresses (typically four bytes) with the tag representing the last store that wrote into this area. In Figure 3.3, we show an example containing a part of memory and the part associated in the RDT. For each four bytes into the main memory (i.e. the memory used by the program without DFI), two bytes are reserved into the RDT. When a store writes into the main memory, it also writes its tag into the associated space in the RDT. In this example, a four-bytes store targeting the address 0x10000 in the main memory writes its two-bytes tag at address 0xB0008002 into the RDT. When a load instruction is executed (e.g. loading four bytes at address 0x10004), the associated tag is retrieved from the RDT (in this example, at address 0xB0008004). The DFI instrumentation then checks if this tag belongs to the valid tag set of the load and raises an exception if it is not the case.

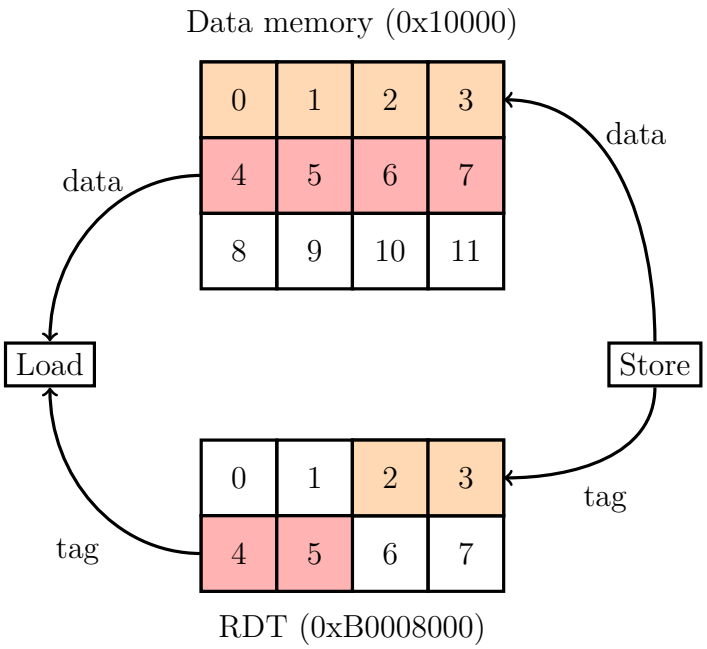


Figure 3.3: Runtime Definition Table

As the whole protection is based on storing and reading tags into the RDT, the DFI must ensure that an attacker cannot corrupt the RDT. Indeed, if an attacker can modify the tag stored in RDT, he/she can associate a valid tag with a corrupted data, breaking the guarantees of the DFI. To prevent this situation, each store in the program is instrumented such that it cannot modify neither the RDT, to prevent its falsification, nor the program instructions, which would also break the protection. To ease this instrumentation, the RDT is placed as the section with the highest address as presented in Figure 3.4. This allows to check that the target address of a store does not belong to the RDT by verifying that the target address is lower than the beginning of the RDT.

Figure 3.5 presents the instrumentation of stores (3.5a) and loads (3.5b) with pseudo-instructions to remain implementation independent. At runtime, the **check_sandbox** pseudo-instruction ensures that the target address of the store (**addr**) is not in the RDT nor the program code. The **rdt_addr** pseudo-instruction computes the address in the RDT where the tag is to be written or loaded. The **store_tag** (resp. **load_tag**) stores (resp. loads) the tag in the RDT at the previously computed address by the **rdt_addr** pseudo-instruction. Finally, **check_tag** checks whether the loaded tag belongs to the valid tag set of the load by comparing the loaded tag with each tag in the set. If the tag

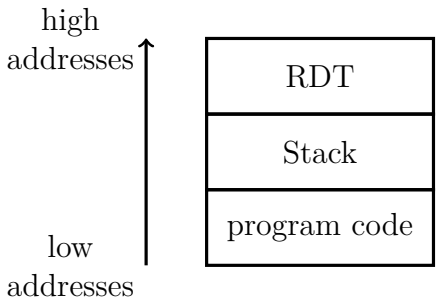


Figure 3.4: Memory layout of the programs protected by DFI

is found in the set, the program continues to execute normally. However, if the tag is not found, an exception is thrown to let the system react to this illegal data-flow. We present how these pseudo-instructions are implemented in Section 3.2.2.

```
check_sandbox(addr)
tag_addr = rdt_addr(addr)
store_tag(tag_addr, tag)
initial store(addr,val)
```

(a) Store pseudo-instructions

```
tag_addr = rdt_addr(addr)
tag = load_tag(tag_addr)
check_tag(tag, valid_tags)
val = initial load(addr)
```

(b) Load pseudo-instructions

Figure 3.5: Instrumentation steps for load and store instructions

3.1.3 DFI optimizations

To reduce the runtime overhead of DFI, some optimizations have been presented by Castro et al. [CCH06]. These optimizations are not real-time oriented but as they all try to remove instructions, they have a similar impact on the WCET as they have on the average case execution time. We present the four main optimizations: **redundant elimination**, **safe store optimization**, **equivalent classes optimization** and **tag check optimization**.

Redundant elimination. The **redundant elimination** detects pairs of successive store/load instructions, in the same basic-block, that have the same target address. By successive, we mean that there is no other memory instruction in between the pair of instructions (but there could be other instructions such as arithmetic instructions). When such a pair appears, one of the instructions has a redundant instrumentation that can be

```
1 store data1, addr
2
3 store data2, addr
```

(a) Unprotected program

```
1 check_sandbox(addr)
2 tag_addr = rdt_addr(addr)
3 store_tag(tag_addr, 3)
4 store data1, addr
5
6 check_sandbox(addr)
7 tag_addr = rdt_addr(addr)
8 store_tag(tag_addr, 4)
9 store data2, addr
```

(b) DFI protected program without redundant elimination

```
1 check_sandbox(addr)
2
3
4 store data1, addr
5
6
7 tag_addr = rdt_addr(addr)
8 store_tag(tag_addr, 4)
9 store data2, addr
```

(c) DFI protected program with redundant elimination

Figure 3.6: Redundant elimination example

eliminated. For example, in the case of two successive stores, the first store tag is never used as it is immediately overwritten by the tag of the second store. Thus, we can remove the parts of DFI that write the tag (**rdt_addr** and **store_tag**) in the instrumentation of the first store. We can also remove the **check_sandbox** part of the second store as the store address has already been verified at the first store and both stores target the same address. An example is illustrated in Figure 3.6 with the code written using pseudo-instructions. Figure 3.6a presents the original snippet of code without DFI. It is composed of two stores targeting the same address. In Figure 3.6b, the first store (line 4) is followed by the second store (line 9), both instrumented. As both store instructions belong to the same basic-block, we can remove the redundant parts of the instrumentation without losing the protection as presented in Figure 3.6c.

Another case where we can apply the **redundant elimination** is when a load follows a store. In this case, the loaded tag is necessarily the tag of the store and thus, the load instrumentation can be removed (as the tag should be in the valid tag set of the load and no exception should be triggered). The last case appears when a load follows a load. The static analysis should find that both valid tag sets are equal (as there is no store in between). Thus, passing the first **check_tag** ensures that the second **check_tag** passes, and we can remove the instrumentation of the second load. Note that removing

these redundant parts does not weaken the protection as the attacker cannot jump in the middle of a basic-block without already breaking the protection. Indeed, to use an attack such as ROP (see 2.2.2) to jump in the middle of a basic-block, the attacker would need to modify a saved register. As such a register is also protected by the DFI and does not alias with anything else, this kind of attack should be detected by the DFI, thus preventing the attacker from bypassing the instrumentation.

Safe store optimization. The **safe store optimization** statically detects store instructions that cannot write into the RDT or on the program instructions. If such store instructions are found, then we can remove the **check_sandbox** pseudo-instruction in their instrumentation, to reduce the overhead of the instrumentation. This is typically the case of stores using the stack pointer and a constant offset. As we can ensure that the stack pointer is not corrupted (because else the DFI would have detected it as the stack pointer aliases with nothing) and the offset is known at compile time, we can be sure that this store does not target an address higher than the beginning of the stack. By placing the RDT after the stack, we can then ensure that such a store won't access the RDT.

Equivalent classes optimization. The **equivalent classes optimization** reduces the cost of checking tags by reducing the size of the valid tag sets. This optimization searches for groups of tags that always appear together in all the valid tag sets of the program. Each group can then be collapsed into a single tag shared by all the store instructions with their tag in this group. This preserves the level of protection of DFI while reducing the size of the valid tag sets, allowing faster tag checks. In Table 3.1, we present an example with three valid tag sets S_1, S_2 and S_3. We note the tags with letters, and we assign to each tag a *tag representation*, i.e. the value used in the instrumentation. The valid tag sets are presented in the first part of the table while the **equivalent class optimization** is presented in the second part of the table. The other parts are used as an example for the **tag check optimization** presented later in this section. In this example, we can see how the **equivalent class optimization** can group two tags (A and B) into a single tag A which reduces the number of tags to check for the valid tag sets. Another advantage of this optimization is to reduce the total number of tags required by the protection, which reduces the required number of byte per tag. In practice, two bytes per tag are sufficient to store all the tags, even for large programs (as shown in [CCH06]).

Valid tag sets				Tag representations
Without optimization (Part 1)				
S_1	$=$	$\{A, B, C, E\}$		$A \to 1$ $B \to 2$ $C \to 3$ $D \to 4$ $E \to 5$
S_2	$=$	$\{A, B, E\}$		
S_3	$=$	$\{A, B, D\}$		
Equivalent classes optimization A∼B (Part 2)				
S_1	$=$	$\{A, C, E\}$		$A \to 1$ $D \to 4$ $C \to 3$ $E \to 5$
S_2	$=$	$\{A, E\}$		
S_3	$=$	$\{A, D\}$		
Greedy tag check optimization (Part 3)				
S_1	$=$	$\{A, C, E\}$	$\sim$ $[\![1, 3]\!]$	$A \to 1$ $D \to 4$ $C \to 2$ $E \to 3$
S_2	$=$	$\{A, E\}$	$\sim$ $[\![1, 1]\!] \cup [\![3, 3]\!]$	
S_3	$=$	$\{A, D\}$	$\sim$ $[\![1, 1]\!] \cup [\![4, 4]\!]$	
Optimal tag check optimization (Part 4)				
S_1	$=$	$\{A, C, E\}$	$\sim$ $[\![1, 3]\!]$	$A \to 3$ $D \to 4$ $C \to 1$ $E \to 2$
S_2	$=$	$\{A, E\}$	$\sim$ $[\![2, 3]\!]$	
S_3	$=$	$\{A, D\}$	$\sim$ $[\![3, 4]\!]$	

Table 3.1: Valid tag sets optimizations - Part 1: Original tags - Part 2: Group A and B into A - Part 3: Tag representation of C and D are modified to improve tag checks - Part 4: Optimal tag representations

Tag check optimization. The **tag check optimization** aims at reducing the impact of the load instrumentation on the runtime. This optimization is composed of two parts. The first part improves the way valid tag sets are checked when they contain consecutive tag representations. For example, the set $\{1, 3, 4, 5\}$ contains the interval $[\![3, 5]\!]$. In this case, we can check if the loaded tag value is between three and five rather than checking if it is three, four or five. We show this optimization in Figure 3.7 for the set $\{1, 3, 4, 5\}$. Figure 3.7a corresponds to the naive way of checking the loaded tag (**tag**) against this set, with one condition for each element in the set. Figure 3.7b shows how to check **tag** interval per interval (considering $\{1\}$ as $[\![1, 1]\!]$). In general, if we note T the loaded tag value and $[\![l, h]\!]$ the interval against which we want to check, we have $T \in [\![l, h]\!] \iff l \leq T \leq h$. This already reduces the number of checks executed at runtime to verify the loaded tag. However, it still requires two checks per interval, one against the lower bound (l) and one against the upper bound (h), except for single-tag intervals. We can further reduce the number of instructions using unsigned integer arithmetic by remarking that $l \leq T \leq h \iff 0 \leq T - l \leq h - l$. By subtracting l and considering the result as an unsigned integer, we just have to verify that $T - l$ is below

$h - l$. If $T < l$, then $T - l$ would underflow and would thus be a positive integer far greater than $h - l$, ensuring that the protection still holds. In Figure 3.7c, we show this final optimization (with a *uint* keyword to represent an unsigned check). This shows that only two checks remain, one against 1 and one against the interval $[\![3, 5]\!]$ transformed into an unsigned check against $[\![0, 2]\!]$.

```
if tag == 1 or
   tag == 3 or
   tag == 4 or
   tag == 5:
 continue
else:
 raise Exception()
```

(a) Naive

```
if tag == 1 or
   3 <= tag <= 5:
 continue
else:
 raise Exception()
```

(b) Interval optimization

```
if tag == 1 or
   (uint) tag-3 <= 2:
 continue
else:
 raise Exception()
```

(c) Single check interval

Figure 3.7: Implementations of the check_tag (tag, [1,3,4,5])

The second part of the **tag check optimization** uses a greedy algorithm that modifies the tag representation associated to each store to reduce the number of checks required to verify the loads. The goal of this algorithm is to minimize the number of distinct intervals composing the largest and the most frequent (in number of loads in the program) valid tag sets. To do so, it associates a score to each distinct valid tag set in the program. This score is computed as follows: the number of tags in the set times the number of loads using this set. Then, the optimization greedily optimizes all the valid tag sets in the decreasing order of their score.

To optimize a set, the algorithm filters out of the set all the tags that already have a new tag representation and then assigns consecutive tag representations to the remaining tags. This places all the tags without representation in a single interval. As the algorithm is greedy, once it decided on a representation for a tag in a given valid tag set, it does not change it later when optimizing another valid tag set, which can lead to suboptimal results. In the example Table 3.1 Part 3, the algorithm first optimizes S_1 and thus assign the consecutive representations 1, 2 and 3 to the tags A, C and E. It then optimizes the set S_2 but there is no tag without representation in this set. Finally, it optimizes the set S_3 and the only remaining tag without representation: D. As the algorithm assigns the tag representations for A, C and E before examining S_2 and S_3, it cannot optimize S_2. However, if we had an algorithm that could optimize all the valid tag sets at once, it would modify the tag representations assigned to A and E to also optimize S_2 and S_3. As an

example, we provide in Figure 3.1 Part 4, the result of an optimal tag check optimization. We can see that, by providing different tag representations, it is possible to check against each valid tag set with a single interval in our example. We further explore this idea of using an improved algorithm in Chapter 4 of this thesis.

3.2 DFI Implementation

To perform our analysis of the DFI overhead, we have to implement our own version of DFI as there exists no reference implementation. Thus, the only reference we have is the description presented in [CCH06] which we tried to follow as much as possible. However, there are two key differences between the original DFI and our own implementation as we want to protect embedded real-time systems. The first difference is the absence of **Memory Management Unit** (MMU) and virtual memory in our implementation compared to the original one. We discuss this difference in Subsection 3.2.1. The other key difference is that we use a RISCV architecture for our implementation where the original DFI uses x86. This difference and its consequences are presented in Subsection 3.2.2.

3.2.1 Absence of virtual memory

The original DFI has access to an MMU and uses virtual memory. These features are used for two reasons. First, the virtual memory allows the original DFI to place the RDT at a fixed address while leaving the first gigabyte of memory for the original program. Second, the MMU is also used in the original DFI to protect the program code and some other sections of the program, preventing their modification by an attacker. In particular, this helps to further reduce the cost of the **check_sandbox** part as it only has to protect the RDT against malicious writes and can rely on the MMU to protect the program code.

However, embedded real-time systems rarely integrate an MMU as the use of virtual memory hurts the predictability of the micro-architecture (in particular cache analyzes) which worsens the estimation of the WCET. Instead, real-time systems often only have access to a permission mechanism, such as a **Memory Protection Unit** (MPU) that can assign read-write-execute permissions to a limited number of memory regions. As these systems do not impact the WCET estimation, they are often preferred in the context of real-time systems compared to a MMU. In the rest of this document, we suppose that we have access to such a mechanism for a few reasons. First, these mechanisms are

already available on many processors used for embedded real-time systems. Second, our implementation can then use the same optimization for the **check_sandbox** part of DFI as the original DFI.

As with the original DFI, we place the RDT in a section at a higher address than the addresses used by the original program, as presented in Figure 3.8. However, this section is directly placed just after the highest address used by the program (most of the time, the top of the stack) to avoid wasting memory space.

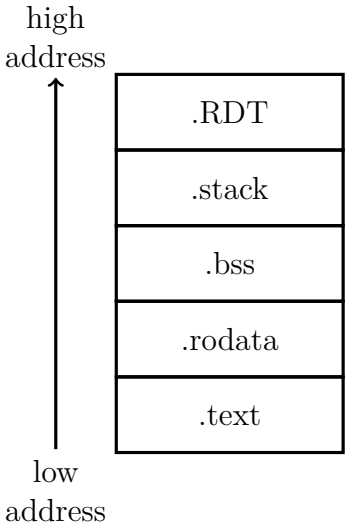


Figure 3.8: Memory layout of the programs protected by DFI, section by section

3.2.2 Using RISCV architecture instead of x86

Another issue with the original DFI [CCH06] is that it was written for x86 architectures. As we focus our work on embedded real-time systems, we want an architecture used by this kind of systems. For example, ARM and RISCV architectures are well suited and adopted for such systems. We choose to use a RISCV architecture as we have a real-time oriented RISCV softcore at our disposal, called RudolV[1], with an FPGA implementation. Furthermore, this softcore is analyzable by AiT [FH04], the WCET estimator that we use in our experiments. Another advantage of the RISCV architecture is the simplicity of its instruction set. In particular, there are very few instructions that can perform loads or stores, which greatly ease the instrumentation step of DFI.

[1] https://github.com/bobbl/rudolv

```
# Place the base RDT address into t3
lui t3, hi(RDT_BASE)
# Substract the offset to RDT
subi t3, t3, off
# Check RDT access
blt raddr, t3, ok
# Ckeck failed, throw exception
ebreak
# Check passed
ok:
```

(a) check_sandbox(**raddr**+**off**)

```
lh t4, t3(0)
```

(b) load_tag(t3)

```
# Place raddr + off into t3
addi t3, raddr, off
# Compute the offset of t3 in the RDT
slri t3, t3, 2
slli t3, t3, 1
# Compute the address in the RDT
lui t4, hi(RDT_NORM)
add t3, t3, t4
```

(c) rdt_addr (**raddr**+**off**)

```
# Prepare tag
addi t4, zero, 2
# Store tag
sh t4, t3(0)
```

(d) store_tag(t3, 2)

```
# Check t4 == 1
subi t4, t4, 1
beq t4, zero, ok
# Check 3 <= t4 <= 5
subi t4, t4, 2
addi t3, zero, 2
bltu t4, t3, ok
# Check failed, throw exception
ebreak
# Check passed
ok:
```

(e) check_tag(t4, {1,3,4,5})

Figure 3.9: Example of a RISCV implementation of the DFI pseudo-instructions (with the optimizations presented in Section 3.1.3)

We now present how we implement the pseudo-instructions presented in 3.1.2 on a RISCV architecture. An example for each pseudo-instruction is presented in Figure 3.9. In RISCV, every memory instruction is composed of the instruction itself (store or load), a register that contains the value to store (or will contain the value after the load), a **base register** and an **integer offset** (a 12-bits signed integer). The target address in memory of the instruction is the sum of the base register and the offset. In this example, **raddr** represents the base register of the protected instruction and **off** is the offset of the same instruction.

We also need two constants linked to the RDT: **RDT_BASE** and **RDT_NORM**. **RDT_BASE** contains the starting address of the RDT section while **RDT_NORM** is a normalized address equal to RDT_BASE − TEXT_ADDR//2 with **TEXT_ADDR** the address of the *.text* section and // the integer division. The normalized address arises to optimize the computation of the addresses in the RDT and is in particular different of the starting address if the program does not start at address 0x0. Indeed, addresses in the RDT are computed as:

$$\text{RDT\_BASE} + ((\text{raddr} + \text{off} - \text{TEXT\_ADDR})//4) * 2$$

This formula maps every four bytes of the program to two bytes in the RDT. However, if we used this formula directly, we would need to subtract the constant **TEXT_ADDR** at each computation of an address in the RDT. Instead, we ensure that *.text* section is aligned (which is mandatory in a RISCV architecture anyway). We can then simplify the computation of the addresses in the RDT by introducing the **RDT_NORM** constant, which is computable at link time. The RDT address formula thus becomes:

$$\text{RDT\_NORM} + ((\text{raddr} + \text{off})//4) * 2$$

To ensure that we are able to instrument all the memory instructions (including the ones that are created due to register scavenging and prologue and epilogue of functions), we reserve two registers (**t3** and **t4**) to be used by the DFI only. This ensures that these two registers are always available and thus, that the DFI instrumentation will not interfere with the program execution (as long as there is no illegal data-flow). Thus, all the pseudo-instruction implementations use these two registers.

Finally, the **check_tag** pseudo-instruction requires a valid tag set of reference (normally obtained with a static analysis). In our example, we choose the arbitrary set {1,3,4,5}.

The **check_sandbox** pseudo-instruction is lowered in RISCV by checking that **raddr** is strictly lower than RDT_BASE−off. This is equivalent to check that the target address is lower than **RDT_BASE** except we do not need to modify **raddr** as we need to use it later for the protected instruction.

In this example, we suppose that the RDT is placed at higher addresses than the memory used by the original program, allowing to check that the target address is not in the RDT by only checking that it is lower than the base RDT address.

The **rdt_addr** pseudo-instruction computes the address in the RDT by computing first an offset based on the target address and then adding this offset to the RDT address. The offset should be computed as such: $((\text{raddr} + \text{off} - \text{TEXT\_ADDR})//4) * 2$ but we reduce this to the computation of $((\text{raddr}+\text{off})//4)*2$ with the help of the **RDT_NORM** constant. The integer division and multiplication is performed using a shift right and a shift left for improved performances.

The **check_tag** pseudo-instruction checks the tag contained in the register **t4** by checking first if this tag is equal to one (by comparing $t4 - 1$ to 0). If the test fails, the code then tests if the tag is contained in $[\![3, 5]\!]$ using an optimization explained in 3.1.3.

Finally, there is a last optimization specific to RISCV architecture to reduce the cost of the **rdt_addr** DFI part. We call this optimization **offset collapsing**. This optimization pre-computes the impact of the integer offset (of the protected memory instruction) on the associated RDT address. If the integer offset (known at compile time) is a multiple of four, then we can transform the computation of **rdt_addr**. In this case, we have $((\text{raddr} + \text{off})//4) * 2 = (\text{raddr}//4) * 2 + \text{off}//2$. As we can pre-compute the off//2 part at compile time, we can add this part as the offset of load/store to the RDT and only compute the raddr//4 * 2 part at runtime. The advantage of doing so is that we can remove the first **addi** instruction of the **rdt_addr** part. Note that if **off** is not properly aligned (i.e. is not a multiple of four), the equality does not hold and we can't perform this optimization. For example, with $\text{off} = 3$ and $\text{raddr} = 1$:

$$2 = ((\text{raddr} + \text{off})//4) * 2 \neq (\text{raddr}//4) * 2 + \text{off}//2 = 0 + 1 = 1$$

3.3 Cost analysis

To optimize DFI for real-time systems, we seek to gain a better understanding of its overhead. Real-time systems are primarily interested in the overhead on the estimated WCET, as the value of the WCET impacts the schedulability of the system. Thus, we performed

two experiments to better understand the impact of the DFI on the WCET and to find out how the overhead is distributed between the different pseudo-instructions that compose the DFI instrumentation (**rdt_addr**, **check_sandbox**, **check_tag**, **store_tag** and **load_tag**).

In this section, we first present the experimental setup of our experiments in Subsection 3.3.1. We then present and analyze a first experiment in Subsection 3.3.2. This experiment establishes the overhead of the DFI on the WCET and is used as a baseline throughout the rest of this document. Finally, we analyze a second experiment in Subsection 3.3.3 where we estimate the cost of each part of the DFI.

3.3.1 Experimental setup

Details for all our experiments are resumed in Appendix A. For our experiments, we use TACLeBench [FAH+16], a real-time oriented benchmark suite. The TACLeBench benchmarks are divided into 5 groups: **app** that contains two real applications, **kernel** that contains small functions such as computation kernels and a binary search, **sequential** that contains benchmarks with large function blocks, **test** that contains programs that challenge WCET analysis tools and finally **parallel** that contains two modified real world parallel applications.

To compile the benchmarks with DFI we use the compilation workflow presented in Figure 3.10. Each benchmark is compiled into an **LLVM Intermediate Representation** (IR) [LA04] using *Clang*, the LLVM compiler. We then analyze the IR with a modified version of the *PhASAR* [SHB19] data-flow analyzer. This analysis assigns to each store a tag and to each load its valid tag set. The result of this analysis is annotated in the IR file. This file is finally fed back the LLVM compiler to produce the executable. We modified the LLVM backend to add the DFI and to perform the DFI optimizations presented in Section 3.1.3. Overall, this required to change about $5,000$ lines of code in *LLVM* and about $1,000$ lines of code in *PhASAR*. We compiled the benchmark programs with the -O1 optimization flag. This compilation flag drastically reduces the number of load and store instructions compared to the -O0 optimization flag as the compiler starts using registers to store local variables instead of always using the stack.

The DFI instrumentation is done at the end of the backend pipeline, just before emitting the assembly instructions as this is the only place where we can ensure that there are no memory instruction that can be further added to the program. Note that between the data-flow analysis (done at the IR level) and the last stage of the backend,

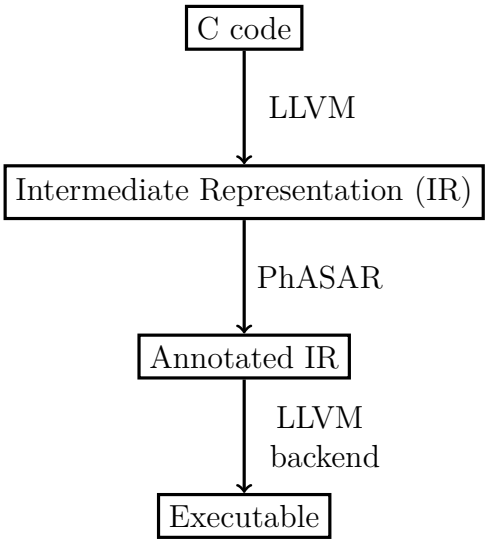


Figure 3.10: DFI instrumentation workflow

some memory instructions can still appear due to the generation of prologue/epilogue of functions or register spilling. We treat these cases as special cases in the backend by reserving a special tag for them. In our implementation, we choose to assign them the tag representation 0.

We target *RudolV*, a 32bits RISCV softcore dedicated to hard real-time systems with the M extension (allowing multiplications and integer divisions). *RudolV* executes each instruction in a precise number of cycles and avoids caches and branch prediction as they worsen the predictability of the architecture.

3.3.2 Overhead of DFI on the WCET

We first aim at estimating the overhead of DFI on the WCET. To estimate the WCET, we use the *AiT* WCET estimator [FH04]. We compute the overhead of DFI by estimating the WCET of an executable without DFI (*baseline*) and of a DFI protected executable (*DFI*). We then compute the normalized overhead as a factor : $\frac{\text{WCET}_{\text{DFI}}}{\text{WCET}_{\text{baseline}}}$. Note that a benchmark with a normalized overhead equal to one means that the DFI do not add any overhead on the WCET for this benchmark.

To estimate the WCET, we only have to provide a few annotations to *AiT*. In particular, we need to provide loop bounds in general, and we need to prevent *AiT* from considering DFI exceptions as a normal behavior for the program. Annotations to ignore

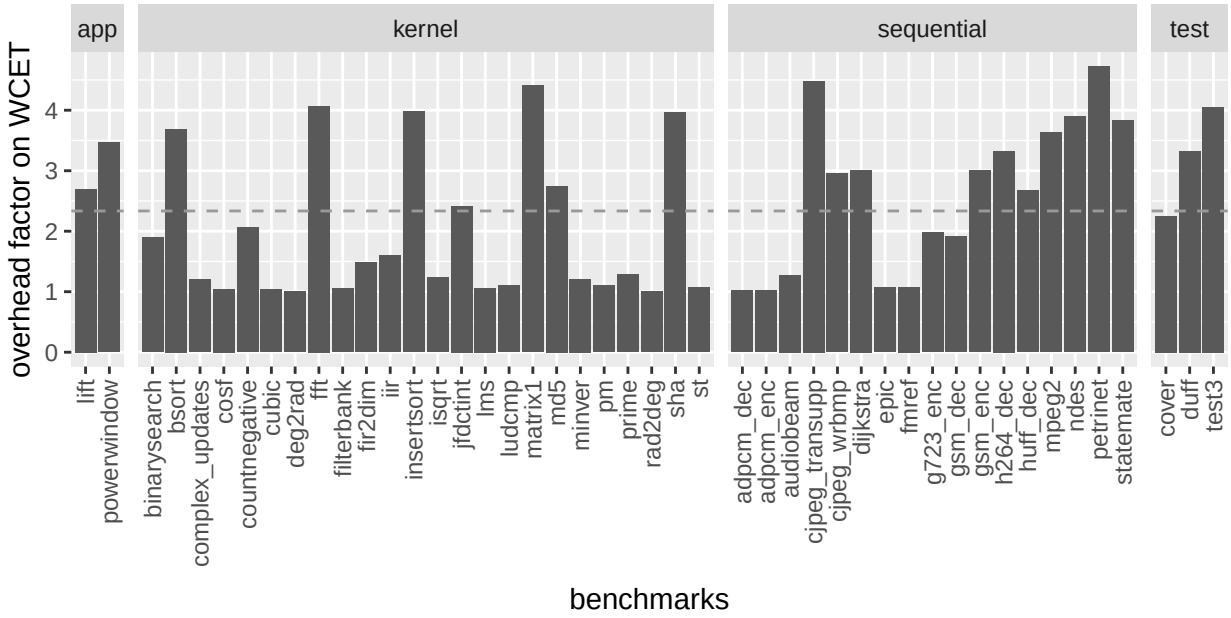


Figure 3.11: Overhead factor of DFI on the WCET using the state-of-the-art DFI of [CCH06] with the mean as a gray dashed line

the DFI exceptions can be automatically generated. On the other hand, loop bounds annotations have to be provided by hand and can be hard to get, especially when compiling with optimizations flags as the compiler may modify these loop bounds between the source code and the resulting executable. This is the main reason why we do not activate more aggressive optimization flags.

We excluded two benchmarks from our experiment (namely *Debie* and *Papabench*) from the **parallel** testbench as they are supposed to run multiple tasks in parallel while we consider a bare-metal execution of the programs without relying on a Real-Time Operating System (RTOS) and thus without supporting parallel execution of tasks. We also excluded the *bitonic*, *bitcount*, *fac*, *quicksort*, *recursion*, *ammunition*, *anagram* and *huff_enc* benchmarks as it was harder to obtain the recursion bounds than loopbounds for the *AiT* WCET estimator with benchmark programs compiled with -O1 optimization flags. Furthermore, *rijndael_dec* and *rijndael_enc* cannot be safely estimated by *AiT* as they violate DFI due to an off-by-one read to a buffer. In both these benchmarks, a custom *memcpy* function copies a buffer into another buffer. However, this function reads four bytes past the end of the buffer just before the end of the function. While these bytes are never used later, this single read triggers the DFI (as the read value is on a saved register in the stack). As *AiT* detects this in its value analysis, it considers that a DFI exception

is triggered and that its WCET analysis is unsafe.

Figure 3.11 shows the normalized overhead factor for the TacleBench benchmarks. 20 out of the 47 benchmarks have an overhead between ×1.03 and ×1.6 while the remaining 27 benchmarks have an overhead between ×1.9 and ×5. The mean overhead is ×2.38. As observed in [CCH06], DFI can incur a high overhead on the protected program, depending on the number of store/load instructions of the program. This confirms the need to reduce the impact of DFI on the WCET.

3.3.3 Overhead per pseudo-instruction

We now search to understand how each part of the DFI instrumentation participates in the overall overhead of DFI. Thus, we want to separate the overall overhead of DFI between each pseudo-instruction of DFI. However, it is hard to get these data using the estimation of WCET. Indeed, the estimation of WCET provides the WCEP, WCET as well as some data on preliminary analyzes used to obtain this estimation. However, the best granularity of information we have access to is at the basic-block level. As multiple load/store can be in the same basic-block, it is very hard to separate the cost of different parts of DFI that belong to the same basic-block. For this reason, we prefer to adopt another method to estimate the relative participation of each part of DFI into the overhead of DFI. Instead of estimating their participation on the WCET, we compute the cost of each part of DFI along an execution trace of the benchmark and measure the overhead of each part instead of a worst-case analysis. Note that, while we cannot normally compare execution obtained through measurement with the estimated WCET, we only seek to better understand which parts of DFI have the most weight in the overhead and thus even an approximation is enough.

In this experiment, we use a RISCV simulator called COMET[2] to obtain an execution trace for each benchmark. Based on these trace, we estimate the cost of each pseudo-instruction for each benchmark.

We know the number of cycles of each instruction as this data is provided by the RudolV architecture. From the compilation phase, we can extract the first and last addresses for each pseudo-instruction. Thus, we just have to iterate on the trace, finding when we are at the starting address of a pseudo-instruction and accumulating cycles into a counter associated to the pseudo-instruction until we reach the ending address for this

[2] https://gitlab.inria.fr/srokicki/Comet

pseudo-instruction. Once we finish iterating the trace, we obtain a counter associated with each pseudo-instruction containing its overhead in cycles. We then compute the relative participation (in percentage) of this pseudo-instruction to the overall DFI overhead by computing the ratio between the counter of the pseudo-instruction and the sum of all the counter.

We present in Figure 3.12 the median overhead percentage across all benchmarks for each pseudo-instruction.

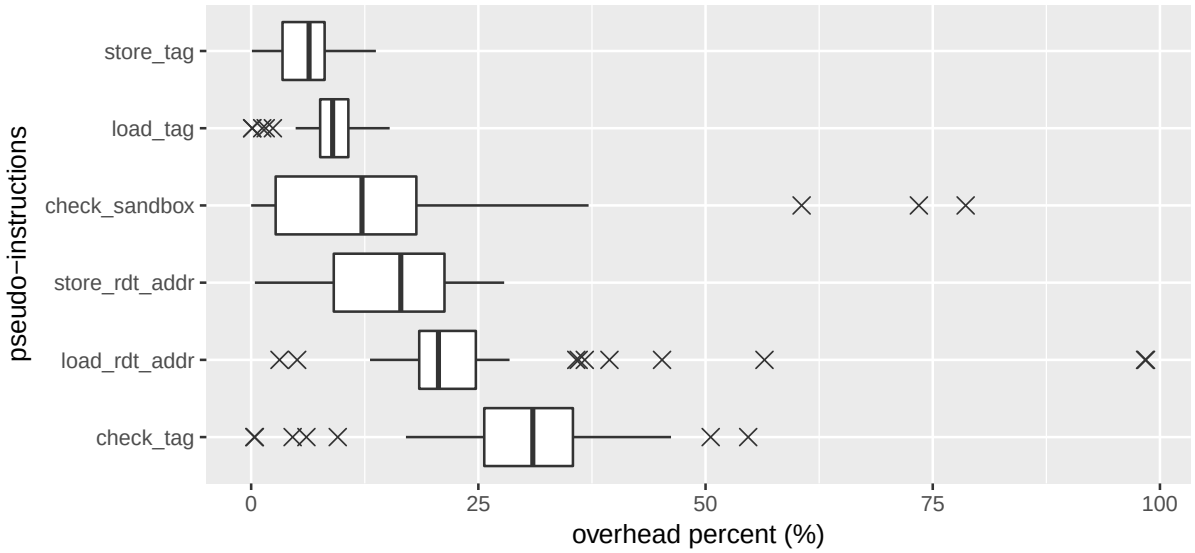


Figure 3.12: Decomposition of the DFI overhead per pseudo-instruction. The box represents the interquartile with the bar inside as the median. Whiskers incorporate up to 1.5 times the interquartile. Extreme values (outside the whiskers) are represented as cross marks

The first point of interest is that the majority of the overhead is due to two pseudo-instructions : the **check_tag** pseudo-instruction that represents a median at 30.9% of the overhead and the **load_rdt_addr** (resp. **store_rdt_addr**) pseudo-instruction that represent a median at 20.6% (resp. 16.5%) of the overhead. When combining load and store **rdt_addr** pseudo-instructions, the median percentage reaches 39.0% of the overhead. From this, we can deduce that we should prioritize the optimization of the **check_tag** and the **rdt_addr** pseudo-instructions.

The second interest is the variation of the distribution in function of the pseudo-instruction. Indeed, we can see in Figure 3.12 that some pseudo-instruction relative overheads are fluctuating more in function of the benchmarks than others. In particular, we

see that the **load_rdt_addr**, **check_sandbox** and **check_tag** pseudo-instructions can have multiple outlier points where the benchmark spend more than 50% of its overhead in these pseudo-instructions. On the other hand, the **load_tag**, **store_tag** and **store_rdt_addr** seem more stable with no high outlier points. This provides us with another incentive to focus our work on the three pseudo-instructions: **load_rdt_addr**, **check_sandbox** and **check_tag**. They represent the majority of the overhead in general and, even when some benchmarks have peculiar behavior, at least one of these pseudo-instruction tends to concentrate the overhead.

To confirm this analysis and have a better understanding of which benchmarks generate the outlier points and why, we present in Figure 3.13 the decomposition of the overhead for each benchmark.

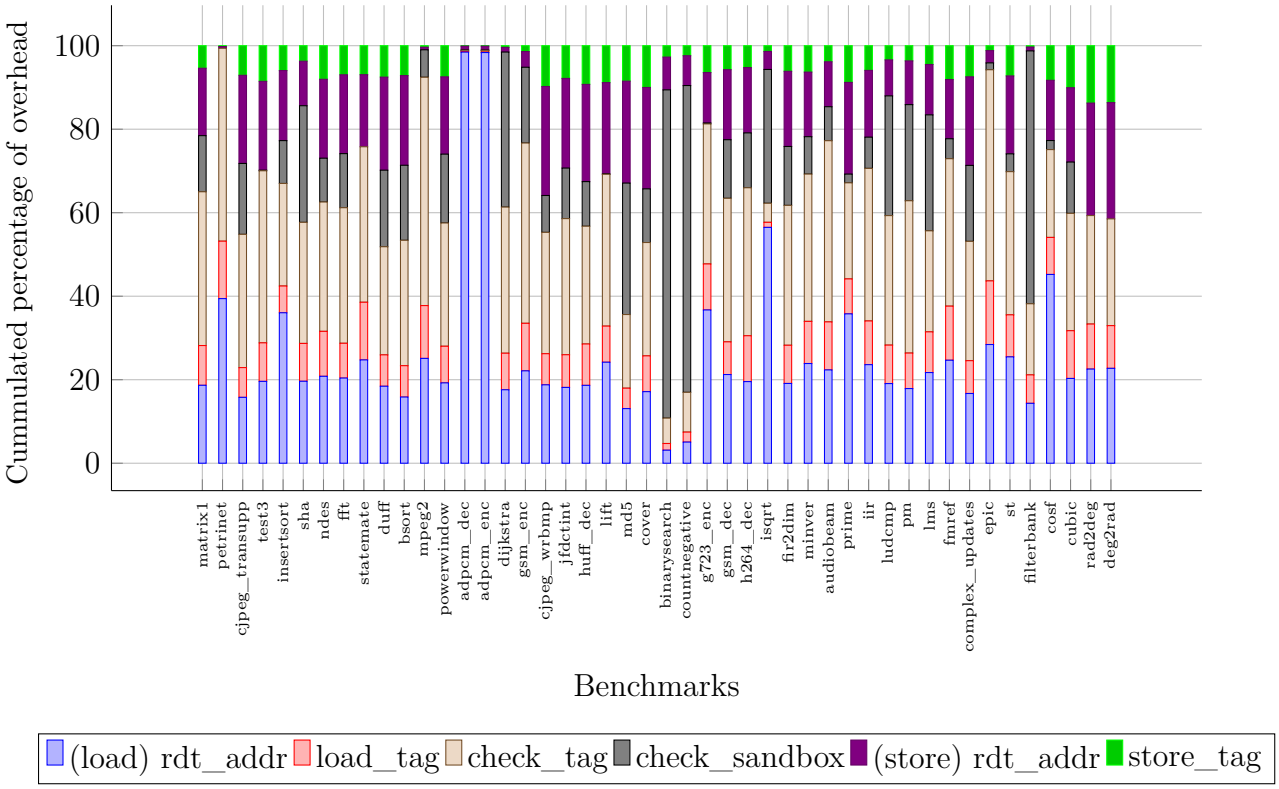


Figure 3.13: Decomposition of the DFI overhead per pseudo-instruction for each benchmark in the TACLeBench.

The most important outliers are the benchmark *adpcm_dec* and *adpcm_enc* for which the **load_rdt_addr** pseudo-instructions represents 98% of their DFI overhead. These two benchmarks performs many loads and computation with pointers and tables. However,

as in these computation the address changes between each memory instructions (typically to access different elements of the table), the redundant elimination cannot detect a constant address and thus cannot remove any redundant load instrumentation. Combined with small valid tag sets and many variables that can be placed into registers rather than in memory, thus leaving only very few stores in the program, this results in extreme examples of programs with DFI overhead dominated by the **load_rdt_addr** pseudo-instruction associated to its loads. This kind of behavior tends to appear even in other benchmarks when they manipulate tables and we present an optimization in Chapter 5 to reduce the overhead of this part of DFI.

Regarding the **check_sandbox** pseudo-instruction, the main outliers are the *binary_search*, *filterbank* and *countnegative* benchmarks. These three benchmarks spend most of their time accessing and storing values in global arrays using variables as indexes in these arrays. This prevents the static analysis from detecting that the store is safe as it would require that it detects bounds on the variables used as indexes. As the static analysis we use cannot estimate bounds on these variables, the store are pessimistically considered unsafe, which increases the number of **check_sandbox** required by the programs. Tackling this issue is more complicated as it requires to modify the optimization that detect safe store so that it can estimate bounds on variables and use these information to improve its result. This can quickly increase the complexity and time required by the analysis. Without going to such length, we provide a small optimization in Chapter 5 as a side product of our main optimization.

The two upper outliers for the **check_tag** pseudo-instruction are *epic* and *mpeg2*, two benchmarks that contain the largest valid tag set (up to 12 tags), thus explaining the predominance of this pseudo-instruction. Any optimization that manages to reduce the number of checks per load would immediately be beneficial, not only for these two benchmarks but for most benchmarks in general as this pseudo-instruction is one of the main source of the DFI overhead. We study one such optimization in Chapter 4.

All the lower outliers for all the pseudo-instructions can be explained as benchmarks that are upper outliers for another pseudo-instruction.

3.4 Conclusion

In this chapter, we presented with more details the DFI and its current optimizations in Section 3.1. We then presented our own implementation in Section 3.2. Finally, we

proceeded to an analysis of the time overhead of our implementation. We analyzed its overall overhead on the WCET and used a measurement-based method to distinguish the overhead of each pseudo-instruction composing the DFI.

To conclude, the analysis of the DFI overhead shows that the two main sources of overhead are two pseudo-instructions: **check_tag** and **rdt_addr**. The pseudo-instruction **check_sandbox** can also lead to a large overhead in some benchmarks but this is less prevalent in general. Furthermore, optimizing this pseudo-instruction could make the optimization intractable for large programs as eliminating this pseudo-instruction requires to ensure that the protected store cannot corrupt the RDT, which would require to compute the possible range of target addresses for the stores. This leads us to search optimizations for the two pseudo-instructions: **check_tag** and **rdt_addr**. In Chapter 4, we focus our attention on finding better tag representations to reduce the overhead of the **check_tag** pseudo-instruction in the context of real-time systems. In Chapter 5, we present an optimization that reduces redundancies of **rdt_addr** for loads and stores in the same basic-block that target addresses in the same structure or array.

Chapter 4

Reducing load check costs with RT-DFI

Chapter overview

This chapter tackles the overhead of DFI on real-time systems. To reduce this overhead, we present an optimization of the **check_tag** part of DFI. We developed a tool, called RT-DFI, that perform this optimization and targets specifically the WCET. This tool uses the results of the WCET analysis to construct the optimization of the **check_tag** part such that it impacts the WCET. We show with our experiments that RT-DFI can reduce the overhead of DFI by a mean 7.6% on the WCET compared to the WCET obtained in Chapter 3.

We saw in Chapter 3 that one of the main source of overhead of the DFI is the part that checks the tag associated to the loaded data. To reduce its impact, we propose a new optimization that targets the **check_tag** pseudo-instruction. This optimization uses data collected on the WCEP to specifically target loads on this path, with the goal of reducing the WCET. To do so, this optimization has two parts. First, it finds integers to represent the tags that identify each store instruction in order to reduce the number of intervals checked along the WCEP. This part can be seen as a variation of the **tag check optimization** presented in Section 3.1.3, specialized for a specific path. The other part of the optimization searches for a better scheduling of the intervals to check at each load instruction. The goal is to place the intervals most likely to contain the tag first such as the WCET estimation can deduce that there is no need to check the tag against all the intervals.

Our contributions in this chapter are the following:

- We designed an optimization focused on improving the **check_tag** part of DFI. This optimization uses data collected on the WCEP to construct an optimization problem, later solved with an **Integer Linear Programming** (ILP) solver.

- We present a tool, called RT-DFI, that compiles the program, uses an external WCET estimator (in our case *AiT*) to retrieve the data on the WCEP, performs the optimization and can perform multiple passes of this optimization to handle changes in the WCEP due to previous passes of the optimization.
- We tested RT-DFI on the TacleBench benchmark suite and showed that it reduces the overhead of DFI on the WCET by a mean 7.6% compared to the WCET obtained in Chapter 3.

In this chapter, we first present our optimization and how we use WCEP information in Section 4.1. We then formally define the construction of the ILP problem that our optimization uses in Section 4.2. Finally, we test our optimization in Section 4.3.

The work presented in this chapter has been published in the conference paper: *RT-DFI: Optimizing Data-Flow Integrity for Real-Time Systems*, in ECRTS 2022 - 34th Euromicro Conference on Real-Time Systems.

4.1 Optimizing the DFI impact on WCET

4.1.1 Using WCEP information to optimize tag checks

The tag check overhead can be decomposed into two factors. The first factor is the number of checks required to cover the whole valid tag set of the load. Since the tags are checked using intervals, we can improve this factor by modifying the representation of the tags to reduce the number of intervals that cover the valid tag set. The second factor is the order in which we check these intervals. The tag check verifies if the tag belongs to each interval covering the valid tag set, and jumps to the rest of the code as soon as it finds an interval containing the tag. Thus, we can reduce the tag check overhead by first checking the most frequent intervals. In the rest of this chapter, we use the term **interval order** to describe the order of the checks against the intervals.

RT-DFI uses the context data (number of executions in the WCEP and the possible tag in memory obtained with the value analysis) to optimize these two factors. To improve the WCET, we perform the optimization specifically on the WCEP. Thus, we only focus on contexts present in the WCEP. This approach also decreases the number of loads/contexts that we consider, reducing the complexity of our optimization problem.

In the rest of this chapter, we assume that the WCET estimator performs a value analysis and that we have access to its results and more particularly, we can retrieve

the possible tags stores in memory according to this analysis. The WCET estimator uses this value analysis to improve the IPET, avoiding paths in each context where the value analysis detects that these paths are infeasible. For example, the value analysis may establish that a condition is always satisfied in some context C_0. In this case, the IPET only considers the path when the condition holds for the context C_0, even if the alternative path is more costly for the IPET in general. The same principle applies to the tag checks, which are series of conditions. Thus, although all the tag checks are present in the program code, the value analysis may refine the IPET in some contexts. If all the possible tags (according to the value analysis) are verified before the last check, the IPET will not even consider the remaining checks as they belong to an infeasible path, thus improving the precision of the WCET. Even when the value analysis cannot restrict the possible tag values for a given load, it still considers that the possible values are restricted by the valid tag set of the load since it only reasons on the legitimate executions of the program.

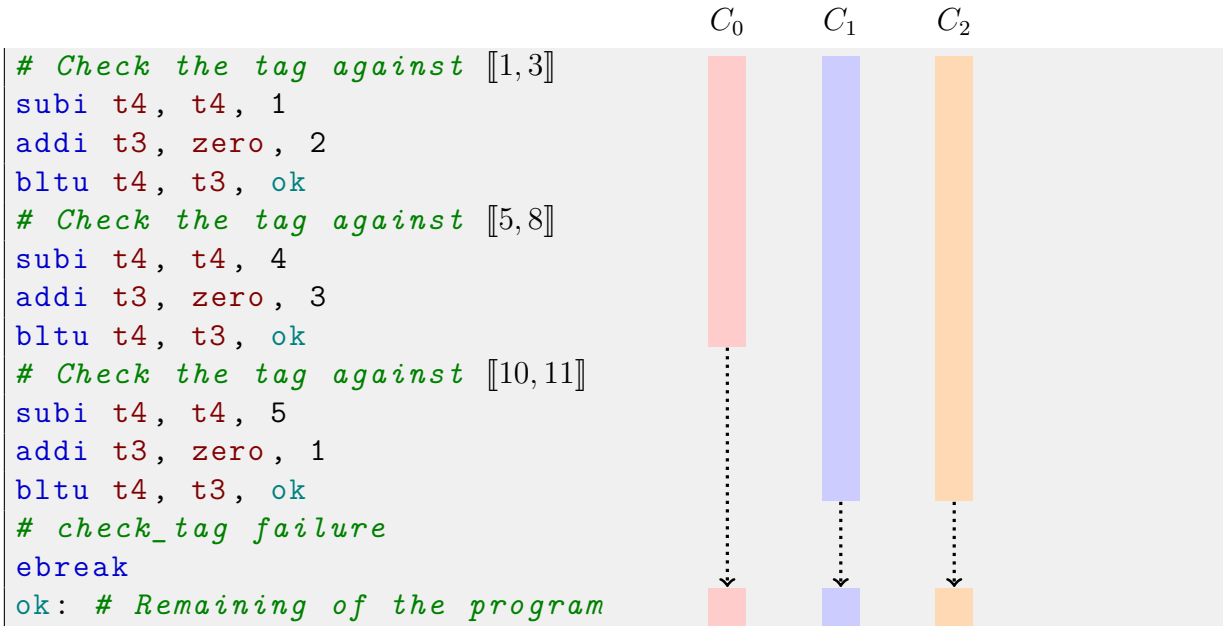


Figure 4.1: Example of check_sandbox (t4, {1,2,3,5,6,7,8,10,11}) and the resulting path analysis in three contexts: C_0 (tag $=$ 7), C_1 (tag $\in \{1, 10\}$) and C_2 (tag $\in \{1, 2, 3, 5, 6, 7, 8, 10, 11\}$)

For example, we present in Figure 4.1 a tag check with three intervals, $[\![1, 3]\!]$, $[\![5, 8]\!]$, and $[\![10, 11]\!]$. We also present the paths considered by the IPET for three contexts C_0, C_1, and C_2 without optimization. The paths considered by the IPET are represented with

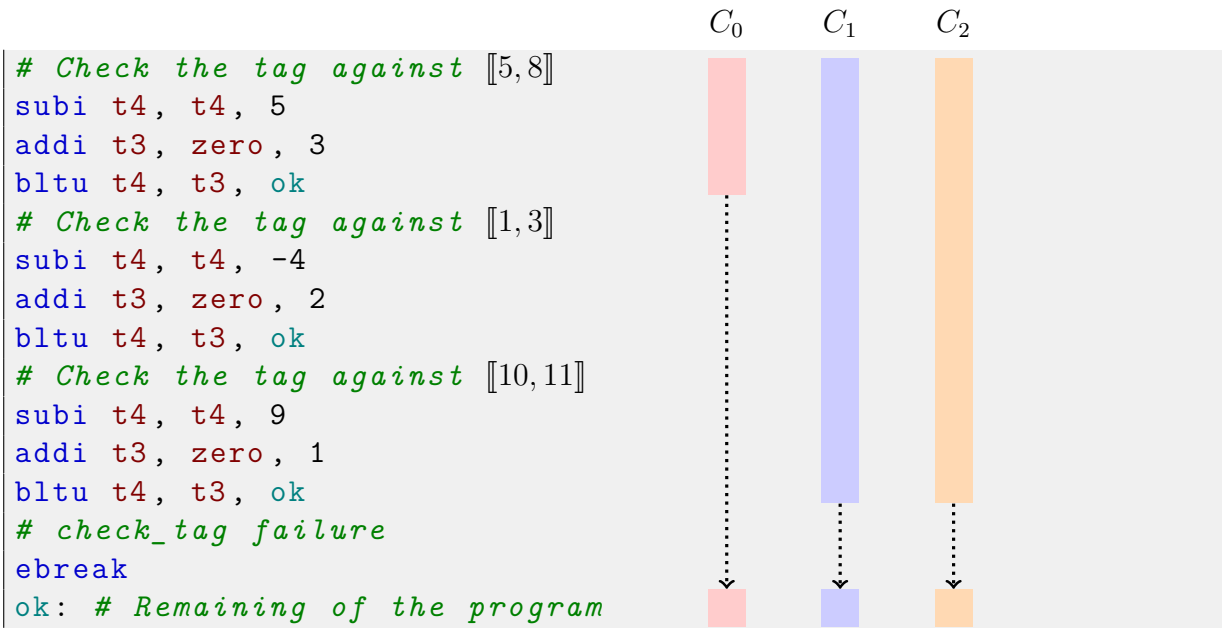


Figure 4.2: Optimized for context C_0

colored bar on the side of the code. We suppose the value analysis infers that the loaded tag is equal to 7 in context C_0, is either 1 or 10 in context C_1, and can take any value of the valid tag set in context C_2, based on the conditions in the path considered in the different contexts. In the context C_0, the considered paths follow the first two checks (against $[\![1, 3]\!]$ and $[\![5, 8]\!]$) and then jump directly to the rest of the program as the loaded tag is equal to 7 in this context. Thus, the cost of this snippet of code in the IPET, for the context C_0, is only the cost of the first two checks and not the third.

We want to show how we can use the information from the value analysis to improve DFI overhead. In Figure 4.2, we show how to optimize the cost for context C_0 by placing the check against $[\![5, 8]\!]$ as the first check. This leads the IPET to only consider the cost of this first check in context C_0, thus reducing the cost. However, as multiple contexts may exit on the same snippet of code, it is not always possible to optimize all the contexts together. For example, optimizing C_1 would require to place the checks against $[\![1, 3]\!]$ and $[\![10, 11]\!]$ as the two first intervals, which is in contradiction with the optimization for context C_0. Furthermore, as context C_2 has no specific restrictions on the loaded tag (except it belongs to the valid tag set), switching the place of tag checks would not affect the cost of such context.

Another possible optimization for context C_1 and C_2 is to modify the tag represent-

ations. If the new representations of 1 and 10 belong to the same interval, we can place this new interval as the first one to check, effectively optimizing context C_1. Furthermore, the only way to optimize C_2 is to reduce the number of intervals that cover the valid tag set by modifying the tag representations.

This example shows how different optimizations are possible in function of the information provided by the value analysis. We can either change the interval orders, modify the tag representations, or both. Modifying the interval orders only has a local impact on the WCEP. On the other hand, modifying the tag representations has consequences not only on the load we are focusing on, but also on every other load that has any of these tags in its valid tag set. Thus, knowing if a tag representation modification is interesting is more complex than for an interval order modification. Furthermore, multiple contexts for the same load can have conflicting optimizations. To deal with this problem, we formulate it using an ILP that provides a good solution along the whole WCEP. In particular, this ILP takes into account that modifying the tag representations to optimize the contexts of one load may result in increasing the number of checks of another load.

Depending on the result of the value analysis, we can extract three kind of possible contexts that have different optimization strategies:

1. **Known tag context.** Context like C_0 where the loaded tag is known. In this case, the best optimization is to place the interval containing the known loaded tag as soon as possible such as the cost of checking against the other intervals is not considered by the IPET.

2. **Partially known tag context.** Context like C_1 where the set of possible loaded tags is a strict subset of the valid tag set of the load. In this case, the IPET jumps to the rest of the code once every tag in the set of possible loaded tags as been checked. Thus, to optimize such context, we need to place first all the intervals that contains a tag in the set of possible loaded tags. This can be combined with a modification of the tag representations to check the set of possible loaded tags with fewer intervals.

3. **Unknown tag context.** Context like C_2 where the set of possible loaded tags is equal to the valid tag set of the load. In this case, the only optimization strategy is to reduce the number of checks by changing the tag representations.

These different kinds of context represent how much information we have on which checks are considered or not in the IPET. The goal is then to use the context data to

improve the tag representation and the interval orders on the WCEP. To do so, we use an ILP model that searches for a minimal number of tests along the WCEP. In contrast with the greedy algorithm of the first **tag check optimization** described in Section 3.1.3 of Chapter 3 (based on [CCH06]), an ILP can provide better solutions based on context information (see Section 4.3). In particular, while the greedy algorithm only performs local improvement load per load, an ILP can find optimizations that have a global impact on multiple loads. Furthermore, the current ILP solvers are very efficient and propose to set a timeout to stop the solving after a given time, which can be used to obtain a good solution while having a bound on the time consumed by the solving algorithm.

The optimization flow, presented in Figure 4.3, is the following: the program is compiled the first time with DFI and all the optimizations present in the original paper by Castro et al. [CCH06]. We then use the WCET analysis to construct an ILP that we optimize to find a better solution on the WCEP. We modify the program with this new solution. We then repeat the optimization process by creating a new ILP based on the new program executable, while still maintaining the same level of optimization on the previous paths. This allows RT-DFI to converge as the WCET reduces or stays the same. The process stops when the WCET does not improve anymore or after a given number of iterations.

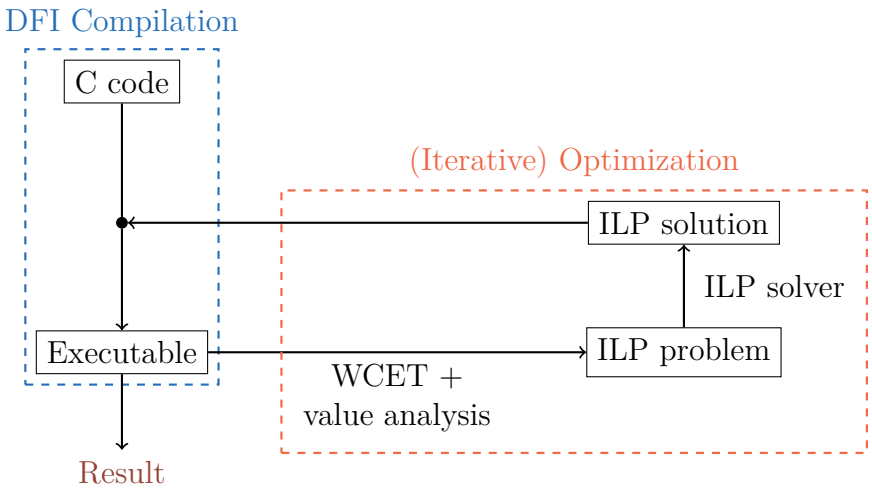


Figure 4.3: High-level workflow of the RT-DFI optimization

4.1.2 Principle of the ILP

Our optimization uses an ILP solver to minimize the number of checks on the current WCEP. Using the ILP, we search an optimized tag representation for the current WCEP combined with, for each load in the WCEP, an optimized interval order. The goal is to reduce the overhead of the **check_tag** pseudo-instruction on the current WCEP. To do so, we use the result of the value analysis for each load in the WCEP. In particular, for each load, we segregate the contexts into known/partially known/unknown tag contexts, and we retrieve the **weight** of each context in the WCET, i.e., the number of times each context is executed in the WCEP. We also assume that the cost of a check is the same on the WCET, whether it checks against an interval or a single tag. This allows us to model single tags as singleton intervals in our ILP. This assumption holds in our implementation (see Section 3.1.3), where we generate code that checks intervals as fast as a single tag using only one branch instruction (with a method explained in [CCH06]).
We construct the ILP with three steps:

1. **Tag representation.** This step associate each tag in the valid tag set of a load in the WCEP with an ILP variable which value is the tag representation of the tag. It also adds constraints to the ILP problem such that two tags do not share a same representation.

2. **Interval orders.** This step generates ILP variables and constraints to represent, for each load in the WCEP, the interval order of each tag contained in its *valid tag set*. In particular, constraints are added such that two tags belonging to the same interval have the same order value and two tags belonging to two different intervals have different order values.

3. **Objective function.** This step constructs the objective function the ILP solver will minimize. The objective function is a weighted sum of the order variables. For every context of a given load, the number of checks is the maximum of the order variables of the possible tags of the context (i.e., how much interval must be checked before we can skip the rest) multiplied by the **weight** of the context.

For example, we present in Table 4.1 four tags with their tag representations (for the entire program) and their interval orders (for a given load l). We see that for this load, tags A and B are tested first, then tag C and finally tag D. We also present 3 contexts C_0, C_1 and C_2 with their possible tags, the cost associated with each context as well as

the overall cost. We note w_0 (resp. w_1 and w_2) the **weight** of context C_0 (resp. C_1 and C_2) in the WCEP, which is the number of time this context cost is taken into account in the WCEP. The context C_0 has B as its only possible tag which is checked first for this load. Thus, the IPET only considers the cost of the first interval for this context and the cost we associate to C_0 is 1 (the interval order of B) times w_0 (the weight of C_0). For context C_1, which has more possible tags (C and D), the IPET considers the worst-case scenario where the loaded tag is D and is checked with the third interval. Thus, the cost of this context is the maximum between the interval orders of all its possible tags (i.e. $\max(2,3) = 3$) times w_1. Finally, for an unknown context such as C_2, the IPET must consider the worst-case possibility which is that the loaded tag is only checked in the last interval. Thus, it has a cost of 3 (i.e. $max(1,2,3)$) times w_2. The objective function is the sum of each context cost for each load.

Tag	Representation	Interval Order (for load l)
A	2	1
B	3	1
C	5	2
D	9	3

(a) Tag representations and interval order.

Context	Tags	Cost
C_0	B	$\max(1) \cdot w_0$
C_1	C,D	$\max(2,3) \cdot w_1$
C_2	A,B,C,D	$\max(1,2,3) \cdot w_2$
All	A,B,C,D	$\max(1) \cdot w_0 +$ $\max(2,3) \cdot w_1 +$ $\max(1,2,3) \cdot w_2$

(b) Contexts and associated costs. w_0, w_1 and w_2 represent the *weight* of the contexts (the number of time they appear in the WCEP).

Table 4.1: An example of how the cost of the contexts are computed by the ILP for an arbitrary load l, knowing a tag representation and the interval order

To deal with changes of the WCEP we add new constraints to our ILP that prevent new optimizations from destructing the previous ones, which allows RT-DFI to converge. These new constraints have the following shape: $O_{\text{previous}} \leq V_{\text{previous}}$ with O_{previous} the previous objective function and V_{previous} the previous objective function value. The O_{previous} is constructed the same way as the current objective function, but with the context of the previous optimization. The V_{previous} is just the minimal value of the objective function found by the previous optimization. These constraints force the ILP to optimize the current WCEP while maintaining the same level of optimization on the previous path. Of course, if we have multiple previous WCEPs, we can add one constraint of this shape per previous WCEP.

4.2 Formal definition of the ILP for WCET-oriented tag check optimization

In this section, we present a formal definition of the ILP we use to optimize DFI on the WCEP. We first give a notation table and describe which data is available to construct the ILP in Subsection 4.2.1. We explain in Subsection 4.2.2 how to compute the number of checks for a load when the tag representations and the order of the intervals are known. We then present in Subsection 4.2.3 the ILP that minimizes the number of checks on the WCEP by optimizing the tag representations and the order of the intervals at the first iteration of the algorithm. Finally, we explain in Subsection 4.2.4 the constraints we add to the ILP to handle potential WCEP changes.

4.2.1 Notation table and problem formal definition

Notation	Type	Signification
$[\![a, b]\!]$	Interval	**Integer** interval between a and b
l	Load	A protected load
L	Set[Load]	Set of loads in the WCEP
t	Tag	A tag
T	Set[Tag]	Set of tags checked in the WCEP
r_t	$\mathbb{N}$	The tag representation of t (fixed)
s_l	Set[Tag]	Valid tag set of load l
$I_{l,t}$	Interval	Interval specific to l containing r_t
$\phi_{l,t}$	$\mathbb{N}$	Order of $I_{l,t}$ (check order) (fixed)
C_l	Context	A context for l in the WCEP
T_{C_l}	Set[Tag]	Possible tags for the context C_l
w_{C_l}	$\mathbb{N}$	Number of occurrence of C_l in the WCEP
N_l	$\mathbb{N}$	Number of checks of l in the WCEP
Succ(t)	Tag	t' such as $r_{t'} = r_t + 1$

Table 4.2: Notation table for the mathematical terms

Two tables are used to formally describe the ILP. Table 4.2 contains the mathematical terms we use while Table 4.3 lists the ILP variables and constants used in the ILP formulation. In this second table, a few notations represent values also present in the mathematical domain. The difference between them is that r_t or $\phi_{l,t}$ are already determined (noted *fixed*) when we use them in the mathematical expressions while R_t and $\Phi_{l,t}$ are variables of the ILP that we want the solver to find (noted *free*).

Notation	Type	Signification
M	Constant	Number greater than any factor in the ILP (see big-M notation [GNS09])
start	Constant Tag	Special tag used as the start of tag representation
end	Constant Tag	Special virtual tag used as the end of tag representation
V	Set[Tag]	$T \cup \{\text{start}, \text{end}\}$
v_t	$\mathbb{N}$	Vertex representing tag t
entry_t	$\mathbb{N}$	Number of edges entering v_t
exit_t	$\mathbb{N}$	Number of edges exiting v_t
$e_{t,t'}$	$\mathbb{B}$ (boolean)	There is a directed edge from t to t' (or not)
R_t	$\mathbb{N}$	Represent r_t (free)
$\lambda^+_{l,t}$	$\mathbb{B}$	Represents if $\text{Succ}(t) \in s_l$ or not
$\Lambda^+_{l,t,t'}$	$\mathbb{B}$	$\lambda^+_{l,t}$ if $R_t < R_{t'}$ else $\lambda^+_{l,t'}$
$\Phi_{l,t}$	$\mathbb{N}$	Represent $\phi_{l,t}$ (free)
$\Phi^+_{l,t}$	$\mathbb{N}$	$\Phi_{l,t'}$ for t' such as t' = $\text{Succ}(t)$
$\Delta_{l,t,t'}$	$\mathbb{N}$	Represents $\|\Phi_{l,t} - \Phi_{l,t'}\|$
$\Delta^+_{l,t,t'}$	$\mathbb{N}$	Represents $\|\Phi^+_{l,t} - \Phi_{l,t'}\|$
$\Gamma_{l,t,t'}$	$\mathbb{N}$	$\Delta^+_{l,\alpha,\beta}$ if $\text{Succ}(\alpha) \in s_l$ else 0 with $\alpha, \beta \in \{t, t'\}$, $R_\alpha < R_\beta$

Table 4.3: Notation table for ILP variables and constants

We first recall the problem at hand, and we make explicit which data we have before the ILP. We then dive into the formal construction of the ILP in the next subsections. Our goal is to construct an ILP that can select the tag representations (globally) and interval orders (for each load on the WCEP) to minimize the number of checks on the current WCEP. To do so, we have data on all the contexts of each load in the WCEP. For each context C_l of the load l, present on the WCEP, we have the result of the value analysis (in the worst-case, the result is the valid tag set s_l of l) as well as the number of occurrences of C_l in the WCEP. When we want to iterate the optimization after a change of WCEP, we also consider that we have the same kind of data for the previous WCEPs (as we just optimized them) as well as the value of the objective function of the previous iterations. These data are used in subsection 4.2.4 to never undo previous optimizations.

4.2.2 Computing the number of checks

For this part, we consider that we know every tag representation, written r_t. We regroup the tag representations into intervals for each load l and we assign an arbitrary order to these intervals. Note that although s_l is a set of tags, we can map it to a union of intervals

t	r_t	$I_{l,t}$	$\phi_{l,t}$
A	1	$[\![1,2]\!]$	1
B	2	$[\![1,2]\!]$	1
C	4	$[\![4,4]\!]$	2

Table 4.4: Example of the tag representation, constructed intervals and intervals order

containing only valid tags. This union can be made of minimal size as follows: we consider all tag representations are single-element intervals and we merge adjacent intervals until there is no more fusion possible. We note $I_{l,t}$ the interval specific to l that contains r_t. As the intervals can be checked in an arbitrary order, we assign to each interval $I_{l,t}$ an index $\phi_{l,t}$ (starting at 1) which represents the order of the checks of the intervals (the interval with index 1 is checked first then the one with index 2, etc.). Note that for two tags t and t', if $r_t \in I_{l,t'}$ then $I_{l,t} = I_{l,t'}$ and $\phi_{l,t} = \phi_{l,t'}$. We also have $\forall l, \bigcup_{t \in s_l} r_t = \bigcup_{t \in s_l} I_{l,t}$.

We provide an example with 3 tags A, B and C in Table 4.4. As the tag representations are assigned for the entire program, the tag representations of A, B and C have no reason to be contiguous. We provide the interval of each tag and an arbitrary index for each interval.

Let C_l be a context for the load l with T_{C_l} the set of values provided by the value analysis and w_{C_l} the number of occurrences of C_l in the WCEP. We obtain the following number of checks for C_l in the WCEP:

$$\max_{t \in T_{C_l}}(\phi_{l,t}) \cdot w_{C_l} \tag{4.1}$$

This formula appears because the IPET only stops passing by checks once they have covered all the possible tags of the context, and because the context appears w_{C_l} times in the WCEP.

The number of checks for a given load is an aggregation of the number of checks for every context of this load in the WCEP (4.2). The number of checks over the whole WCEP is the sum over all the loads (4.3).

$$N_l = \sum_{C_l} (\max_{t \in T_{C_l}}(\phi_{l,t}) \cdot w_{C_l}) \tag{4.2}$$

$$\sum_{l \in L} N_l \tag{4.3}$$

4.2.3 Transformation into an ILP problem

In the previous part, we described how to compute the number of checks once the tag representations and the interval orders are known. Thus, to construct the ILP, we only need to construct these two components. As we describe the ILP, we use a few notation shortcuts, like $(x < y)$, $(x \cdot b)$ and $\max(x, y, ...)$ to represent variables with the same value as these functions. For more information on how to construct such variables, we refer to [GNS09].

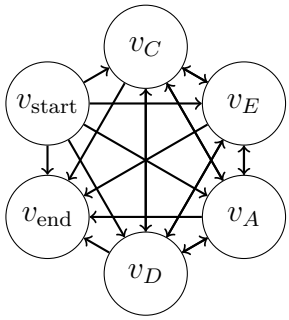
We choose the tag representations to form a contiguous interval since this mapping requires the lesser space to store the tags during the program execution, and it maintains the space overhead of DFI.

In this case, we can sort the tags by their representation. Moreover, every tag, except the first and last tags of the interval, has a successor and a predecessor. These are precisely the properties of a path in a directed graph, which is easily expressible as an ILP. Furthermore, since, in principle, every tag can have any representation, we need to start from a complete graph to allow all possible paths. Thus, we use a complete directed graph to construct the tag representations, whose vertices v_t represent the tags t.

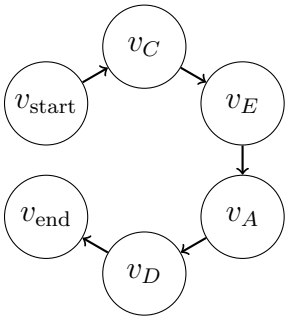
In this graph, we want to select a vertex-cover path, which provides us with the tag representations. For a given path, the edge $e_{t,t'}$ from v_t to $v_{t'}$ is present if $r_{t'} = r_t + 1$. Note that $e_{t,t}$ does not make sense and thus, we never construct it in our ILP. To ease the construction of the ILP, we introduce two virtual tags, **start** and **end**, which are the start and end of the path. Note that, as we now that **start** starts the path (resp. **end** ends the path), we do not need to add the edges (t,start) (resp. (end, t)).

In Figure 4.4a, we present an example of the graph for four tags, plus **start** and **end**. We also provide in Figure 4.4b an example of a path in this graph from v_{start} to v_{end}, and the corresponding tag representation mapping. We obtain this mapping by following the path and assigning consecutive representations to each tag.

We introduce entry_t (resp. exit_t), which counts the number of edges entering (resp. exiting) the vertex v_t as well as R_t the variable containing the tag representation of t. The



(a) All possible tag representations



$\text{start} \to 0$

$C \to 1$

$E \to 2$

$A \to 3$

$D \to 4$

$\text{end} \to 5$

(b) One tag representation and its mapping

Figure 4.4: Example of the tag representation by the ILP

constraints for the path are the following[1][2]:

$$\sum_{t,t' \in V, t \neq t'} e_{t,t'} = \text{Card}(V) - 1 \tag{4.4}$$

$$\forall t \in V, \text{entry}_t = \sum_{t' \in T \setminus \{t\}} e_{t',t} \tag{4.5}$$

$$\forall t \in V, \text{exit}_t = \sum_{t' \in T \setminus \{t\}} e_{t,t'} \tag{4.6}$$

$$\forall t \in T, \text{entry}_t = 1 \tag{4.7}$$

$$\forall t \in T, \text{exit}_t = 1 \tag{4.8}$$

$$\forall t, t' \in V, t \neq t', (R_{t'} + 1) - \text{Card}(T) \cdot (1 - e_{t',t}) \leq R_t \tag{4.9}$$

$$\forall t, t' \in V, t \neq t', R_t \leq (R_{t'} + 1) + \text{Card}(T) \cdot (1 - e_{t',t}) \tag{4.10}$$

$$\text{entry}_{\text{start}} = 0, \text{exit}_{\text{end}} = 0, R_{\text{start}} = 0, R_{\text{end}} = \text{Card}(V) - 1 \tag{4.11}$$

Constraint (4.4) forces the path to have no more edges than necessary for a path passing by each vertex only once. Constraint (4.5) (resp. (4.6)) defines entry_t (resp. exit_t). Constraint (4.7) (resp. (4.8)) forces each vertex except v_{start} (resp. v_{end}) to have only 1 entry edge (resp. 1 exit edge). Constraints (4.9) and (4.10) force $R_t = R_{t'} + 1$ if and only if v_t is the vertex next to $v_{t'}$ in the path. Finally, constraint (4.11) deals with the special cases of tags start and end. The overall design of this part of the ILP is a classic directed graph

[1] We use the classic encoding of boolean variable in ILP with $false = 0$ and $true = 1$

[2] $\text{Card}(S)$ the cardinal of S

representation [DFJ54] combined with constraints (4.9), (4.10) and (4.11).

We now explain how the ILP computes the interval orders. We create for each tag t and each load l a variable $\Phi_{l,t}$ that contains the interval order of $I_{l,t}$. In the case t does not belong to s_l, we assign an arbitrary number to $\Phi_{l,t}$ such that two different tags (both not belonging to s_l) have different indexes and such that these indexes are higher than the maximum interval index of l (i.e., greater than $\text{Card}(s_l)$). As the ILP computes the tag representations R_t, it must also compute the intervals and their orders, as the number of intervals and the interval themselves depends on the tag representations. We implicitly define the intervals with the following lemma:

Lemma 4.2.1. $\forall t, t' \in T, I_{l,t} = I_{l,t'} \iff \phi_{l,t} = \phi_{l,t'}$

Lemma 4.2.1 expresses that two tags are in the same interval if and only if they have the same index. Thus, we can encode in the ILP that two tags t and t' are in the same interval for the load l if and only if $\Phi_{l,t} = \Phi_{l,t'}$.

Writing constraints that represent the fact that two tags t and t' are in the same interval (for a given l) is complex in the ILP, as it requires checking that every tag with a representation in between R_t and $R_{t'}$ belongs to s_l. To handle this problem, we use Lemma 4.2.2.

Lemma 4.2.2. $\forall a \leq b, \forall S \subset \mathbb{N}, [\![a, b]\!] \subset S \iff (a \in S) \wedge ([\![a+1, b]\!] \subset S)$

Rather than verifying that all the tags in between R_t and $R_{t'}$ belongs to s_l, we just verify that the $\text{Succ}(t)$ (considering that $R_t < R_{t'}$) belongs to s_l and let another part of the ILP handle the verification of a smaller interval. We can then recursively use the same Lemma 4.2.2 until we have no more tags in between R_t and $R_{t'}$ to check.

We can thus rewrite these two lemmas to obtain the following relation:

$$\forall t, t' \in s_l, t \neq t', r_t < r_{t'}, \phi_{l,t} = \phi_{l,t'} \iff \phi_{l,\text{Succ}(t)} = \phi_{l,t'} \tag{4.12}$$

with $\text{Succ}(t)$ the tag t' such that $r_{t'} = r_t + 1$. Relation 4.12 explains that we can infer that two tags belong to the same interval by knowing if the successor of one of the tags belongs to this interval, as long as a few conditions are met. As we do not know whether $R_t < R_{t'}$ before executing the ILP, we use ILP variables that represent $\|R_{l,t} - R_{l,t'}\|$[3], and

[3] $\|x\|$ being the absolute value of x

we build constraints that enforce Relation 4.12.

$$\forall t \in s_l, \Phi^+_{l,t} = \sum_{t' \in T} (e_{t,t'} \cdot \Phi_{l,t'}) \tag{4.13}$$

$$\forall t \in V, \lambda^+_{l,t} = \sum_{t' \in s_l} e_{t,t'} \tag{4.14}$$

$$\forall t, t' \in s_l, \Lambda^+_{l,t,t'} = (R_t < R_{t'}) \cdot \lambda^+_{l,t} + (R_{t'} < R_t) \cdot \lambda^+_{l,t'} \tag{4.15}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} = (\Phi_{l,t'} < \Phi_{l,t}) \cdot (\Phi_{l,t} - \Phi_{l,t'}) + (\Phi_{l,t} < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi_{l,t}) \tag{4.16}$$

$$\forall t, t' \in s_l, \Delta^+_{l,t,t'} = (\Phi_{l,t'} < \Phi^+_{l,t}) \cdot (\Phi^+_{l,t} - \Phi_{l,t'}) + (\Phi^+_{l,t} < \Phi_{l,t'}) \cdot (\Phi_{l,t'} - \Phi^+_{l,t}) \tag{4.17}$$

$$\forall t, t' \in s_l, \Gamma_{l,t,t'} = (R_t < R_{t'}) \cdot \lambda^+_{l,t} \cdot \Delta^+_{l,t,t'} + (R_{t'} < R_t) \cdot \lambda^+_{l,t'} \cdot \Delta^+_{l,t',t} \tag{4.18}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} <= \Gamma_{l,t,t'} + (1 - \Lambda^+_{l,t,t'}) \cdot M \tag{4.19}$$

$$\forall t, t' \in s_l, \Delta_{l,t,t'} >= \Gamma_{l,t,t'} + (1 - \Lambda^+_{l,t,t'}) \tag{4.20}$$

Constraint (4.13) defines $\Phi^+_{l,t}$, a variable that represents $\phi_{l,\text{Succ}(t)}$. Constraint (4.14) defines $\lambda^+_{l,t}$, a binary variable equal to 1 if and only if $\text{Succ}(t) \in s_l$. Constraint (4.15) defines $\Lambda^+_{l,t,t'}$, a binary variable equals to $\lambda^+_{l,a}$ with a the tag with the lowest tag representation between t and t'. Constraint (4.16) (resp. (4.17)) defines $\Delta_{l,t,t'}$ (resp. $\Delta^+_{l,t,t'}$) which represents $\|\phi_{l,t} - \phi_{l,t'}\|$ (resp. $\|\phi_{l,\text{Succ}(t)} - \phi_{l,t'}\|$). Constraint (4.18) defines $\Gamma_{l,t,t'}$, which represents $\Delta^+_{l,\alpha,\beta}$ if $\text{Succ}(\alpha) \in s_l$ else 0, with $\alpha \neq \beta \in \{t, t'\}$ s.a. $R_\alpha < R_\beta$. Finally, constraint (4.19) (resp. (4.20)) provide an upper bound (resp. lower bound) on $\Delta_{l,t,t'}$ that enforces the relation expressed in (4.12) and handles the case where $\text{Succ}(t)$ and/or $\text{Succ}(t')$ are not in s_l.

All these constraints organize the $\Phi_{l,t}$ variables such that we obtain the intervals and their orders. We can then use the $\Phi_{l,t}$ variables to build the objective function (4.3) (seen in Subsection 4.2.2) that we want to minimize.

4.2.4 Handling WCEP changes

A common issue when optimizing for the WCET is that, at each optimization, we must focus on the path with the highest estimated execution time if we want to decrease the WCET. As each optimization may change which path has the highest estimated execution time (i.e. is the WCEP) we must be careful to always apply the optimizations on this path. However, we must also ensure that we do not worsen the estimated execution time of the other paths, or one might become the new WCEP and ruin the optimization effort.

Our optimization has the same issue. Optimizing our program WCET may change the

WCEP. To pursue the optimization, we would then want to optimize the new WCEP, such that we continue to improve the WCET. However, doing so may interfere with the optimization of the previous WCEP. To ensure that any optimization may only improve or stabilize the WCET, we add new constraints to the ILP. For each previous WCEP, we prevent the ILP from increasing the number of checks considered by the WCET analysis on this path when optimizing a new WCEP. To do so, we add constraints of the form $\text{objective}_i <= \text{result}_i$ with objective_i being the objective function of the previous i^{th} iteration and result_i the minimal value of this objective function, found by the previous iteration of the ILP solver.

As we use the same kind of variables as the objective function of the current WCEP, all the constructions explained previously remain the same. This approach also allows us to reduce the complexity of the ILP when the same tags/loads appear in two distinct WCEP (be it the previous or current one) as we can use the same variables and constraints to represent both.

4.3 Experimental results

To validate our optimization, we implemented it in a tool called RT-DFI and applied it to the TACLeBench benchmark suite. In Section 4.3.1, we present how we implemented RT-DFI[4] and the methodology of our experiments. In Section 4.3.2, we provide and analyze the results of these experiments.

4.3.1 Experimental setup

We modified the workflow presented in Chapter 3 to add our optimization. Figure 4.5 presents these modifications as the dashed box titled **(Iterative) Optimization**. We recall from Chapter 3 that we use a modified version of *Clang* to instrument the program with DFI and *PhASAR* to obtain the DFG required by DFI. After emitting the executable, we use *AiT* [FH04] to compute the WCET, obtain the WCEP and perform the value analysis. RT-DFI extracts these data and generates the ILP problems corresponding to our optimization described in section 4.2. This ILP problem is then fed to *CPLEX* [BBL14] (we used the version 20.1) that solves it and finally, RT-DFI uses the solution to optimize the LLVM IR. The whole orchestration of the programs (*AiT*, *CPLEX*, *Clang*, *PhASAR*),

[4] https://gitlab.inria.fr/nbellec1/rt-dfi

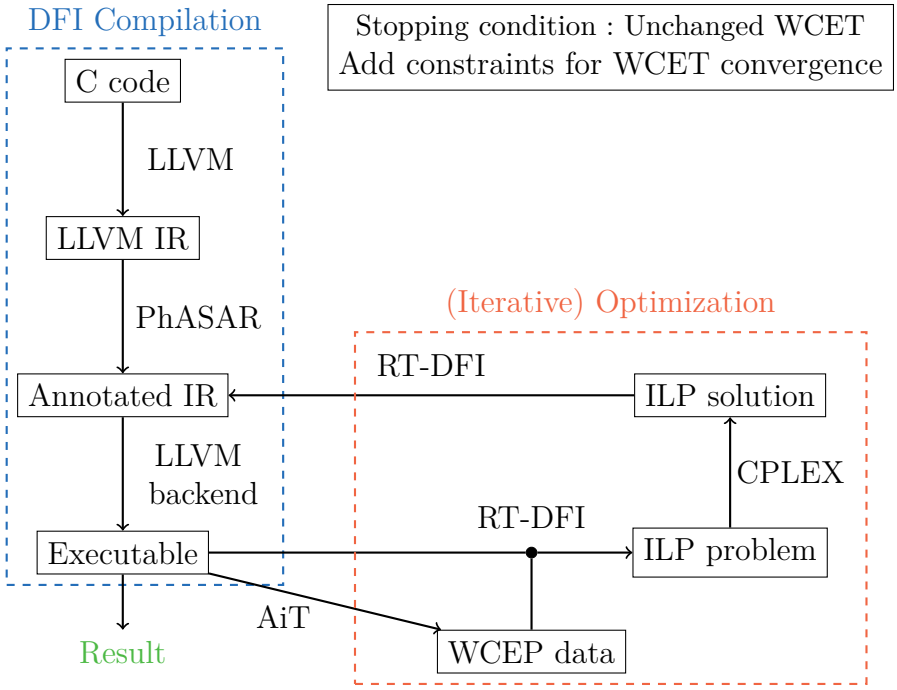


Figure 4.5: RT-DFI instrumentation workflow

the aggregation of data retrieved from *AiT* and the compilation chain (in particular the **valid tag sets**), the generation of the ILP and the optimization of the *LLVM IR* is written in Python 3.8 and represents about 8,000 lines of code. All the configuration for each software is recapped in Appendix A.

We focus our experiment on the **AiT estimated WCET** after executing RT-DFI compared to our initial implementation of DFI presented in Chapter 3, including the optimizations. We perform two optimizations based on the data provided by *AiT*.

RT-DFI. The first optimization is RT-DFI, presented in Section 4.2. For this optimization, we executed four iterations for each benchmark, to address potential WCEP changes. However, we did not see improvement past the first iteration, so we just provided the improvement after the first iteration in our analysis of the results. We did not bound the ILP runtime, as all ILP were solved in less than 40 seconds.

Value analysis improvement. The second optimization uses the value analysis of *AiT* to improve the valid tag set of our protection. In some cases, we can further reduce the valid tag sets provided by *PhASAR* by using the value analysis of *AiT*. In particular,

the data-flow analysis used with *PhASAR* is field-insensitive (as [CCH06]), and thus fails to distinguish between the cells of an array, which forces the analysis to keep some tags that could be removed. On the other hand, the value analysis of AiT is performed at the memory level and can help to refine the valid tag set of *PhASAR*. However, we can not use only AiT to obtain the valid tag sets, as their construction use information only available in the LLVM IR. This improvement of the valid tag sets is a byproduct of using the value analysis of a WCET solver to optimize the WCEP. We only present it in this section as the same result could be obtained by improving our data-flow analysis. Since it provides interesting results, we ought to present it.

4.3.2 Results

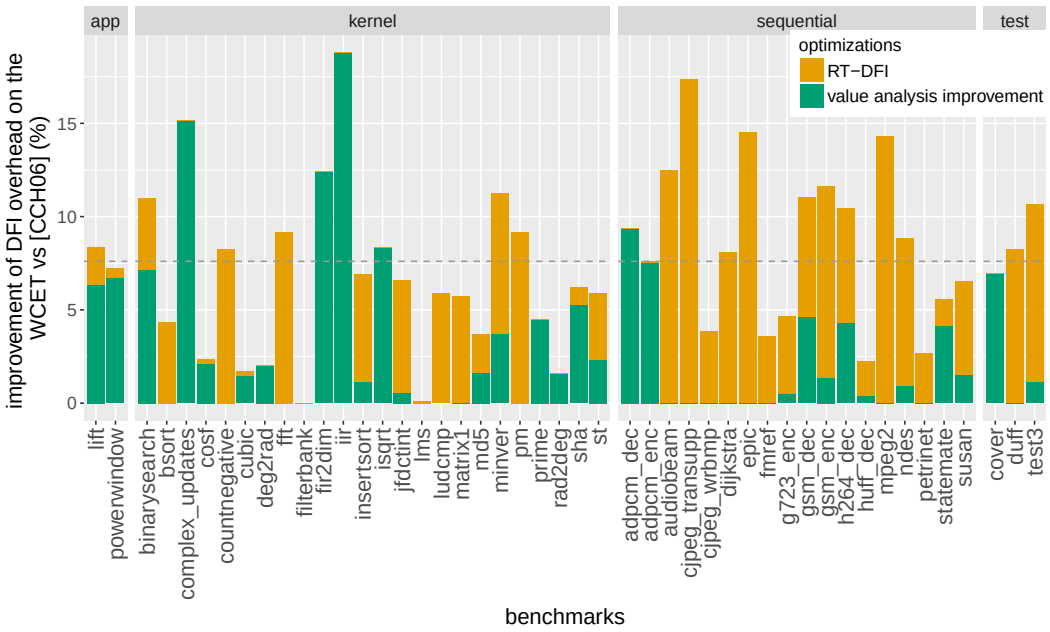


Figure 4.6: Improvement on the overhead of DFI on the WCET with value-analysis optimization and RT-DFI compared to our implementation of Castro et al. [CCH06] DFI presented in Chapter 3 with its optimizations. The cumulated mean is represented as a gray dashed line.

Figure 4.6 shows for each benchmark the improvement of the WCET in percentage compared to our initial DFI implementation when improving valid tag sets with the value analysis (*value analysis improvement* in the figure) and then with RT-DFI (*RT-DFI* in the figure). The mean improvement with both optimizations is 7.6% with a standard deviation

of 4.4 percentage points. We note that for 29 benchmarks out of the 47, RT-DFI is the most impactful, while 18 of the benchmarks are more impacted by the value analysis improvements. However, among these 18 benchmarks, 10 have no improvement by RT-DFI because every load on the WCEP has a valid tag set with a single tag, thus preventing any further optimization. These 10 benchmarks are *adpcm_dec*, *complex_updates*, *cover*, *deg2rad*, *filterbank*, *fir2dim*, *iir*, *isqrt*, *prime* and *rad2deg*. All these benchmarks have either a very small WCET without protection or a small DFI overhead (less than $\times 1.24$ if the WCET without protection is higher than $10,000$ cycles). This tends to prove that having very small valid tag sets does reduce the overhead to an acceptable level. The overhead of the smallest benchmark skyrockets as they are mostly composed of load/store instructions.

Our optimization process spent most of its runtime executing the WCET analysis ($\sim$ 66% on average) or compiling the new executable ($\sim$ 29% on average). For all the benchmarks, the ILP solver part of RT-DFI took less than 40 seconds per problem to solve. Thus, the main runtime cost of RT-DFI is due to our iterative process that requires to re-launch a WCET analysis and re-build an executable at each step. As the number of steps remains low, we do not think this is an issue for the application of RT-DFI. Furthermore, as iterative optimization does not seem efficient, we only need two WCET analysis to perform RT-DFI.

4.3.3 Notes on iterative optimization

As explained before, we did not include the results for more than one iteration of RT-DFI, as more iterations do not further reduce the WCET. We have two hypotheses as to why iterations do not provide more improvement:

1. The constraints of the previous WCET prevent improvement on the new WCEP.
2. The WCEP has only small changes that have almost no impact on the final WCET[5].

To test hypothesis 1, we relaxed the constraints of the previous WCEPs ($\text{objective}_i <= 1.05 \cdot \text{result}_i$) to give more freedom to the solver. There was no overhead reduction past the first iteration of RT-DFI. While we can argue if the relaxation is enough and that it should depend on the WCET improvement, this tends to reject hypothesis 1. However, we must remain prudent and more experiments are required to fully reject this hypothesis.

[5]in particular, due to the nature of the benchmarks that contains few paths close to the WCEP

Hypothesis 2 is hard to prove or reject because it is hard to compute path differences between two binaries, where instruction addresses and control-flow graphs may change due to the optimization. We manually examined and saw this problem arise for two benchmarks, namely *lift* and *powerwindow*, by visually examining the control-flow graphs and WCEPs in *AiT*. However, this method is time-consuming and error-prone. The automatization requires handling changes in the tag representations/interval orders that are equivalent in terms of ILP and/or WCET but may change many parts of the program. As this is a complex issue, we do not have enough data to judge this hypothesis at the moment.

4.4 Conclusion

In this chapter, we presented an optimization that targets the **check_tag** part of DFI, specifically for real-time systems. This optimization uses data retrieved when estimating the WCET to reduce the impact of checking the loaded tag on the WCET. This is a first step to break down the overhead of DFI for real-time systems. However, as we have seen in Chapter 3, other parts of DFI have an important weight in the DFI overhead. In the next chapter, we present a new optimization that targets the **rdt_addr** part of DFI and can even reduce the overhead of the **check_sandbox** part in some specific cases.

Chapter 5

Reducing address computation redundancies

Chapter overview

In this chapter, we tackle the part of the DFI overhead that comes from computing the addresses in the RDT table (the **rdt_addr** part of DFI in Chapter 3). The RDT table stores the tags used by DFI and is thus accessed before each memory instruction. We first present how repeated access to arrays or structure can generate redundancies in the RDT address computation. To detect and optimize these redundant computations, we present a structure called **Load Store Chain** (LSChain). We show how to construct and use these LSChains at compile time to perform an intra-basic-block optimization of the DFI. As a byproduct, we can also use these constructions to reduce the cost of the **check_sandbox** part of the DFI that ensures that stores can not corrupt the RDT. We experimentally show that this optimization effectively reduces the overhead of DFI on the WCET.

We saw in Chapter 3 that about 39% of the overhead induced by DFI comes from computing the RDT address at each memory instruction. In the presence of an array (or a structure), this cost can be exacerbated by repeated accesses to the array. We recall that every memory instruction in RISCV is implemented using a base register and an offset. When encountering an array, the compiler often stores the address of the array in a register and executes several memory instructions with that register as base register, using different offsets to access different elements of the array. This optimizes the program by updating already computed address with just a fixed offset. On its end, the DFI instrumentation computes every address in the RDT from scratch, loading the RDT base address and recomputing the offset in the RDT based on the target address at each memory instruction. In the case of successive accesses with a known offset, we could use the same strategy as with arrays for addresses in the RDT, computing a first address in

the RDT and then updating it with a fixed offset for each successive access.

In this chapter, we study a method to reduce the DFI overhead by removing redundant computations of the RDT address. Instead of recomputing the RDT address, we perform an intra-basic-block optimization of the DFI instrumentation by using available registers to retain and reuse the previously computed RDT address. As this optimization optimizes all the basic-blocks, it optimizes the average case execution time as well as the WCET. We first present a motivating example in Section 5.1. We then explain how to detect these redundancies using a structure we call Load Store Chain and how to reduce DFI overhead using these redundancies in Section 5.2. Finally, we present an experimental evaluation of our optimizations in Section 5.3. The work presented in this chapter has not yet been published but should soon be submitted.

Note that, while we do not apply the RT-DFI optimizations in this chapter, both optimizations can be applied at the same time and should not interfere one with another as they impact two distinct part of the DFI.

5.1 Motivating example

The main idea behind the optimizations presented in this chapter is to find redundant parts of the DFI instrumentation that can be eliminated. As a motivating example, we present the C code in Figure 5.1a and its assembly language equivalent in Figure 5.1b. This C code example is a very simple and common code where we write at two locations in a given integer array named **tab**. In the assembly code, this translates into two **addi** instructions (lines 1 and 2) that define the values to be stored and two store instructions (lines 10 and 18). We assume that register **s1** contains the address of the array **tab**. These two store instructions are protected by the DFI instrumentation (**rdt_addr** and **store_tag**) as described in Chapter 3. As these two stores are based on the same register (**s1**), the value of this register does not change between these two instructions and both instructions can be optimized with the **offset collapsing** optimization (as defined in Section 3.2.2), both **rdt_addr** pseudo-instructions compute the same value (i.e. $(\text{raddr}//4) * 2$). It is therefore possible to use an available register (in this example **s3**) to store the RDT address computed at the first store instruction and reuse it for the second store instruction (instead of computing it again), only adjusting the offset. This is presented in Figure 5.1c. In this case, we can remove the entire **rdt_addr** pseudo-instruction from the second store.

```c
int tab[10];
tab[1] = 5;
tab[3] = 7;
```

(a) C code example

```
addi s0, zero, 5
addi s2, zero, 7
slri t3, s1, 2
slli t3, t3, 1
lui t4, %hi(DFI_BASE)
add t3, t3, t4
addi t4, zero, 1
sh t4, 2(t3)
store1:
sw s0, 4(s1)
slri t3, s1, 2
slli t3, t3, 1
lui t4, %hi(DFI_BASE)
add t3, t3, t4
addi t4, zero, 3
sh t4, 6(t3)
store2:
sw s2, 12(s1)
```

rdt_addr (s1+4)

store_tag(t3+2,1)

rdt_addr (s1+12)

store_tag(t3+6,3)

(b) DFI protection of two store instructions

```
addi s0, zero, 5
addi s2, zero, 7
slri t3, s1, 2
slli t3, t3, 1
lui t4, %hi(DFI_BASE)
add s3, t3, t4
addi t4, zero, 1
sh t4, 2(s3)
store1:
sw s0, 4(s1)
addi t4, zero, 3
sh t4, 6(s3)
store2:
sw s2, 12(s1)
```

(c) Optimized DFI protection of two store instructions

Figure 5.1: Motivating example

In larger code, we could find many more memory instructions (loads and stores) that use the same base register and could be optimized. However, there are a few issues that must be taken into account to extend this idea.

The first issue is the presence of other instructions between memory instructions. Indeed, we could for example have a load, some arithmetic instructions and then a store to a close address as in Figure 5.2. In this case, we would need a register not used by any other instruction between the first and the last memory instruction of a sequence to optimize. This could be difficult to find as the more instructions there are in the sequence, the more registers are likely to be used and thus not available for the optimization. Furthermore, some registers are already in use before the sequence and we can not use them either

which can further restrict our choice of registers.

```
1 lw s1, 8(s2)
2 addi s1, s1, 7
3 slli s1, s1, 2
4 addi s1, s1, 2
5 sw s1, 12(s2)
```

Figure 5.2: Example of multiple memory instructions with arithmetic instructions in between

This brings us to our other issue: we may want to hold multiple addresses at the same time as multiple optimizable sequences may interleave as in Figure 5.3. In this case, each sequence requires its own distinct register. As we may not have enough registers available for all the sequences, we may need to choose which sequence to optimize.

```
1 lw s4, 0(s1)
2 lw s3, 8(s2)
3 sw s4, 4(s1)
4 sw s3, 12(s2)
```

Figure 5.3: Example of multiple interleaved sequences of memory instructions

To solve these issues and optimize the instrumentation, we introduce the notion of a **Load Store Chain** (LSChain).

5.2 Load Store Chains construction and use

Load Store Chain

A **Load Store Chain** (LSChain) is a chain of consecutive memory instructions having the same base register and such that no intermediate instruction, except the last one in the chain, can modify this register. We also associate the base register to the LSChain.

A LSChain contains all memory instructions where redundant instrumentation could be found and removed with the optimization presented in the motivating example. A LSChain also provides a minimum lifetime for any register that could be used to optimize

it. To perform the optimization on the memory instructions in the LSChain, we must find an available register from the beginning to the end of the chain. In particular, we note that several LSChains (with a different base register) can overlap (as seen previously with Figure 5.3) and thus the optimization of one of them may prevent the optimization of another one.

We first present how we build LSChains in Subsection 5.2.1. Then, we explain how to use them to optimize the DFI instrumentation when registers are available in Subsection 5.2.2. Finally, we present an extension that allows us to improve the DFI instrumentation by using the information retrieved during the construction of the LSChains without requiring any available register in Subsection 5.2.3.

In this chapter, we study LSChains restricted to a single basic-block and that stop at function calls. We discuss possible extension in Section 5.4.

5.2.1 Constructing LSChains

As presented above, a LSChain provides a lower bound on the lifetime of the register used to optimize this LSChain. If no available register can meet this constraint, then we are not able to optimize the instrumentation for this LSChain. Thus, we have two antagonistic objectives. On one hand, the longer the chain is, the more memory instructions we can optimize. On the other hand, a longer chain means saving a register for a longer time, which increases the pressure on the registers and may preclude other chains from being optimized.

We choose to study the LSChains basic-block per basic-block (thus without considering inter basic-blocks LSChains) because many interesting LSChains can already be found at the basic-block level. Indeed, programs are often written (and compiled) in such a way that multiple accesses to an array (or a structure) are done in the same basic-block. Thus, allowing a LSChain to continue beyond the boundaries of a basic-block puts pressure on the registers of the other basic-blocks. This could typically prevent the optimization of another, more interesting, LSChain. To illustrate this problem, consider the program whose structure is shown in Figure 5.4. In this example, we note "r - n" a chain of **n** memory instructions with **r** as its base register. As such, the first basic-block (BB1) contains a LSChain with eight memory instructions based on the register **r1**. Then a conditional branch leads to two possible basic-blocks, BB2 and BB3. In basic-block BB2 (resp. BB3), we have a new LSChain containing four (resp. five) memory instructions for the register **r6** (resp. **r4**) and then a new memory instruction on the register **r1**. If

we use inter basic-block LSChains, then BB1's LSChain extends to BB2 and BB3, also propagating the register pressure when optimizing it. If there is only one register available in BB2 and/or BB3, it follows that an entire chain would not be optimized just to extend the chain starting in BB1 by a single instruction. On the other hand, if we use intra-basic-block LSChains, the three interesting LSChains (which contain more than one memory instruction) are independent and can be optimized as long as there is a single register available.

As LSChains could still be used for inter-basic-block optimizations in some cases, we present some ideas to extend our optimizations in Section 5.4.

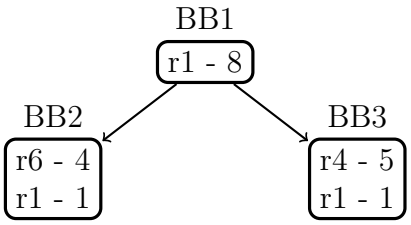


Figure 5.4: Example of problem with inter basic-block LSChain

To build the LSChains in each basic-block of a program, we perform a backward analysis on the basic-block. Note that, as we work only on basic-block, the analysis itself is simple as there is no control-flow by definition of a basic-block. This analysis is done just before instrumenting the program with DFI to avoid discrepancies between the analysis and the instrumentation due for example to changes in the register allocation or new load/store instructions appearing after the analysis. The analysis initializes an empty LSChain for each possible register in our architecture. It then iterates over the instructions of the basic-block, adding to the **R** register LSChain any memory instructions using **R** as its base register. When it encounters an instruction that modifies the register **R**, the current LSChain for **R** is saved and a new empty LSChain for **R** is created.

For example, on Figure 5.5, we present a few lines of code containing three overlapping LSChains for two registers : **sp** and **a1**. The scope of each LSChain is represented by a bar along the code, separated for each register. Each chain also has an identifier (written in the bar). The number also identifies the order in which the chains are found during the backward analysis. In this example, the backward analysis first detects the memory instruction at line 9. This instruction uses the **sp** register as its base register and thus, the instruction is added to the LSChain of the **sp** register. Since this is the first instruction

for this LSChain, this chain is given the identifier LS_1. The analysis then encounters the *sw* instruction line 8, which use the register **a1**. This instruction is added to the current LSChain for register **a1** (LS_2). Upon encountering the instruction at line 7, the analysis adds it to LS_2. Then, an instruction modifying **a1** is found (line 6), LS_2 is saved, and another empty chain is created for **a1** (LS_3). The next encountered instruction at line 5 is added to this new LSChain (LS_3), the instruction at line 4 is added to LS_1, and the instruction at line 2 is also added to LS_3. Note that the instruction at line 3 is not a memory instruction but since it only affects the register **a0**, it has no effect on LS_1 and LS_3. Finally, the last instruction (line 1) is added to LS_1 and the analysis ends.

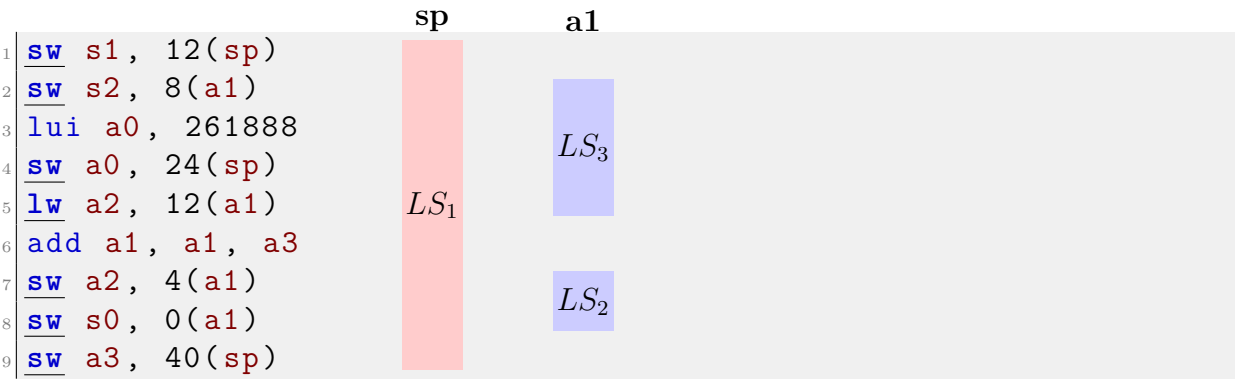


Figure 5.5: Example of LSChain construction

In the end, LS_1 contains the memory instructions lines 9, 4 and 1 with the **sp** base register. LS_2 contains the memory instructions lines 8 and 7 while LS_3 contains the memory instructions lines 5 and 2, both with base register **a1**.

5.2.2 Exploiting LSChains to remove redundant RDT address computation

We now present how we use the constructed LSChains to optimize the DFI instrumentation. We first introduce three conditions that must be satisfied by a LSChain to be able to optimize it. Furthermore, we present how we compute the available registers for a LSChain. Finally, we show how we select the LSChains for optimization when the register pressure is too high, preventing the optimization of all LSChains.

In the rest of this section, we consider that we only use available registers to optimize the LSChains. In particular, we do not consider the spilling of registers (saving and

restoring registers to memory) to optimize a LSChain. Indeed, spilling would introduce new memory instructions that would also have to be instrumented and taken into account in the computation of the LSChains. This would complicate the optimization of the LSChains, and it would be necessary to evaluate for each LSChain whether we can spill a register and whether the cost of spilling would be outweighed by the gain of the optimization. Thus, we restrict the optimization to LSChains that have a register available throughout the LSChain to avoid these problems.

The three conditions to be able to optimize a LSChain (without considering register pressure due to other LSChains) are the following:

C1 The LSChain must have two or more instructions. Otherwise, we would have no redundant instructions.

C2 We must find a register available throughout the LSChain to store the previous tag address of the base register, so that no register spill is necessary.

C3 The offset difference between any two memory instructions in the LSChain must be a multiple of four.

The third condition C3 arises because we allocate two bytes in the RDT for four bytes in the data memory. This relation implies that four aligned bytes in the main memory are all mapped to the same two bytes in the RDT. Thus, when the difference between two offsets in a LSChain is a multiple of four, we are sure that we can statically compute the address difference in the RDT because for every four bytes of difference in the main memory, we add two bytes to the RDT address, whatever the value contained in the base register (aligned or not). On the other hand, if the offset difference is not a multiple of four, the exact address difference in the RDT depends on the alignment of the initial address. For example, suppose the first address in the LSChain is 0x4 and the second 0x6. In this case, both addresses are mapped in the RDT to the address 0x2 $((\text{addr}//4) * 2)$. If the addresses were 0x6 and 0x8, the difference between the offsets would be the same (i.e. two), but they would be mapped to two different addresses in the RDT (0x2 and 0x3 respectively). If we tried to optimize these LSChains, we would have to correct the address at each memory instruction in the LSChain, which is as expensive as recomputing the address or requires more registers per chain to propagate the correction. Note that we can know when the address difference is a multiple of four as the address difference is equal to the offset difference, the base register being constant along the LSChain. Thus,

we restrict ourselves to LSChains that are optimizable without putting too much pressure on the registers, and we add this third condition.

To compute the available registers along a LSChain, we use the liveness information provided by the compiler. For each instruction in the lifetime of the LSChain (whether it is in the LSChain or not), we can ask the compiler which registers are alive[1] or dead[2]. We then combine all these sets to find the registers that are never alive along the LSChain. We also remove from this set the registers reserved for the DFI instrumentation as well as any registers that are written but never used (in this case, they are never alive, but are modified along the LSChain).

When too many LSChains are intertwined, they can put too much pressure on the registers. In this case, we have to choose which LSChains we optimize and which we do not. To make this selection, we first sort the LSChains in the decreasing gain they allow, i.e. the number of instructions they contain. Since the optimization computes the address once and then reuse it for every other instruction in the same LSChain, the length of a LSChain minus one represents the number of memory instruction we can optimize by selecting the chain. Once we have sorted the LSChains, we try to optimize them in this order. We greedily select an available register for the first LSChain and optimize the LSChain with that register. We then remove this register from the available registers of each LSChain interleaving with the one we have just optimized. If one of these LSChain has no more available register, we cannot optimize it and thus, we remove it from our set of optimizable LSChains. We apply the same process to the next optimizable LSChain in order. We stop when we have no more optimizable LSChain that have not yet been optimized.

5.2.3 Beyond RDT address computation redundancy

Our main motivation for the use of LSChains is to avoid recomputing addresses between two DFI-protected memory instructions. However, we also use LSChains to further reduce the redundancies in other parts of the DFI instrumentation. In particular, multiple **check_sandbox** pseudo-instructions can be merged into a single one.

To merge multiple **check_sandbox**, we can use the fact that, if the highest targeted address in a LSChain is safe, then all other addresses are also safe. As the RDT is set as the highest writable section of the program, we only need to execute the **check_sandbox**

[1] *alive*, which contains a value later used by the program
[2] *dead*, which value is not used later by the program, thus the register can be used to store a new value

```
1 check_sandbox(tab+4)   <-------+
2 tag_addr = rdt_addr(tab+4)     |
3 store_tag(tag_addr, 1)         |  replaces
4 store data, tab+4              |
5                                |
6 check_sandbox(tab+12)  --------+
7 tag_addr = rdt_addr(tab+12)
8 store_tag(tag_addr, 3)
9 store data, tab+12
```

Figure 5.6: Example of sandbox redundancy

with the highest offset (since the base register is constant along the LSChain) in the first occurrence of a **check_sandbox** in the LSChain and remove all other occurrences in the LSChain.

An example of this optimization is shown in Figure 5.6. In this example, which is a translation of Figure 5.1a with **check_sandbox**, we have two store instructions with the pseudo-instructions that protect them. In this case, the optimization of the **check_sandbox** replaces the first **check_sandbox** (line 1) with the second (line 6) and remove the second. As these two stores belong to the same LSChain, we are sure that the register containing **tab** is not modified between them and thus checking that $\text{tab} + 12$ is not inside the RDT ensures that $\text{tab} + 4$ is not in it either. We recall that an attacker can not jump in the middle of a basic-block as this would require to break the DFI protection beforehand.

5.3 Experimental results

In this section, we evaluate our optimizations using the same tool chain as in Chapter 3 to compile our program with DFI.

We added our redundancy analysis and optimizations to the pass that instruments the program with DFI to directly generate the optimized DFI instrumentation. We use the same benchmarks as in the previous chapters and focus on the real-time property of the programs, since we are interested in real-time programs.

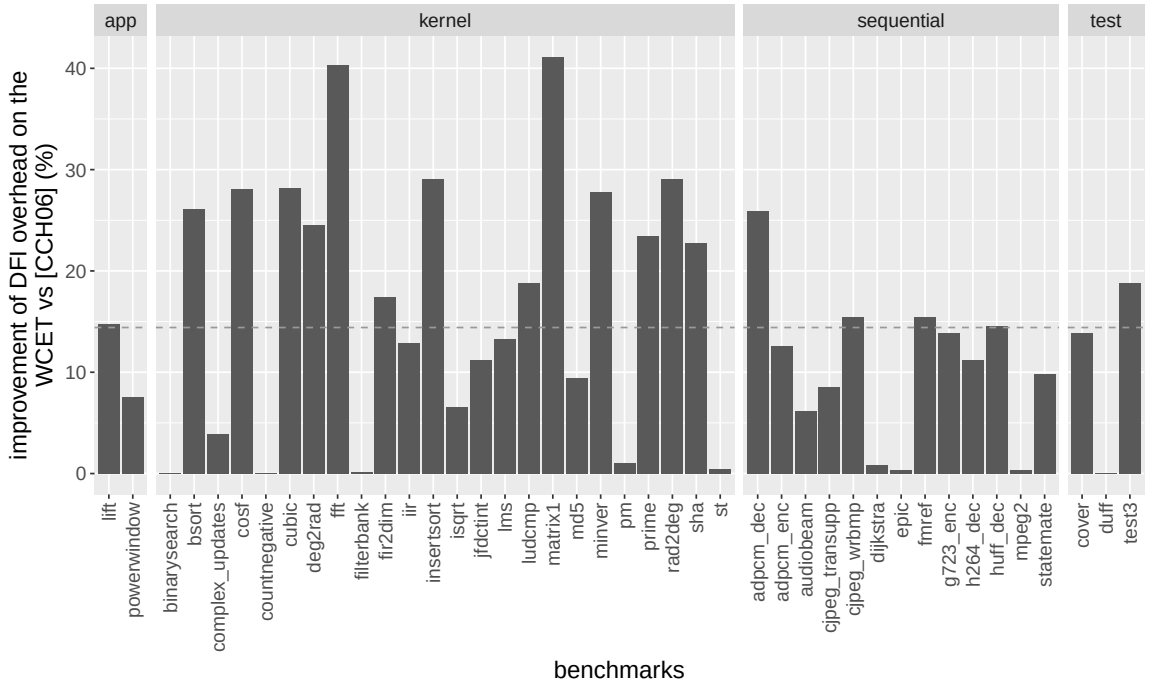


Figure 5.7: WCET overhead improvement of the redundancy optimization (**rdt_addr** + **check_sandbox**) compared to our implementation of Castro et al. [CCH06] DFI presented in Chapter 3 with its optimizations, the mean over all benchmarks is marked with a doted line.

To measure the gain of our optimizations, we first estimate the WCET (in number of cycles) of binaries without DFI protection. This measure represents our baseline for each benchmark. We consider that, for a given benchmark, any increase in the WCET past its baseline is part of the DFI overhead. We then estimate the WCET of binaries with the DFI protection including the optimizations presented in Chapter 3. From this estimate, we compute the DFI overhead without the optimizations presented in this chapter (in Section 5.2). Similarly, we estimate the WCET of the binaries with the optimizations presented in this chapter and compute the DFI overhead with these optimizations. Finally, we calculate the ratio between the two overheads to see how much the DFI overhead has been reduced by our optimizations.

We present the result in Figure 5.7. For each benchmark, we reported the improvement in DFI overhead between the binary without and the binary with our optimizations. This improvement is presented as a percentage reduction of the overhead. Across all benchmarks, our optimizations reduce the overhead by an average of 14.4% (dotted line in the figure) with some spikes at 40%.

For most of the benchmarks with less than 5% improvement, we find two reasons for the small improvement. First, in some cases, most of the WCET is spent in a single loop without the possibility of applying our optimizations (e.g., due to pointer redirection or because there is only one memory instruction in the loop). Second, most of the WCET is spent on software floating-point functions that are not instrumented by DFI as they do not dereference any pointer and thus can not corrupt the memory. This reduces the potential for improvement of our optimizations. In the second case, since most of the WCET is spent in functions that are not instrumented by DFI, the overhead of the DFI without our optimizations is already very low (less than 5%), limiting the potential of any optimization of the DFI.

5.4 Discussion

Through this chapter, we have presented a method to detect and reduce redundancies between the instrumentation of consecutive memory instructions. This method is based on the detection and exploitation of LSChains. In this section, we discuss possible extensions that we could study in future works to improve the results.

The first possible extension of our approach is related to the size of the LSChains. As presented earlier, we chose to restrict the LSChains to a single basic-block to avoid putting too much pressure on the registers. Therefore, a first extension would be to relax this constraint and allow inter-basic-block LSChains. One method to construct such LSChains at the function level might be to start with the LSChains that we know how to construct at the basic-block level. Then, we could try to merge the LSChains at the interface between two basic-blocks. To do so, we merge LSChains two-by-two with a first LSChain (that we call LS_{end}) at the end of a basic-block and the second LSChain (that we call LS_{start}) at the start of one of its successor. To merge these two LSChains, two conditions are required:. First, they must both be based on the same register. Second, the register value must not change between the end of LS_{end} and the beginning of LS_{start}. Given these two conditions, we can have a new inter-basic-block LSChain that starts at the beginning of LS_{end} and end at the end of LS_{start}.

We present an example in Figure 5.8. Figure 5.8a presents a CFG with the LSChains based on register **r** for each basic-block. Note that in this case, we do not check whether the LSChains are optimizable because two non-optimizable LSChains could become optimizable when we merge them. This is the case for all the LSChains except LS_1 in our

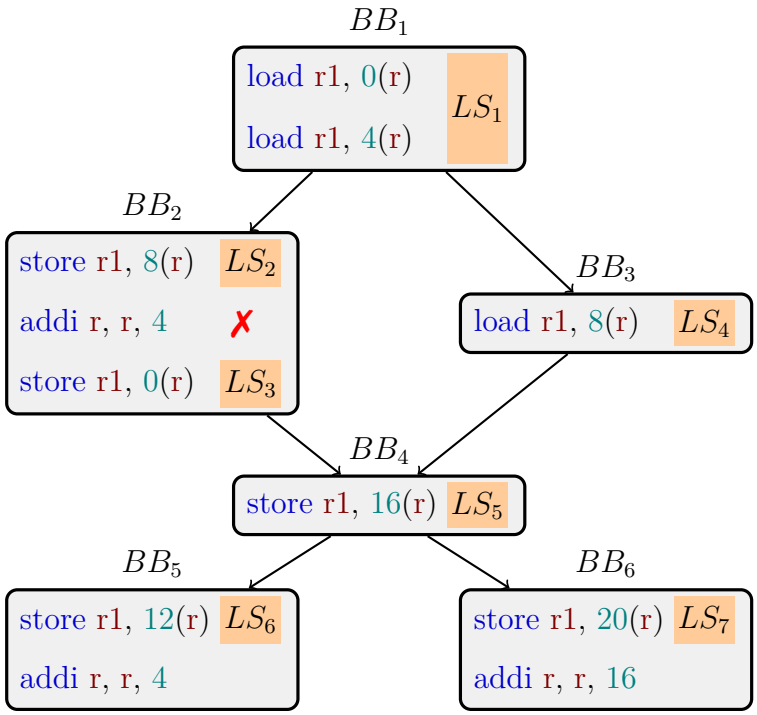
example as they all contain only one instruction. We can then merge LSChains at the basic-block interface. In this example, we can merge LS_1 with LS_2 and LS_4, LS_4 with LS_5, LS_3 with LS_5 and LS_5 with LS_6 and LS_7. In the end, we get a directed acyclic graph of memory instructions with multiple beginnings and ends. This graph is presented in Figure 5.8b for our example and has two beginnings (LS_1 and LS_3) and three endings (LS_2, LS_6 and LS_7).

As we wrote earlier, the main problem with inter basic-block LSChains is register pressure. With longer chains (potentially spanning the entire function), the register pressure may be too high to find available registers for optimization. This could ruin our optimization and degrade performance. Therefore, we also need a way to prevent this degradation. One idea would be to allow the optimization to split a long LSChain into smaller LSChains. This has two advantages. First, in the worst case, we could split the inter-basic-block LSChains between each basic-block to regain intra-basic-block LSChains and thus ensure not to degrade the WCET compared to our current optimizations. Second, we could split long LSChains that put too much pressure to obtain multiple LSChains, some of which we could optimize, potentially even across basic-block boundaries. However, using this method would require the development of a new heuristic to decide when and where to split the LSChains.

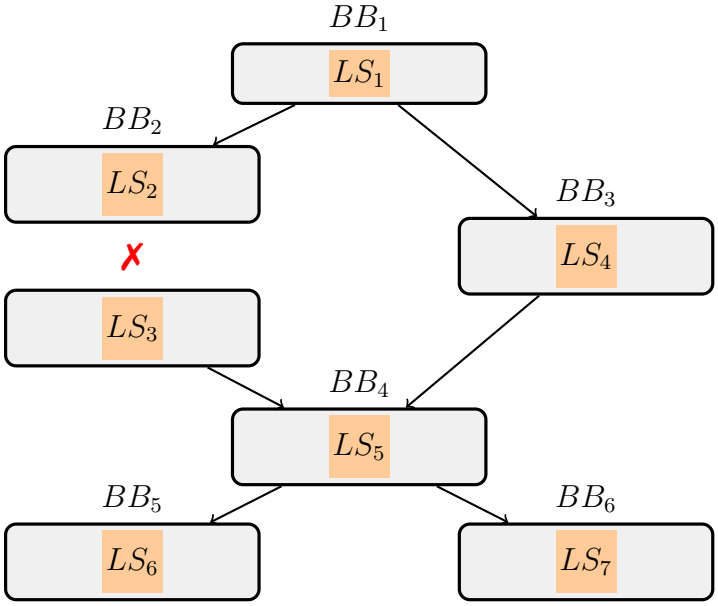
Another extension could be to allow the spilling of one or several registers to optimize a LSChain. As mentioned earlier, this presents multiple issues for updating the DFI protection, the constructed LSChains but also requires studying when the trade-off between the cost of spilling and the gain of optimization is worthwhile.

A final extension would be to improve the heuristic to select the LSChains to optimize when the pressure on the registers is too high. Our current heuristic, although showing interesting results in practice, can also lead to suboptimal solutions in some extreme cases. An example of such a case is presented in Figure 5.9.

In this example, we have three LSChains: one with four memory instructions based on the register **a1** (LS_1) and two other LSChains based on the register **t2** and which both contain three memory instructions (LS_2 and LS_3). LS_1 starts before and ends after LS_2 and LS_3. If there were only one register available for all these LSChains, our heuristic would choose to optimize LS_1, thus optimizing three instructions, whereas we would prefer to optimize LS_2 and LS_3 because that would optimize four instructions. In theory, we could add many more LSChains based on register **t2** and containing three memory instructions before LS_1 ends, reducing as much as we want the ratio between an optimal



(a) Control-Flow Graph with the LSChain per basic-block



(b) Resulting LSChain Graph

Figure 5.8: Example of a Control-Flow Graph with its load/store graph

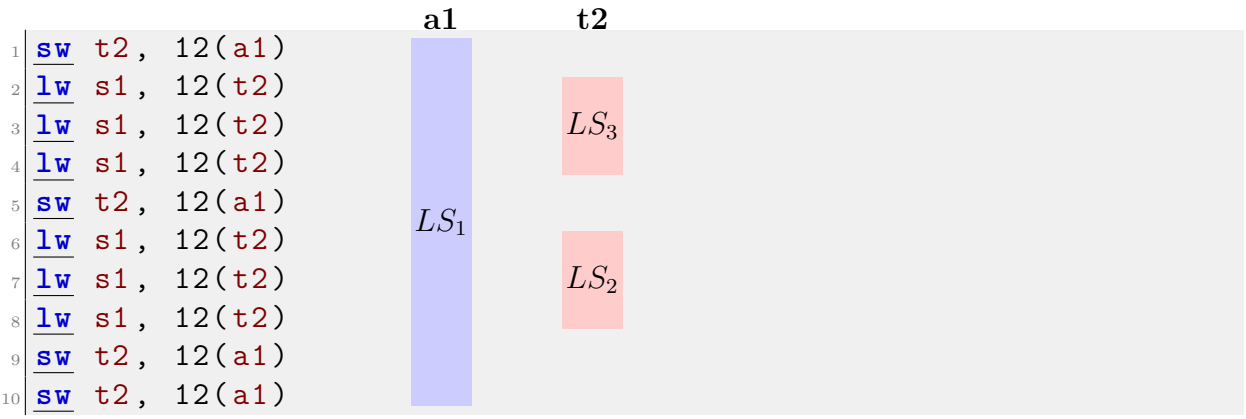


Figure 5.9: Example of worst-case behavior of the heuristic

solution and the solution provided by our heuristic. A significant improvement would be to find a heuristic with a guarantee on the produced solution compared to an optimal solution (in terms of number of optimized memory instructions). This requires, in particular, formalizing the optimization problem, finding a better heuristic and proving that it approximates an optimal solution by a given function.

5.5 Conclusion

In this chapter, we have shown how to optimize DFI by reducing redundancies between the instrumentation of multiple memory instructions. To do this, we first construct the LSChains at the basic-block level and use the available registers to optimize the instrumentation along these LSChains. We then presented an heuristic for managing register pressure without having to spill registers. We evaluated our optimizations on the TACLeBench benchmark suite and showed that they reduce the DFI overhead on the WCET by an average 14.4% and up to 40% on a few benchmarks. Finally, we identified possible improvements for this method, including using inter-basic-block LSChains, splitting LSChains when the register pressure is too high and finding an improved heuristic.

Chapter 6

Conclusion & Future Work

Chapter overview

In this chapter, we summarize our work, discuss the limitations of our approach, and we provide some ideas to lift these limitations and push our research.

6.1 Summary of the contributions

In this thesis, we proposed to adapt DFI protection to the constraints of real-time systems. A major issue of DFI for real-time systems is its overhead on the WCET. Thus, we focused our research on identifying the causes and reducing the overhead of DFI on the WCET.

We first presented our implementation of the DFI protection as well as a detailed analysis of its overhead in Chapter 3. This analysis showed that the overhead was mostly the result of three components of the DFI: the verification of the tags at each load, the computation of addresses in the RDT performed at each memory instruction and the verification before each store that the targeted address does not modify the program instructions nor the RDT.

In Chapter 4 we presented a first method that uses information collected when estimating the WCET coupled with an ILP solver to reduce the cost of the tag verification part. This optimization optimizes the representation of the tags belonging to the WCEP as well as the order of the checks at each load to reduce the overall number of checks on the WCEP which reduces the overhead of DFI on the WCET.

In Chapter 5 we discussed the presence of redundancies in the instrumentation inserted by the DFI. We then proposed a method to reduce this redundancy at the basic-bock level, by detecting chains of loads and stores based on a constant register. Identifying and removing these redundancies allows us to heavily reduce the overhead due to the RDT address computation but also (on a lesser extend) to remove some sandbox checks that are redundant. Thus, this method allows to tackle the overhead of the other two components

responsible for a large part of the DFI overhead.

6.2 Limitations & Future Works

The main limitation to our contributions is that we are unable to protect tasks that use shared data. Indeed, as our Data-Flow Graph Analysis constructs the graph for each task individually, it is unable to detect that shared data could be written by other tasks in the system. Furthermore, to avoid false-positive, the analysis would need to take into account all the possible interleaving of accesses to the shared data, which can greatly increase the complexity of such analysis and worsen the precision of the data-flow graph. On the other hand, if we were to protect a task without its shared data, this would create a hole in our protection, which the attacker could use undetected. While we still believe that our solution can be used to protect individual tasks in a system, it prevents us from protecting many tasks that uses shared data to communicate. This is particularly troublesome as an attacker able to corrupt a shared data could use it to attack other tasks in the system, effectively propagating its attack from task to task.

Thus, a first possible improvement would be to develop a DFG analysis that can deal with shared variables between multiple tasks. A first rough solution could use our current DFG analysis combined with a taint analysis with each shared variable tainted with a different value. Once each task have been analyzed, we could combine all the results to find the tag associated to each shared variables across all tasks and extend the valid tag set of every load using a given shared variable. However, such an analysis has a few issues. First, it would largely over-approximate the real behavior of the system as we would not take into account any constraint on the interleaving of the tasks. Second, each task would need distinct tags with distinct tag representations to ensure that an attacker can not use a tag in one task to confuse the protection in another task. Finally, that would mean that we could not optimize the tasks separately. Indeed, changing the tag representations in one task may create conflicts with the tag representations in another task and separate into more intervals the tag checks when loading shared variables of other tasks. Thus, we would need to modify our RT-DFI to optimize multiple tasks at once (potentially using information on the schedule). More generally, this issue requires more work to develop and assess the precision of such an analysis and ensure DFI can still be optimized.

These thesis contributions reduce the overhead of DFI to help integrate it on RTS. However, DFI protection remains an effective but costly protection. To further reduce this

cost, some research presented hardware-assisted DFI [Rac19] [FHL+22]. The major issue with these protections in our context is that they rely on new hardware caches and buffer that can stall the pipeline when there is a burst of memory accesses. The overhead of such stalls on the WCET remains to be evaluated. As an indication, Feng et al. [FHL+22] have a mean overhead of 17.6% with their hardware DFI on the average execution time, due to these stalls. In a RTS, such stalls have to be modeled to be taken into account by the WCET estimator. In particular, this would require a new static analysis of the code that could find potential stalls as well as their duration.

More generally, this thesis context is the security of real-time systems. We use information available specifically to real-time systems to improve the security of these systems and optimize the protection we adapt. We believe that using data such as the WCET or the scheduling to design targeted protections could be very interesting. For example, we also published a work, [BRP20], that protects a task by detecting fine-level timing deviation based on data extracted from the WCET estimator. Doing so, we can obtain a protection that protects against attacks that deviates the control-flow. Another example, that we did not pursue in this thesis, is to estimate the distribution of execution time of the tasks on the system and then use these distributions to detect abnormal patterns that could indicate that an attacker is modifying the normal behavior of the task. All these example transform the predictability of real-time systems, which is often considered an issue in security, into a strength that helps detect abnormal behavior and potential attacks.

Finally, in this thesis, we considered out of scope how the system reacted to the detection of an intrusion, considering that it could switch to a degraded mode to prevent future attack and protect its user and its environment. However, the question of how the system can survive past the attack, prevent further attacks and continue operating normally (and safely) is very important and further researches are required to find new methods for the systems to survive attacks while guaranteeing that all deadlines are met and the safety of its environment.

BIBLIOGRAPHY

[ABE+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson and Jay Ligatti. Control-flow Integrity, in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, ACM, 2005, ISBN: 978-1-59593-226-6, DOI: 10.1145/1102120.1102165, URL: http://doi.acm.org/10.1145/1102120.1102165 (visited on 27/11/2019), event-place: Alexandria, VA, USA.

[ACR+08] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa and Miguel Castro. Preventing Memory Error Exploits with WIT, in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277, May 2008, DOI: 10.1109/SP.2008.30, ISSN: 2375-1207.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*, PhD thesis, 1994.

[ano21] anonymous. Morris worm, en, November 2021, URL: https://en.wikipedia.org/w/index.php?title=Morris_worm&oldid=1053313243 (visited on 03/11/2021), Page Version ID: 1053313243.

[APM21] Abderaouf N Amalou, Isabelle Puaut and Gilles Muller. We-hml: hybrid wcet estimation using machine learning for architectures with caches, in *2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 31–40, IEEE, 2021.

[AvdWL+13] Fardin Abdi Taghi Abad, Joel van der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso and Sibin Mohan. On-chip control flow integrity check for real time embedded systems, in *1st IEEE International Conference on Cyber-Physical Systems, Networks, and Applications, CPSNA 2013, Taipei, Taiwan, August 19-20, 2013*, pages 26–31, IEEE Computer Society, 2013, DOI: 10.1109/CPSNA.2013.6614242, URL: https://doi.org/10.1109/CPSNA.2013.6614242.

[BBL14] Christian Bliek1ú, Pierre Bonami and Andrea Lodi. Solving mixed-integer quadratic programming problems with ibm-cplex: a progress report, in *Proceedings of the twenty-sixth RAMP symposium*, pages 16–17, 2014.

[BCD+17] Armelle Bonenfant, Denis Claraz, Marianne De Michiel and Pascal Sotin. Early wcet prediction using machine learning, in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[BCP03] Guillem Bernat, Antoine Colin and Stefan Petters. *pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems*, University of York, Department of Computer Science, 2003.

[BHJ12] Florian Brandner, Stefan Hepp and Alexander Jordan. Static profiling of the worst-case in real-time programs, in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 101–110, 2012.

[BRP20] Nicolas Bellec, Simon Rokicki and Isabelle Puaut. Attack detection through monitoring of timing deviations in embedded real-time systems, in Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, July 7-10, 2020, Virtual Conference*, volume 165 of *LIPIcs*, 8:1–8:22, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, DOI: 10.4230/LIPIcs.ECRTS.2020.8, URL: https://doi.org/10.4230/LIPIcs.ECRTS.2020.8.

[CAY+15] Paul Carsten, Todd R Andel, Mark Yampolskiy and Jeffrey T McDonald. In-vehicle networks: attacks, vulnerabilities, and proposed solutions, in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, pages 1–8, 2015.

[CCH06] Miguel Castro, Manuel Costa and Tim Harris. Securing Software by Enforcing Data-flow Integrity, in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, USENIX Association, 2006, ISBN: 978-1-931971-47-8, URL: http://dl.acm.org/citation.cfm?id=1298455.1298470 (visited on 17/12/2019), event-place: Seattle, Washington.

[CKM+19] Francisco J Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella and Tullio Vardanega. Probabilistic worst-case timing analysis: taxonomy and comprehensive survey, *ACM Computing Surveys (CSUR)*, 52*(1)*:1–35, 2019.

[CMP+19] Chien-Ying Chen, Sibin Mohan, Rodolfo Pellizzoni, Rakesh B. Bobba and Negar Kiyavash. A novel side-channel in real-time schedulers, in Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 90–102, IEEE, 2019, DOI: 10.1109/RTAS.2019.00016, URL: https://doi.org/10.1109/RTAS.2019.00016.

[CS06] Hugues Cassé and Pascal Sainrat. Otawa, a framework for experimenting wcet computations, in *Conference ERTS'06*, 2006.

[CW14] Nicholas Carlini and David Wagner. Rop is still dangerous: breaking modern defenses, in *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[CXC+19] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving, in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2267–2281, 2019.

[CXS+05] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[DFJ54] George Dantzig, Ray Fulkerson and Selmer Johnson. Solution of a large-scale traveling-salesman problem, *Journal of the operations research society of America*, 2*(4)*:393–410, 1954.

[DS16] Irene Díez-Franco and Igor Santos. Data is flowing in the wind: a review of data-flow integrity methods to overcome non-control-data attacks, in *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, pages 536–544, Springer, 2016.

[DS17] Irene Díez-Franco and Igor Santos. Data Is Flowing in the Wind: A Review of Data-Flow Integrity Methods to Overcome Non-Control-Data Attacks, en, in Manuel Graña, José Manuel López-Guede, Oier Etxaniz, Álvaro Herrero, Héctor Quintián and Emilio Corchado, editors, *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, Advances in Intelligent Systems and Computing, pages 536–544, Cham, Springer International Publishing, 2017, ISBN: 978-3-319-47364-2, DOI: 10.1007/978-3-319-47364-2_52.

[ELO+15] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity, in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, ACM, 2015, ISBN: 978-1-4503-3832-5, DOI: 10.1145/2810103.2813646, URL: http://doi.acm.org/10.1145/2810103.2813646 (visited on 17/12/2019), event-place: Denver, Colorado, USA.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann and Simon Wegener. Taclebench: a benchmark collection to support worst-case execution time research, in Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, 2:1–2:10, Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[FGG18] Joachim Fellmuth, Thomas Göthel and Sabine Glesner. Instruction caches in static wcet analysis of artificially diversified software, in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[FH04] Christian Ferdinand and Reinhold Heckmann. Ait: worst-case execution time prediction by static program analysis, in *Building the Information Society*, pages 377–383, Springer, 2004.

[FHL+22] Lang Feng, Jiayi Huang, Luyi Li, Haochen Zhang and Zhongfeng Wang. Rvdfi: a risc-v architecture with security enforcement by high performance

complete data-flow integrity, *IEEE Transactions on Computers*, 71*(10)*:2499–2512, 2022, DOI: 10.1109/TC.2021.3133701.

[FHP+17] Joachim Fellmuth, Paula Herber, Tobias F. Pfeffer and Sabine Glesner. Securing real-time cyber-physical systems using wcet-aware artificial diversity, in *15th IEEE Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2017, Orlando, FL, USA, November 6-10, 2017*, pages 454–461, IEEE Computer Society, 2017, DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88, URL: https://doi.org/10.1109/DASC-PICom-DataCom-CyberSciTec.2017.88.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems, *Real-time systems*, 17*(2)*:131–181, 1999.

[GGD+07] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking, in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 263–272, Springer, 2007.

[GNS09] Igor Griva, Stephen G Nash and Ariela Sofer. *Linear and nonlinear optimization*, volume 108, Siam, 2009.

[HRP17] Damien Hardy, Benjamin Rouxel and Isabelle Puaut. The heptane static worst-case execution time estimation tool, in *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[JMG+02] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[JP86] Mathai Joseph and Paritosh Pandya. Finding response times in a real-time system, *The Computer Journal*, 29*(5)*:390–395, 1986.

[JRZ10] Li Jie, Guo Ruifeng and Shao Zhixiang. The research of scheduling algorithms in real-time system, in *2010 International Conference on Computer and Communication Technologies in Agriculture Engineering*, volume 1, pages 333–336, IEEE, 2010.

[KL18] Jaeheon Kwak and Jinkyu Lee. Covert timing channel design for uniprocessor real-time systems, in Jong Hyuk Park, Hong Shen, Yunsick Sung and Hui Tian, editors, *Parallel and Distributed Computing, Applications and Technologies, 19th International Conference, PDCAT 2018, Jeju Island, South Korea, August 20-22, 2018, Revised Selected Papers*, volume 931 of *Communications in Computer and Information Science*, pages 159–168, Springer, 2018, DOI: 10.1007/978-981-13-5907-1_17, URL: https://doi.org/10.1007/978-981-13-5907-1%5C_17.

[KTF19] Marine Kadar, Sergey Tverdyshev and Gerhard Fohler. System calls instrumentation for intrusion detection in embedded mixed-criticality systems, in Mikael Asplund and Michael Paulitsch, editors, *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems, CERTS@ECRTS 2019, July 9, 2019, Stuttgart, Germany*, volume 73 of *OASICS*, 2:1–2:13, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, DOI: 10.4230/OASIcs.CERTS.2019.2, URL: https://doi.org/10.4230/OASIcs.CERTS.2019.2.

[LA04] Chris Lattner and Vikram Adve. Llvm: a compilation framework for lifelong program analysis & transformation, in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Pages 75–86, IEEE, 2004.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration, in *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, 1995.

[LTH02] Marc Langenbach, Stephan Thesing and Reinhold Heckmann. Pipeline modeling for timing analysis, in *International Static Analysis Symposium*, pages 294–309, Springer, 2002.

[MCG21] Tanmaya Mishra, Thidapat Chantem and Ryan M. Gerdes. Survey of control-flow integrity techniques for embedded and real-time embedded

systems, *CoRR*, abs/2111.11390, 2021, arXiv: 2111.11390, URL: https://arxiv.org/abs/2111.11390.

[MV15] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle, *Black Hat USA*, 2015.

[NB17] Mitra Nasri and Bjorn B Brandenburg. An exact and sustainable analysis of non-preemptive scheduling, in *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–23, IEEE, 2017.

[NCB+19] Mitra Nasri, Thidapat Chantem, Gedare Bloom and Ryan M. Gerdes. On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks, in Björn B. Brandenburg, editor, *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2019, Montreal, QC, Canada, April 16-18, 2019*, pages 103–116, IEEE, 2019, DOI: 10.1109/RTAS.2019.00017, URL: https://doi.org/10.1109/RTAS.2019.00017.

[NCH+05] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak and Westley Weimer. Ccured: type-safe retrofitting of legacy software, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27*(3)*:477–526, 2005.

[NZM+09] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c, in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[One96] Aleph One. Smashing the stack for fun and profit, *Phrack magazine*, 7*(49)*:14–16, 1996.

[Rac19] Abhijith Reddy Rachala. *Evaluation of Hardware-based Data Flow Integrity*, en, Thesis, July 2019, URL: https://oaktrust.library.tamu.edu/handle/1969.1/186252 (visited on 02/03/2020), Accepted: 2019-11-20T23:05:34Z.

[RMF+19] Federico Reghenzani, Giuseppe Massari, William Fornaciari and Andrea Galimberti. Probabilistic-wcet reliability: on the experimental validation of evt hypotheses, in *Proceedings of the International Conference on Omni-Layer Intelligent Systems*, pages 229–234, 2019.

[RNN22] Sayra Ranjha, Geoffrey Nelissen and Mitra Nasri. Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks, in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[SHB19] Philipp Dominik Schubert, Ben Hermann and Eric Bodden. Phasar: an inter-procedural static analysis framework for c/c++. In *TACAS (2)*, pages 393–410, 2019.

[SPP+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu and Dan Boneh. On the effectiveness of address-space randomization, in *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, Association for Computing Machinery, October 2004, ISBN: 978-1-58113-961-7, DOI: 10.1145/1030083.1030124, URL: https://doi.org/10.1145/1030083.1030124 (visited on 06/01/2022).

[SPW+13] Laszlo Szekeres, Mathias Payer, Tao Wei and Dawn Song. Sok: eternal war in memory, in *2013 IEEE Symposium on Security and Privacy*, pages 48–62, IEEE, 2013.

[TSH+03] S Thesing, J Souyris, R Heckmann, F Randimbivololona, M Langenbach, R Wilhelm and C Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software, in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* Pages 625–632, IEEE, 2003.

[vdVAS+17] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos and Cristiano Giuffrdia. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1675–1689, New York, NY, USA, ACM, 2017, ISBN: 978-1-4503-4946-8, DOI: 10.1145/3133956.3134026, URL: http://doi.acm.org/10.1145/3133956.3134026 (visited on 28/11/2019), event-place: Dallas, Texas, USA.

[VHH08] Marcus Völp, Claude-Joachim Hamann and Hermann Härtig. Avoiding timing channels in fixed-priority schedulers, in Masayuki Abe and Virgil D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information,*

Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008, pages 44–55, ACM, 2008, DOI: 10.1145/1368310.1368320, URL: https://doi.org/10.1145/1368310.1368320.

[VPK+19] Nils Vreman, Richard Pates, Kristin Krüger, Gerhard Fohler and Martina Maggio. Minimizing side-channel attack vulnerability via schedule randomization, in *58th IEEE Conference on Decision and Control, CDC 2019, Nice, France, December 11-13, 2019*, pages 2928–2933, IEEE, 2019, DOI: 10.1109/CDC40024.2019.9030144, URL: https://doi.org/10.1109/CDC40024.2019.9030144.

[VSM+22] Sergi Vilardell, Isabel Serra, Enrico Mezzetti, Jaume Abella, Francisco J Cazorla and Joan del Castillo. Using markov's inequality with power-of-k function for probabilistic wcet estimation, in *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[WBB+19] Robert J. Walls, Nicholas F. Brown, Thomas Le Baron, Craig A. Shue, Hamed Okhravi and Bryan C. Ward. Control-Flow Integrity for Real-Time Embedded Systems, in Sophie Quinton, editor, *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, volume 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 2:1–2:24, Dagstuhl, Germany, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019, ISBN: 978-3-95977-110-8, DOI: 10.4230/LIPIcs.ECRTS.2019.2, URL: http://drops.dagstuhl.de/opus/volltexte/2019/10739.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra et al. The worst-case execution-time problem-overview of methods and survey of tools, *ACM Transactions on Embedded Computing Systems (TECS)*, 7*(3)*:1–53, 2008.

[WFU+12] Julian Wolf, Bernhard Fechner, Sascha Uhrig and Theo Ungerer. Fine-grained timing and control flow error checking for hard real-time task execution, in *7th IEEE International Symposium on Industrial Embedded Systems, SIES 2012, Karlsruhe, Germany, June 20-22, 2012*, pages 257–266, IEEE, 2012, DOI: 10.1109/SIES.2012.6356592, URL: https://doi.org/10.1109/SIES.2012.6356592.

[WWN+15] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie et al. Cheri: a hybrid capability-system architecture for scalable software compartmentalization, in *2015 IEEE Symposium on Security and Privacy*, pages 20–37, IEEE, 2015.

[YMC+16] Man-Ki Yoon, Sibin Mohan, Chien-Ying Chen and Lui Sha. Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems, in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 111–122, IEEE Computer Society, 2016, DOI: 10.1109/RTAS.2016.7461362, URL: https://doi.org/10.1109/RTAS.2016.7461362.

[YMC+17] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu and Lui Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system, in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017*, pages 191–196, 2017, DOI: 10.1145/3054977.3054999, URL: https://doi.org/10.1145/3054977.3054999.

[YNB19] Beyazit Yalcinkaya, Mitra Nasri and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks, in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233, IEEE, 2019.

Appendix A

Experimental setups

In this appendix, we present all the configurations we use for our experiments. We start by presenting the benchmarks in Section A.1. The software tools are presented in Section A.2 while the hardware we use is presented in Section A.3.

A.1 Benchmarks

For our experiments, we use TACLeBench [FAH+16]. This benchmark suite is composed of five groups of benchmarks:

1. **kernel** contains small kernel functions such as a binary search or a md5 hash.
2. **sequential** contains large function blocks (e.g. cryptographic or compression algorithms).
3. **test** contains three synthetic programs designed to challenge the WCET analysis tools.
4. **parallel** contains two modified real-world parallel applications, *Debie* and *PapaBench*.
5. **app** contains two real applications, *lift* and *powerwindow*.

A full benchmark description can be found in [FAH+16]. Out of the five groups, we did not run our experiments on the **parallel** group as we consider a bare-metal execution of the programs without relying on a Real-Time Operating System (RTOS), which defeats the purpose of the **parallel** group. We also excluded the *bitonic*, *bitcount*, *fac*, *quicksort*, *recursion*, *ammunition*, *anagram* and *huff_enc* benchmarks as it was hard to obtain the recursion bounds for the *AiT* WCET solver. Furthermore, *rijndael_dec* and *rijndael_enc* violates the DFI principle due to a use-before-initialization error and thus, we cannot obtain a trace for them.

A.1.1 Notes on the security

Even without mounting a real attack scenario, RT-DFI detected errors on three benchmarks. The first two errors were detected by *AiT* in *rijndael_dec* and *rijndael_enc*. They are due to an off-by-one read to a buffer that loads a stack-saved register and triggers DFI. As *AiT* detects this in its value analysis, it considers that DFI exception is triggered and that its WCET analysis is unsafe. This means that providing *AiT* with a DFI-protected program can help find bugs that can be corrected before reaching the market. The third error appeared when executing *sha* using *Qemu* to test our DFI implementation. DFI detected a read to a saved register when executing the function *sha_wordcopy_fwd_aligned*. This function writes, at each iteration, a word into a buffer and then reads the word for the next iteration. Thus, at the last iteration, a word is read past the buffer, which triggers DFI. Note that this error is detected at the execution by DFI but not when using the analysis of *AiT*. This shows that even simple benchmarks can have non-trivial errors that are not detected, and the requirement for improved security protection in RTS.

A.2 Software

We list here all the software used for our experiments.

- **LLVM / Clang** version 10.0.0, a compiler toolchain written in C++, modified with about 8000 lines of code (LoC) to emit the DFI and its optimizations (already existing and presented in this thesis). We compile the benchmarks with -O1 optimization flags.
- **PhASAR**, a data-flow analysis framework on the LLVM IR written in C++, modified with about 1000 LoC to obtain the DFG we need for DFI.
- **Qemu**, a fast emulator used to execute programs compiled for RISC-V systems without a RISC-V processor.
- **Comet**, a simulator of RISC-V processor. We use this software to efficiently execute the RISC-V software step by step to obtain execution traces.
- ***AiT***, a static WCET estimator. To estimate the WCET of our compiled benchmarks, we have to provide a few annotations. Provided the following annotations:
  - We used the base unrolling factor (that is equal to 2) for the loops.

– We provided the loop bounds for each loop in the benchmarks.

– For DFI protected binary, we added annotations that prevent *AiT* from considering raising an exception (for the DFI) as the normal behavior of the program, such that *AiT* does not take into account this case in the WCET.

A.3 Hardware

We target *RudolV*, a 32bits RISCV softcore dedicated to hard real-time systems with the M extension (allowing multiplications and integer divisions). *RudolV* executes each instruction in a precise number of cycles and avoids caches and branch prediction as they worsen the predictability of the architecture. The cost of each instruction is presented in Table A.1.

Instruction class	Examples	Cycles
arithmetic	ADD, SUB, LUI	1
barrel shifter	SLL, SRL	1
not taken branch	BEQ, BLT	1
memory access	LH, LW, SH, SW	2
CSR access	CSRRW, CSRRS, CSRRC	2
taken branch	BEQ, BNE, BLT	3/4
unconditional jump	JAL, JALR	3/4
exception	ECALL, EBREAK	3/4
RV32M	MUL, DIV, DIVU, REM, REMU	35

Table A.1: Cycle to execute the instructions in the RudolV architecture (Extracted from https://github.com/bobbl/rudolv)

List of Tables

List of Figures

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Amélioration de la sécurité des systèmes embarqués, temps réels et critiques

Mot clés : Système temps réel, Sécurité, Corruption mémoire, Intégrité du flux de donnée

Résumé : Les systèmes temps-réels embarquent de plus en plus de moyen pour communiquer sans fils avec des utilisateurs extérieurs. Ces mêmes moyens peuvent être détournés pour attaquer ces systèmes, brisant les garanties de ces derniers et pouvant engendrer des accidents. Pour protéger les systèmes temps-réels contre ces nouvelles attaques, il est nécessaire de développer de nouvelles protections prenant en compte les spécificités de ces systèmes.

Dans cette thèse, nous cherchons à améliorer la sécurité des systèmes temps-réels contre des attaques dites par corruption de mémoire. Ces attaques utilisent une mauvaise gestion de la mémoire dans un programme pour modifier son comportement. Nous nous intéressons en particulier à une défense appelée Intégrité du flux de donnée, qui peut protéger contre une vaste classe d'attaque par corruption de mémoire. Nous adaptons cette protection au contexte des systèmes temps-réels en optimisant le temps d'exécution dans le pire cas, une métrique fondamentale pour garantir la bonne exécution de ces systèmes.

Title: Security enhancement in embedded hard real-time systems

Keywords: Real time systems, Security, Memory corruption, Data-Flow Integrity

Abstract: Real-time systems have more and more ways to communicate wirelessly with external users. These same means can be hijacked to attack these systems, breaking their guarantees and potentially causing accidents. To protect real-time systems against these new attacks, it is necessary to develop new protections taking into account the specificities of these systems.

In this thesis, we seek to improve the security of real-time systems against so-called memory corruption attacks. These attacks use a bad memory management in a program to modify its behavior. We are particularly interested in a defense called Data Flow Integrity, which can protect against a large class of memory corruption attacks. We adapt this protection to the context of real-time systems by optimizing the worst-case execution time, a fundamental metric to ensure the correct execution of these systems.