Techniques de test pour des critères de couverture avancés

Thibault Martin

université PARIS-SACLAY

Techniques de test pour des critères de couverture avancés

Testing techniques for advanced test coverage criteria

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique

Thèse préparée dans l'unité de recherche de l'Institut LIST (Université Paris-Saclay, CEA) sous la direction de Nikolai KOSMATOV, ingénieur-chercheur et le co-encadrement de Virgile PREVOSTO, ingénieur-chercheur.

Thèse soutenue à Paris-Saclay, le 30 novembre 2022, par

Thibault Martin

Composition du jury

Pascale Le Gall Professeure, CentraleSupélec, Université Paris-Saclay	Présidente
Roland Groz Professeur, Grenoble INP, Ensimag, Université Grenoble Alpes	Rapporteur & Examinateur
Ioannis Parissis Professeur, Grenoble INP, Esisar, Université Grenoble Alpes	Rapporteur & Examinateur
Antoine Rollet Maître de conférences, Bordeaux INP, Enseirb-matmeca, Université de Bordeaux	Examinateur
Nikolai Kosmatov Ingénieur-chercheur, CEA List, Université Paris-Saclay & Thales Research and Technology	Directeur de thèse

THESE DE DOCTORAT

NNT : 2022UPASG077

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Techniques de test pour des critères de couverture avancés
Mots clés : Méthodes formelles, critères de couverture, génération de tests, injection de fautes

Résumé : La vérification de logiciels représente un défi important à l'heure où ces derniers sont présents partout dans notre vie quotidienne. La technique la plus employée pour s'assurer qu'un logiciel répond à certaines exigences reste le test, c'est à dire exécuter le programmes sur un certain nombres d'entrées représentatives. Un enjeu majeur est alors de s'assurer que ces jeux d'entrée couvrent suffisamment de situations différentes. De nombreuses propositions ont été faites, appelées critères de couvertures, qui définissent des objectifs de tests, par lesquels doit passer au moins une exécution au cours du test. Dans cette thèse, nous examinons différentes possibilités pour combiner les méthodes formelles (techniques mathématiques) et le test, dans l'objectif d'améliorer l'efficacité de ce dernier. Nous nous sommes principalement intéressés aux critères de flot de données, qui observent le comportement des définitions et utilisations des variables du programme, et en particulier à la détection des objectifs polluants. Nous proposons plusieurs techniques, avec leurs implémentations et évaluations sur des exemples de programmes C réels. Nous étudions ensuite comment réaliser une génération de tests efficace pour les objectifs de flot de données. Enfin, nous terminons en concevant une méthode permettant d'utiliser la détection d'objectifs polluants pour vérifier l'efficacité des contre-mesures contre les attaques par injection de fautes, et de trouver ainsi les erreurs dans leur implémentation. Cette méthode a été appliquée avec succès sur le bootloader de WooKey, un périphérique de stockage chiffré.

Title : Testing techniques for advanced test coverage criteria
Keywords : Formal methods, coverage criteria, test generation, fault injection

Abstract : At a time where programs are omnipresent in our daily life, verifying their safety and security presents an important challenge for the industry. The most used technique to ensure that a software meets certain requirements is testing, where we run the program with some controlled input sets. A major issue is then to ensure that these input sets cover enough different situations. Many proposals have been made, called coverage criteria, which define test objectives through which at least one execution must pass during tests. In this thesis, we examine different possibilities to combine formal methods (mathematical techniques) and testing in order to improve the efficiency of the testing process. We focused our efforts on dataflow criteria, which observe the behavior of definitions and uses of each variable in the program, and the detection of polluting objectives for these criteria. We propose several techniques, with their implementations and evaluations on real C programs. We then study how to efficiently realize test generation for dataflow criteria. Finally, we conclude by designing a method, using polluting objectives detection, to verify the effectiveness of countermeasures against fault injection attacks, and thus find errors in their implementation. This method was successfully applied to the bootloader of WooKey, an encrypted storage device.

Remerciements

Je voudrais commencer par remercier Roland Groz et Ioannis Parissis d'avoir accepté de rapporter ma thèse. Leur lecture attentive et les retours qu'ils m'ont fait ont permis de corriger des erreurs et imprécisions dans le manuscrit. Merci à Antoine Rollet d'avoir accepté de faire partie du jury de la soutenance et également merci à Pascale Le Gall d'avoir accepté de le présider.

Un grand merci à mes deux encadrants pour leur accompagnement tout au long de mon stage et de la thèse qui en a découlé, et notamment les très nombreuses réunions bien trop matinales pour moi. Dans un premier temps Virgile Prevosto, mon encadrant au CEA, et en particulier son aide (extrêmement précieuse) pour comprendre et utiliser Frama-C, ainsi que les nombreuses conversations techniques pour réaliser les différentes implémentations de cette thèse. Et Nikolai Kosmatov, mon directeur de thèse, qui malgré sa migration du CEA vers Thalès aura su rester très présent (beaucoup de réunions je disais) pour m'aider à avancer dans les moments difficiles, et pour ses talents de rédaction.

Je remercie mes professeurs de l'université pour m'avoir encouragé à poursuivre mes études, et en particulier Matthias Roux et Sylvain Conchon pour m'avoir transmis l'amour du langage de programmation OCaml, déterminant pour le choix de mon stage, et donc sans lequel je ne serais pas arrivé jusqu'ici.

C'est au CEA que j'ai rencontré Virgile R., mon co-bureau et maintenant ami, pendant le stage, qui par pur hasard s'est trouvé être mon co-bureau pendant la thèse. Merci pour ces presque cinq années, beaucoup de discussions, d'inspiration, de soutien et d'entraide, et surtout beaucoup de fous rires ! Merci également à Alexandre pour son accueil au CEA et Romain qui, même si on ne s'est pas beaucoup vus à cause des confinements et du télétravail, a été d'une compagnie très agréable.

J'aimerais rendre hommage aux deux personnes qui avec moi formaient le triumvirat de la pause café, Pauline et Florent. Beaucoup de discussions sérieuses et moins sérieuses, de détente, de soutien moral, et d'usure des canapés.

Je tiens également à remercier les (post-)doctorant·e·s, stagiaires et permanent·e·s du laboratoire, qui ont fortement contribué à rendre la vie du laboratoire très agréable, un point très important dans la décision d'y faire ma thèse. Dans le désordre : David, Maxime, Valentin, François, Lesly, Yaëlle, André, Michele, Augustin, Louis, Florent K., Grégoire, Adrien, Basile, Christophe, Olivier, Myriam, Dario, Julien & Julien, Aymeric, Zak, Soline, Dorian, Guillaume et bien d'autres... Mention spéciale à Allan, pour les regards en coin suivant nos références obscures. Merci à tous et à toutes, pour les pauses café, les sorties et soirées, les randonnées, et de manière générale la vie dans et en dehors du laboratoire. Merci également à Frédérique et Ewa, et toute l'aide et le travail qu'elles fournissent pour que le laboratoire fonctionne aussi bien.

Je remercie tous les ami·e·s, dont certain·e·s déjà cité·e·s, qui m'ont entouré avant, pendant et après la thèse. Florian (qui est maintenant mon confrère) et Vincent, Jean-Jacques, Sébastien, Sofian, Josh et Nicolas.

Pour finir, je remercie ma famille. Ma mère, mon frère Killian et ma sœur Louise pour m'avoir toujours supporté (dans tous les sens du terme). Mon père et Agnès pour leur accueil quand j'avais besoin de me changer les idées. Mon cousin Loïc, qui a fait sa thèse 1 an avant la mienne, et tous les conseils qu'il m'a donnés pour y arriver, sans oublier Valérie, Olivier et Maud. Et enfin, mes grands-parents, pour tout leur amour et leur bienveillance, leurs encouragements. En particulier merci à mon grand-père François, décédé en décembre 2021, qui était si fier de moi et qui rêvait de me voir réussir.

Table des matières

Table des figures

Liste des algorithmes

Chapitre 1

Introduction

L'histoire de l'informatique a connu de nombreux dysfonctionnements et accidents. Parmi les exemples les plus connus, en 1991, une erreur logicielle dans le système Patriot [Gen92] a causé la mort de 28 américains en ratant l'interception d'un missile ennemi à cause d'une erreur de calcul. En 1996, un bug dans le système de guidage de la fusée Ariane 5 [Lio+96] a conduit à son autodestruction. En 2014, une vulnérabilité dans le logiciel OpenSSL nommée HeartBleed [Dur+14] était rendue publique, compromettant la sécurité de nombreux services et sites internet. Plus récemment en 2021, une erreur de communication dans le logiciel de certaines voitures Tesla était découverte et pouvait avoir comme effet de signaler une fausse alerte de collision frontale ou de déclencher le freinage d'urgence du véhicule. Près de 12000 voitures ont été rappelées.

Dans les dernières décennies, l'essor de l'informatique et d'internet a grandement bouleversé nos modes de vie. Nos voitures, avions, trains, sont équipés de systèmes informatiques puissants. Nous pilotons certaines de nos centrales nucléaires avec des logiciels et nous sauvons même des vies grâce à des équipements médicaux avancés. Avec la miniaturisation du matériel informatique et grâce à internet, la majorité de la population possède et garde avec elle un ordinateur miniature faisant office de téléphone, console de jeu, outil de travail, livre, etc. Ce sont même tous nos objets du quotidien qui peuvent posséder des systèmes embarqués les reliant à internet et permettant leur contrôle à distance : l'Internet des Objets [Lee+15] (Internet of Things, IoT).

Naturellement, tous ces systèmes sont conçus et développés par des humains qui sont susceptibles de commettre des erreurs lors de leur conception ou de leur l'implémentation. On trouve sur internet un grand nombre de sites pour signaler des bugs et trouver de l'aide pour les corriger. Les utilisateurs sont, pour beaucoup, habitués à rencontrer des bugs, qui sont souvent des dé-

fauts mineurs sans impact plus fort qu'un léger désagrément.

Malheureusement, comme évoqué ci-dessus, certains de ces bugs peuvent avoir des conséquences très graves et coûteuses pour ceux qui en sont victimes. Cela peut aller de lourds dégâts matériels à des problèmes de sécurité et fuites de données, le tout pouvant induire des coûts financiers importants, voire dans le pire cas, des problèmes de sûreté mettant en danger des vies. Beaucoup d'autres bugs, avec des conséquences plus ou moins graves, ont été recensés au fil des années [Wik22]. Lorsqu'on s'intéresse à des systèmes critiques, les développeurs et les utilisateurs ont besoin d'avoir des garanties fortes sur ces derniers, et en particulier sur les logiciels qui y sont utilisés. De nombreuses techniques de validation et de vérification, allant du test aux méthodes formelles, ont été mises en place afin garantir le plus possible la sécurité et la sûreté des logiciels.

1.1 Validation et vérification de logiciels

Le test est une technique très utilisée pour valider les logiciels. On exécute le programme en choisissant des entrées pour observer son comportement. Cependant, cette technique n'est pas exhaustive, on ne peut pas tout tester, et elle dépend donc du choix des données de tests considérées. Il est donc possible que le programme passe tous les tests, mais que ces derniers ne testent pas un comportement particulier (peut-être rare) menant à une erreur.

En parallèle du test, les méthodes formelles sont des techniques de spécification et de vérification utilisées pour vérifier la correction des programmes par des techniques mathématiques et garantir un haut niveau de confiance. La première étape est donc de donner une spécification du programme, c'est-à-dire déterminer quels sont les comportements acceptables ou non pour ce programme. Cette spécification peut être partielle. Ensuite, à l'aide de différentes méthodes, il faut prouver les propriétés qui ont été spécifiées. Cependant, depuis la preuve de l'indécidabilité du problème de l'arrêt [Tur37] par Alan Turing, suivi de sa généralisation avec le théorème de Rice [Ric53], on sait qu'une propriété sémantique non triviale d'un programme est indécidable. En d'autres termes, étant donné un programme avec des spécifications (définissant les comportements attendus du programme), cela veut dire qu'il est impossible de concevoir une technique de vérification de programme avec toutes les propriétés suivantes :

— **Correction**. Si l'analyse conclut que le programme respecte les spécifications, alors c'est effectivement le cas.

— **Complétude**. Si le programme respecte les spécifications, alors l'analyse conclura que c'est bien le cas.

— **Automatisation.** Aucune aide extérieure n'est nécessaire à l'analyse qui requiert seulement le programme à vérifier et ses spécifications.

Nous avons donné les définitions de correction et complétude pour une technique de vérification. Elles peuvent être définies également pour une technique de recherche d'erreurs. Ces notions seront discutées plus en détail dans la Section 2.4.

Puisqu'il est impossible de créer une analyse avec ces trois propriétés, il faut donc en ignorer au moins une. Par exemple, le *test de programme* ne prouve pas la correction, c'est-à-dire l'absence de bugs, car il est en général impossible de tester toutes les entrées possibles.

Par contraste avec le test, les méthodes formelles ont, de manière générale, tendance à privilégier la correction, en abandonnant soit la complétude, soit l'automatisation. Parmi ces techniques, on trouve la *vérification déductive* [Hoa69] qui permet, en s'appuyant sur des prouveurs automatiques et interactifs, de prouver la correction d'un programme vis-à-vis d'une spécification fournie par l'utilisateur (contrats de fonctions, invariants de boucles, etc.). Cette technique n'est pas automatique, car l'utilisateur doit fournir des spécifications. De plus, même si la preuve peut s'appuyer sur des prouveurs automatiques, l'utilisateur peut être amené à faire des preuves interactives. L'*interprétation abstraite* [Cou+77] utilise des domaines abstraits pour donner une sur-approximation de l'ensemble des valeurs que peut prendre chaque variable en faisant l'union des valeurs possibles de toutes les exécutions. Cette technique est correcte et automatique, mais incomplète, car il y a un risque d'erreurs : elle génère des *fausses alarmes*, c'est-à-dire qu'elle peut détecter un problème potentiel là où il n'y en a pas. L'*exécution symbolique* [Kin76] quant à elle permet de remplacer les variables d'entrée du programme par des variables symboliques et explorer (*exécuter symboliquement*) le programme dans l'objectif de générer des contraintes permettant d'exécuter tel ou tel chemin. Ensuite, à partir des formules logiques obtenues (ou *prédicats de chemins*) qui contiennent ces contraintes sur l'ensemble des branches du chemin, on pourra instancier les variables symboliques par des valeurs concrètes satisfaisant une formule en particulier, dans le but d'exécuter le chemin correspondant.

Ces techniques sont appelées des *analyses statiques* : on raisonne sur le programme sans l'exécuter. Elles s'opposent à l'*analyse dynamique* qui au contraire nécessite d'exécuter le programme pour raisonner dessus. En outre, des combinaisons de techniques statiques et dynamiques ont été également proposées, par exemple, l'*exécution symbolique dynamique* (Dynamic symbolic execution ou DSE en anglais) reprend les résultats de l'exécution symbolique et exécute

le programme et sera alors potentiellement capable de trouver des erreurs supplémentaires (crash, assertions violées, exceptions non rattrapées, etc). Enfin, la mesure de couverture de code par les tests est également un moyen d'évaluer l'efficacité du processus de vérification qui a été mené sur un programme, et elle nécessite l'exécution du programme.

1.2 Motivation

Dans le monde du développement de logiciels, le test est une étape importante pour vérifier le comportement du programme sur des entrées données. Les principaux processus de test sont la création d'objectifs de test, la génération de tests ou encore la mesure de couverture des objectifs par une suite de tests. Afin d'effectuer les tests de manière pertinente, il est crucial de définir ce que l'on souhaite vérifier grâce aux tests, nous permettant de valider ou non leurs résultats, et de s'assurer que le jeu de tests simule un ensemble de situations suffisamment large pour être représentatif de tous les cas possibles. Dans leur livre, Ammann et Offutt [Amm+16] décrivent plusieurs dizaines de critères de couverture de code ayant pour but d'aider les processus de test en fournissant des objectifs à atteindre pour valider une suite de test vis-à-vis des critères choisis. Ces derniers permettent également de guider la génération de tests pour minimiser une suite de tests en gardant le moins de cas de tests possible tels que tous les objectifs sont couverts par la suite de tests.

Cependant, ces critères sont très nombreux et hétérogènes : critères de flot de données, critères logiques, mutations, sur une ou plusieurs traces d'exécution, etc. De nombreux outils permettent d'utiliser un sous-ensemble de ces critères, souvent très réduit, et proposent de réaliser un service spécifique (génération de tests ou mesure de couverture de code). Ces critères sont souvent implémentés en dur dans le logiciel, rendant le support de nouveaux critères compliqué, voire impossible sans restructurer le code.

Un autre problème est la présence d'objectifs de test dits polluants. Ces objectifs ont la particularité de gêner, de différentes façons, les processus de test en ralentissant, par exemple, la génération de tests ou en rendant moins précis les résultats obtenus par la mesure de couverture de code. Comment détecter les objectifs de test polluants, c'est-à-dire qui ne peuvent pas être couverts par les tests ?

Les paires def-use sont une brique essentielle dans la définition des critères flot de données. On s'intéresse aux couples définitions-utilisations de variables. La Figure 1.1 donne un exemple de code écrit en C. Dans cette fonction, on note la présence d'une unique variable `x`. Cette variable est définie (c'est-à-dire

```
1  int main(int cond) {
2    int x = 40;
3    if (cond) {
4      x = x + 1;
5    else {
6      x = x + 2;
7    }
8    return x;
9  }
```

FIGURE 1.1 : Exemple de programme C pour illustrer les paires def-use

qu'elle est assignée) aux lignes 2, 4 et 6, et utilisée (c'est-à-dire que sa valeur est lue) aux lignes 4, 6 et 8. On peut noter (l, l') une paire allant de la définition à la ligne l vers une utilisation à la ligne l'. Ayant 3 définitions et 3 utilisations, on peut alors créer un total de 9 paires def-use :

$$\begin{array}{lll} 1 : (2,4), & 2 : (2,6), & 3 : (2,8), \\ 4 : (4,4), & 5 : (4,6), & 6 : (4,8), \\ 7 : (6,4), & 8 : (6,6), & 9 : (6,8). \end{array}$$

Cependant, on peut remarquer que certaines paires sont problématiques. Par exemple, les paires 4, 7 et 8 ne sont pas réalisables, car elles nécessitent un retour en arrière, remonter dans le graphe de flot de contrôle, ce qui n'est ici pas possible. De même, la paire 5 demande à visiter les deux branches successivement, et pour les mêmes raisons, c'est ici impossible. On pourrait imaginer également que la variable `cond` soit toujours égale à 0, rendant l'accès à la branche positive (ligne 4) impossible, et donc par conséquent la paire 6 deviendrait également impossible à réaliser (on aurait du code mort, impossible à exécuter). Toutes ces paires sont appelées des *objectifs polluants*.

On a mentionné que les paires def-use étaient les briques de base des critères de flot de données, par conséquent la détection des objectifs de flot de données polluants commence par une analyse des paires def-use. Dans cette thèse, et en particulier dans le Chapitre 4, nous nous intéresserons à la détection des paires def-use polluantes en utilisant la vérification déductive, l'interprétation abstraite et également l'analyse de flot de données. Nous verrons également comment cette détection d'objectifs polluants peut être utilisée pour le domaine de l'injection de fautes et la vérification des sécurités dans les logiciels critiques. Enfin, nous étudierons de nouvelles idées de critères de couverture de code, ainsi que des solutions pour leur support efficace lors de la génération de tests.

Afin de réaliser les processus de test dans de bonnes conditions, on a besoin de prendre en compte plusieurs notions importantes dans le monde du test. Tout au long de cette thèse, nous avons eu pour objectif de mettre en avant l'efficacité de nos méthodes, leur généricité et leur accessibilité.

Efficacité des processus de tests et passage à l'échelle. Comme on l'a mentionné, les tests s'effectuent en passant par plusieurs étapes cruciales, de la définition des objectifs de test que l'on souhaite couvrir, à la création de tests et à leur validation. On veut donc s'assurer que chaque étape est réalisée dans les meilleures conditions possibles afin de garantir la qualité d'une suite de tests avec la meilleure couverture possible selon les critères choisis. Par exemple, si les objectifs à atteindre pour valider les tests sont mal définis, on perd alors l'intérêt de cette validation. On a également mentionné les objectifs polluants, il est important de pouvoir les identifier le plus tôt possible dans la chaîne de test afin d'éviter aux services suivants de perdre en performance. Sur des petits programmes, l'exécution des tests est généralement rapide, de même que la génération de ces tests ou la mesure de couverture. Cependant, sur de plus gros programmes (par exemple des logiciels industriels), la performance des outils devient centrale : on veut pouvoir effectuer tous les services de la chaîne de test (création d'objectifs, détection d'objectifs polluants, génération de tests, mesure de couverture) dans un temps raisonnable : nos méthodes doivent être applicables à de vraies applications. Nous nous intéresserons donc aux programmes dans un langage de programmation très répandu : le langage de programmation C.

Accessibilité et automatisation. On a parlé précédemment de la notion d'automatisation dans le cadre de la vérification formelle des logiciels. Pour permettre à n'importe quel utilisateur d'utiliser nos techniques de test avancées, nous avons décidé de tout mettre en place pour réduire le plus possible la charge laissée à ce dernier. Tous les processus de test doivent être les plus automatiques possibles, ne demandant que de fournir les objectifs que l'on souhaite valider avec les tests. On souhaite que l'utilisateur puisse utiliser nos approches sans expertise particulière sur l'utilisation des méthodes formelles.

Généricité. Nous avons vu qu'il existait un grand nombre de critères, et d'outils supportant peu d'entre eux. La généricité est donc un objectif important pour permettre le support du plus grand nombre de critères possible sur l'ensemble des processus de test. Notre objectif est de mettre au point des techniques applicables au plus grand nombre de critères de couverture de code

possible, et ainsi permettre la création de nouveaux critères sans avoir besoin d'adapter toutes les techniques à chaque fois.

Dans cette thèse, nous continuons les travaux menés sur la mise en place d'approches génériques, automatiques et efficaces sur les critères logiques [Bar+14a ; Mar+17a] pour permettre leurs utilisations sur des critères plus complexes comme les critères de flot de données. Nous intégrerons nos techniques et analyses dans une boîte à outils prenant en compte différents critères, et nous étudierons et comparerons les approches pour trouver les plus performantes. Nous utiliserons nos techniques pour mettre en place de nouveaux critères, et également pour vérifier des mesures de sécurité dans des programmes critiques.

Les techniques proposées dans cette thèse ont été implémentées dans la plateforme de vérification de code C Frama-C [Kir+15 ; Bau+21a], l'outil de génération de tests PathCrawler [Wil+05] et la boîte à outils pour le test LTest [Bar+14a ; Mar+17b].

1.3 Contributions et structure du document

Contributions. Les contributions principales de cette thèse sont :

- une formalisation d'un petit langage Mini-C dans l'objectif de simplifier certaines notions pour décrire le flot de données dans un graphe de flot de contrôle et ainsi exprimer les critères de flot de données et comment manipuler ces critères ;
- la description formelle de ce que l'on appelle des objectifs de test polluants pour les critères de flot de données, à savoir les objectifs non-applicables, les objectifs équivalents et les objectifs infaisables, pour proposer et étudier ensuite des techniques permettant d'en détecter le plus possible. On se concentre sur la brique de base de ces critères, les paires définition-utilisation (def-use) ;
- la proposition de techniques de génération de tests, à base d'exécution symbolique dynamique avec l'outil PathCrawler (voir Section 2.5), pour les critères de flot de données. Nous allons introduire deux techniques particulières d'instrumentation pour deux approches d'évaluation des objectifs de flot de données, appelées *évaluation immédiate* et *évaluation retardée*. On s'est intéressé en particulier à deux types de critères plus complexes : les critères de test de flot de données aux limites et la visibilité des objectifs de test dans la sortie des fonctions ;
- l'implémentation des techniques précédentes dans plusieurs greffons de Frama-C (voir Section 2.4). Nous avons implémenté les techniques de dé-

tection des objectifs polluants dans les greffons LAnnotate et LUncov (voir Section 2.4.5), permettant à un utilisateur d'instancier des objectifs pour des critères de flot de données sans objectifs non-applicables et équivalents, et de lancer une analyse automatique pour détecter des objectifs infaisables. Nous avons également implémenté dans l'outil PathCrawler les techniques de génération de tests pour les critères de la visibilité et de flot de données aux limites. Ces techniques ont ensuite été évaluées sur plusieurs exemples de programmes C afin de comparer les différentes approches et de trouver le meilleur compromis entre capacité de détection et temps de calcul ;

- la proposition d'une méthode utilisant la vérification déductive avec Wp sur les objectifs polluants pour prouver l'efficacité d'un type de contre-mesures contre les attaques par injection de fautes. Notre implémentation dans la boîte à outils LTest (voir Section 2.4.5) de cette méthode, testée sur un cas d'étude réel, WooKey, a donné des résultats qui ont permis de prouver la plupart des contre-mesures et de trouver une erreur dans une autre.

Structure du document. Cette thèse est structurée comme suit :

- Le Chapitre 2 présente l'état de l'art, donne le contexte technique et rappelle certaines notions importantes développées par différents travaux précédents qui seront utiles dans la suite de la thèse ;
- Le Chapitre 3 formalise plusieurs notions indispensables pour représenter nos contributions sur les critères de flot de données. On y introduit un petit langage de programmation, donne quelques définitions notamment relatives au flot de données, puis on termine en proposant des techniques de modélisation et de génération d'objectifs dans un programme ;
- Le Chapitre 4 décrit en détail nos contributions relatives à la définition et détection des objectifs de test polluants en utilisant les notions introduites par les chapitres précédents ;
- Le Chapitre 5 revient sur la génération de tests présentée dans l'état de l'art et étudie comment l'utiliser sur les critères de couverture de code liés au flot de données avec les problèmes que cela peut apporter. On donne également des exemples de nouveaux critères de flot de données ;
- Le Chapitre 6 résume comment nous avons implémenté les contributions précédentes dans trois greffons de Frama-C : LAnnotate, LUncov et PathCrawler. Ces outils sont ensuite utilisés pour réaliser des expérimentations qui sont détaillées, avec une mesure des résultats obtenus sur divers programmes réels et les conclusions que l'on peut en tirer ;

- Le Chapitre 7 est consacré à nos travaux sur l'injection de fautes et plus spécifiquement à la vérification des mesures mises en place contre l'injection de fautes. On y réalise un cas d'étude complet sur le programme d'amorçage (bootloader en anglais) de WooKey, un prototype de périphérique de stockage USB sécurisé développé par l'ANSSI ;
- Le Chapitre 8 conclut cette thèse en résumant nos contributions et en donnant des pistes pour de futurs travaux.

Publications. Certaines contributions de cette thèse ont été publiées :

- Un article de la conférence IFM 2020 [Mar+20] présente le travail effectué sur la détection des objectifs de flot de données polluants (Chapitre 4) ainsi que les résultats obtenus sur des exemples de C réels (Section 6.2 du Chapitre 6).

  Les contributions clés de cet article sont résumées en français dans un résumé étendu publié à AFADL 2021 [Mar+21] ;
- Un article à la conférence SAC 2022 [Mar+22] présente notre méthode de vérification formelle de contre-mesures contre l'injection de fautes. Ces travaux ont été testés sur un cas d'étude de WooKey, un prototype de clé USB sécurisé. Le Chapitre 7 présente ce travail.

En plus des articles cités, ces travaux ont été présentés à :

- ICTSS 2019, avec un poster sur la détection des objectifs de flot de données infaisables, un type d'objectifs polluants particulier ;
- Lors des groupes de travail MTV 2022 du GDR GPL et MFS 2022 du GDR Sécurité sur nos travaux sur l'injection de fautes et la vérification de contre-mesures.

Brevet. Un dépôt de brevet a été effectué conjointement entre Thales et le CEA sur les travaux du Chapitre 7.

Chapitre 2

État de l'art et contexte technique

Sommaire

Dans ce chapitre, nous allons dans tout d'abord faire un tour des différentes techniques de vérification étudiées pendant la thèse (Section 2.1), puis nous présenterons le domaine de l'injection de fautes (Section 2.2). Nous détaillerons ensuite les travaux précédents sur la spécification des critères de couverture et de leur représentation sous forme de labels et d'hyperlabels (Section 2.3), et nous présenterons les outils principaux que nous avons utilisés lors de cette thèse (Section 2.4). Nous terminerons par la description du fonctionnement de PathCrawler sur la génération de tests (Section 2.5) et la méthode utilisée pour guider la génération dans le but de couvrir les labels.

2.1 Validation et vérification de programme

2.1.1 Notions de correction et complétude pour la vérification et le test

Soit $S_1 \subseteq S$. Supposons qu'on cherche à identifier le sous-ensemble S_1 de S, et on possède une technique qui retourne $S' \subseteq S$. On dit que cette technique est correcte si $S' \subseteq S_1$. On dit qu'elle est complète si $S_1 \subseteq S'$.

Nous pouvons appliquer ces notions pour les techniques de vérification et de test. Pour ces problèmes, on va considérer des ensembles de propriétés.

Soient $S = \{(loc_1, \varphi_1), \ldots, (loc_n, \varphi_n)\}$ un ensemble de n propriétés, où loc_i est un point de programme et φ_i un prédicat qui porte sur des variables définies en point loc_i.

On considère que le problème de vérification de propriétés (ou preuve, ou recherche de propriétés valides) consiste à identifier le sous-ensemble de propriétés $S_{val} \subseteq S$ qui sont valides (c'est-à-dire, satisfaites pour toutes les exécutions possibles du programme).

On considère que le problème d'invalidation (ou recherche de propriétés invalides) consiste à identifier le sous-ensemble de propriétés $S_{inval} \subseteq S$ qui sont invalides (c'est-à-dire, propriétés pouvant échouer au moins pour une exécution).

L'analyse statique, vue comme une technique (pour résoudre le problème) de vérification, permet d'identifier un sous-ensemble de propriétés S' pour lesquelles l'analyse permet de prouver leur validité. Dans ce cas, $S' \subseteq S_{val}$. Pour cette raison, l'analyse statique est correcte en tant que technique de vérification : les propriétés identifiées comme valides sont effectivement valides. En

revanche, toutes les propriétés valides ne seront pas forcément prouvées donc la technique n'est pas complète.

Le test, vu comme une technique (pour résoudre le problème) d'invalidation, permet d'identifier un sous-ensemble de propriétés S'' pour lesquelles le test permet de montrer l'invalidité. Dans ce cas, $S'' \subseteq S_{inval}$. Pour cette raison, le test est correct en tant que technique d'invalidation : les propriétés identifiées comme invalides sont effectivement invalides. En revanche, toutes les propriétés invalides ne seront pas forcément trouvées (car le test n'est pas exhaustif en général) donc la technique n'est pas complète. On peut citer la fameuse phrase de Dijkstra "Testing shows the presence, not the absence of bugs" [Bux+70].

Le test pourrait être vu comme une technique (pour résoudre le problème) de vérification. Dans ce cas, on définirait l'ensemble S' retourné par la technique comme l'ensemble de propriétés qui n'ont pas été montrées invalides par le test. Cette technique de vérification n'est pas correcte puisque le test est partiel, donc une propriété invalide peut rester non détectée par le test. En revanche, elle serait complète : elle ne va jamais considérer comme invalide une propriété valide.

Puisque les méthodes formelles sont le plus souvent utilisées pour le problème de vérification (ce qui sera également le cas pour la plupart des contributions dans cette thèse), dans ce manuscrit, nous allons utiliser la définition de correction et complétude d'une analyse en tant qu'une technique de vérification.

2.1.2 Test logiciel

Le test est la technique la plus répandue pour vérifier et valider des logiciels. Le test est rarement automatique (beaucoup de tests sont écrits à la main) et correct (on ne peut pas tout tester), mais requiert moins d'expertise pour être mis en place comparé à d'autres techniques plus formelles que nous présenterons après.

Afin d'évaluer les suites de tests, des métriques ont été mises en place : les *critères de couverture de code*. Il en existe un grand nombre [Rap+82 ; Amm+16], permettant de définir ce que nous appelons des *objectifs de test*. En anglais, Ammann et Offutt utilisent le terme "test requirements" [Amm+08, Def. 1.20], alors que les travaux précédents du CEA List sur les critères de couverture utilisent le terme "test objectives" [Bar+14a ; Mar+17a ; Mar+17b ; Mar+18]. Dans la continuité des travaux précédents, nous utilisons dans ce manuscrit le terme "objectif de test" pour les éléments de programme à couvrir qui dépendent du critère donné (instructions, branches, combinaisons de conditions, paires def-use,

etc.). On doit couvrir les objectifs associés au critère donné lors de l'exécution d'une suite de tests pour considérer que cette dernière est assez représentative, qu'elle couvre suffisamment bien les différents comportements possibles du programme. Les critères les plus connus sont par exemple :

- **DC** (Decision Coverage), pour la couverture des décisions : on souhaite couvrir toutes les branches du programme. Pour chaque décision d, on veut couvrir d et $\neg d$;
- **CC** (Condition Coverage), similaire à **DC**, on souhaite couvrir toutes conditions (expression booléenne atomique sans opérateur booléen) de chaque décision du programme. Par exemple si on a une décision $A \wedge B$, on veut couvrir A, $\neg A$, B et $\neg B$;
- **FC** (Fonction Coverage), comme son nom l'indique, on veut s'assurer d'avoir exécuté au moins une fois chaque fonction du programme ;
- **MCDC** (Modified Condition / Decision Coverage), ou l'on souhaite couvrir à la fois **CC**, **DC**, et en plus vérifier que chaque condition peut avoir un impact sur le résultat de la décision ;
- **All-defs** (All-definitions), un exemple de critère de flot de données, où on va s'intéresser aux couples définitions (modification d'une variable) et utilisations d'une variable (appelées paires def-use), pour vérifier que chaque définition est au moins utilisée une fois lors de l'exécution des tests.
- **WM** (Weak Mutation), où on va tester la couverture de versions mutées d'expressions logiques ou arithmétiques. Plusieurs types de mutation existent, par exemple :
  - **ROR** pour la mutation d'opérateurs relationnels (changer $<$ en $<=$, $>$ et $>=$, $==$ en $\not=$ etc.) ;
  - **COR** pour la mutation d'opérateurs booléens (changer $\wedge$ en $\vee$ et réciproquement) ;
  - **AOR** pour la mutation d'opérateurs arithmétiques (changer $+$ en $-$, $*$ et $/$, etc.).

On distingue donc les critères de couverture logiques [Amm+03] (comme **CC** ou **DC**) qui se basent principalement sur les conditions et branches du programme, les critères de flot de données (avec les paires def-use, comme **All-defs**) ou encore les mutations comme **ROR** ou **AOR**. Plusieurs études ont été réalisées dans le but de mesurer l'efficacité des critères [Hem15], ou de les comparer [Fra+93 ; Fra+97 ; And+06].

Les critères de couverture de code étant très hétérogènes, il existe un très grand nombre d'outils [Yan+09] qui en supportent un ensemble plus ou moins

restreint. En 1994, l'outil ATW [Chi+94] (Ada Testing Workbench) était introduit comme un outil générique supportant 21 critères de couvertures de code pour le langage Ada. Il ne semble cependant pas disponible aujourd'hui. Plus modeste, Bullseye [Bul09] permet de mesurer la couverture de code de programmes écrits en C++ selon les critères de couverture de branches **DC** et de conditions **CC**. GCov [Gco] permet quant à lui de mesurer, en plus de la couverture de branches, la couverture de blocs **BBC** et de fonctions **FC** sur les programmes C. LDRACover [Ldr] supporte plusieurs langages (C, C++, Java et Ada95) et offre la possibilité de mesurer la couverture des fonctions, des instructions, des appels de fonction, mais également de **MCDC** [Chi+01], un critère très utilisé dans le test d'applications critiques, parce qu'imposé par les normes. Enfin, Testwell CTC++ [Tes], sur les programmes C, C++ et Java, permet de mesurer la couverture d'une dizaine de critères parmi ceux cités ci-dessus, avec **DC**, **CC**, **MCDC**, **MCC**, **FC**, etc.

Cependant, certains objectifs sont parfois impossibles à réaliser, c'est-à-dire qu'on ne peut pas trouver de cas de test qui couvre cet objectif. Les critères de flot de données en particulier sont souvent concernés, à cause de leur nature complexe sur des chemins d'exécution. Ces problématiques ont été abordées fréquemment dans la littérature [Rap+85 ; Fra+88 ; Din+12]. De manière générale, déterminer la faisabilité des chemins dans un programme est un problème beaucoup étudié. Par exemple, [Alt96] combine une analyse structurelle (énumération de chemins et analyse de flot de contrôle) et une analyse d'exécutabilité (avec de l'exécution symbolique) pour trouver des chemins infaisables. Dans [Che+05], les auteurs essaient de trouver des chemins infaisables en identifiant des conflits entre les définitions de variables et les conditions de branches. On peut également citer [Su+15] qui utilise l'exécution symbolique dynamique et le model checking pour identifier certaines paires polluantes.

Un point important de cette thèse est l'aspect générique de nos approches. On voit que plusieurs outils existent selon les critères que l'on souhaite vérifier et avoir un outil regroupant un très grand nombre de critères est donc pertinent. Nous utiliserons pour ça LTest [Bar+14a ; Mar+17a], un ensemble d'outils permettant de réaliser les services de test (création d'objectifs pour un critère de couverture donné, génération de tests, détection d'objectifs polluants et mesure de couverture) de manière générique (voir Section 2.3).

2.1.3 Vérification déductive

La *vérification déductive* est une technique dont les bases ont été introduites dans les années 1960-1970. Floyd et Hoare [Flo67 ; Hoa69] ont introduit la notion de triplet de la forme $\{P\}S\{Q\}$, où S est un programme et P et Q représentent

respectivement la précondition et la postcondition. Ce triplet est appelé *valide* si pour tout état initial σ qui vérifie P avant l'exécution de S et que S termine dans un état σ', alors σ' vérifie Q. Des règles sont ensuite introduites afin de déduire la validité des triplets. Par exemple, la séquence de programmes est valide si elle respecte la règle suivante :

$$\frac{\{P\}S_1\{R\} \qquad \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

Ici le triplet $\{P\}S_1; S_2\{Q\}$ est valide si l'exécution de S_1 dans un état satisfaisant P nous amène dans un nouvel état satisfaisant R, puis la même chose pour S_2 avec R et Q. L'ensemble de ces règles, connu sous le nom de logique de Hoare ou de Floyd-Hoare, permet de vérifier qu'un triplet de Hoare est valide. Dijkstra [Dij76] introduit en plus le principe de *plus faible précondition*, dont l'objectif est de trouver les préconditions minimales requises pour satisfaire une postcondition. L'avantage de cette technique est qu'il n'est plus nécessaire de trouver manuellement les prédicats intermédiaires. Ils peuvent être calculés par des prouveurs automatiques. Dans l'exemple précédent, l'état R est calculé à partir de S_2 et de la postcondition Q, et ainsi de suite. Cette technique permet donc de vérifier un programme à l'aide de pré- et postconditions globales de manière automatique. On crée ce qu'on appelle des *contrats de fonctions* contenant ces pré- et postconditions et les prouveurs automatiques vérifient qu'elles sont valides. En pratique, certaines constructions comme les boucles nécessitent également des annotations supplémentaires pour aider les prouveurs automatiques, car elles sont compliquées à analyser. Enfin, les pointeurs représentent un challenge important quand on fait de la vérification déductive et il existe plusieurs modèles mémoires [Bor00] pour représenter les pointeurs dans la logique.

De nombreux outils existent pour faire du calcul de plus faible précondition. ESC/Java [Fla+02] est un outil permettant de détecter des erreurs dans des programmes écrits en Java. Sa particularité est qu'il est ni correct, ni complet, car les auteurs ont préféré se concentrer sur la rentabilité de l'outil, et donc de détecter assez d'erreurs pour compenser le coût de l'utilisation de ce dernier. Peu après, c'est KeY [Bec+07], un autre outil de vérification de programme java, qui voyait le jour. À la même période, le langage de programmation SPARK [Bar03] était créé. Basé sur le langage Ada, il permet d'écrire des programmes en Ada avec des contrats de fonctions et de boucles et d'utiliser ensuite un prouveur pour les vérifier. De manière similaire, Why3 [Com+96] est un outil permettant de faire de la preuve de programme via un langage de spécification appelé WhyML, proche du langage OCaml. Enfin, on peut également citer Caveat [Ran+99], un outil de vérification de programme C utilisé notamment par Airbus et son successeur Wp [Bau+22], un greffon de Frama-C [Kir+15].

Dans cette thèse, nous allons utiliser Wp comme outil de vérification déductive.

2.1.4 Analyse de flot de données

L'*analyse de flot de données* est une technique introduite dans les années 1970 [Her76 ; All+76] visant à améliorer la fiabilité des logiciels [Fos+76]. Proche de l'analyse de flot de contrôle [All70], qui a pour objectif d'observer le comportement du graphe de flot de contrôle d'un programme, l'analyse de flot de données consiste à propager de l'information sous forme d'états à travers le graphe de flot de contrôle, avec des opérations de transitions d'un nœud à l'autre et de jointure (fusion de deux états) pour vérifier comment l'information se propage. Une utilisation courante de cette analyse est d'étudier les relations entre définitions et utilisations de variables du programme qu'on appelle des *paires def-use*. On va utiliser le graphe de flot de contrôle du programme et le parcourir en propageant par exemple les définitions de variables rencontrées sur le chemin. En atteignant une utilisation de variable, on peut alors utiliser les informations recueillies pour trouver les définitions de cette variable qui précède cette utilisation dans le graphe, et ainsi identifier ses paires def-use dans le code.

Beaucoup de travaux ont été réalisés autour de cette technique dans le cadre du test de programme [Su+17]. Ces travaux sont globalement rassemblés dans la catégorie de test DFT (Dataflow testing), qui comporte des critères de couverture spécialisés [Rap+82 ; Amm+16]. Ces critères permettent de tester des propriétés relatives aux paires def-use. On peut citer par exemple le critère **All-defs**, pour lequel on veut s'assurer que chaque définition est utilisée au moins une fois durant les tests. De manière similaire, le critère **All-uses** demande à ce que chaque définition d'une variable soit utilisée par toutes les utilisations de cette variable dans le programme. On doit alors exécuter chacun des chemins allant d'un nœud définissant une variable vers un des nœuds qui l'utilisent. Des travaux ont également été effectués sur les paires def-use interprocédurales [Har+89], concernant les variables globales. Enfin, tous ces critères ont été évalués et comparés [Cla+89 ; Wey90]. Des travaux plus récents portent également sur la création de nouveaux critères de flot de données [Kol+21 ; Kol+22] plus complexes.

Dans cette thèse, nous utiliserons une analyse de flot de données implémentée dans de Frama-C (voir la Section 2.4).

2.1.5 Interprétation abstraite

L'*interprétation abstraite* a été introduite par Cousot et Cousot [Cou+77] dans la fin des années 1970. Elle représente un cas particulier d'analyse de flot de données dans laquelle on propage des *états abstraits*, représentant des ensembles d'états concrets, le long du graphe de flot de contrôle. Ces états abstraits vont sur-approximer l'ensemble des états concrets en un point du programme. Si on peut arriver en un point du programme avec un état concret σ, alors σ est inclus dans l'ensemble représenté par l'état abstrait associé à ce point de programme. Par exemple, si notre état abstrait associe à chaque variable (entière) du programme un intervalle de valeurs possibles, et qu'une variable v est associée à $[0; 3]$, alors cette abstraction représente tout état concret où v est associée à une valeur entière entre 0 et 3. Mais on a parlé de sur-approximation, et ici il est par exemple possible que v ne puisse jamais valoir 2, même si 2 appartient à l'intervalle. Un état concret peut donc appartenir à un état abstrait sans pour autant être obtenable pendant l'exécution concrète. En revanche, on souhaite que l'analyse soit correcte, c'est-à-dire qu'on n'oublie pas d'état concret dans notre abstraction, d'où la sur-approximation.

Les fonctions de transitions d'un nœud à l'autre dans l'interprétation abstraite sont croissantes : les états abstraits ne peuvent que s'élargir, c'est-à-dire contenir de plus en plus d'états concrets pendant l'analyse. Cette propriété nous garantit alors que l'analyse trouvera un *point fixe*, qui, si les fonctions sont correctes, est une sur-approximation de l'ensemble des états concrets qui peuvent arriver en chaque point de programme. Cependant, avec une jointure normale, on ne garantit pas encore la terminaison. On peut imaginer, par exemple, une boucle infinie incrémentant une variable entière v, initialisée à 0 avant la boucle. Dans ce cas, elle vaudra 0 au début, puis après une itération, on joint l'état 0 avec l'état 1 et on obtient l'intervalle $[0; 1]$, et ainsi de suite. Après n itérations on aura l'état $[0; n]$ et notre intervalle converge alors lentement vers $[0; +\infty]$. Pour éviter ce genre de cas, et accélérer la convergence de l'analyse, on utilise l'*opérateur d'élargissement* (widening). On note $x \triangledown y$ l'opération d'élargissement, où x représente l'état abstrait déjà calculé et y ce qu'on est en train de propager. Son rôle est, lors d'une jointure, de garder un état abstrait strictement plus grand que l'état qu'on aurait obtenu par une jointure normale. Par exemple, si on fait la jointure, avec un opérateur d'élargissement simple sur le domaine abstrait des intervalles, de deux états abstraits $[u_1; u_2] \triangledown [u'_1; u'_2]$ on obtient les résultats suivants :

- si $u'_1 < u_1$, on remplace la borne inférieure du résultat par $-\infty$, sinon on garde u_1 ;
- si $u_2 < u'_2$, on remplace la borne supérieure du résultat par $+\infty$, sinon on

garde u_2.

Au pire des cas, on atteindra $[-\infty; +\infty]$ en trois itérations. Dans le cas de la boucle infinie de l'exemple précédent, on effectuera l'opération $[0; 0]\nabla[0; 1]$ qui donnera comme résultat $[0; +\infty]$, et à l'itération suivante on vérifie que notre état ne bouge plus (qu'on a atteint un point fixe) et donc qu'on a atteint notre résultat.

On peut imaginer des tas d'opérateurs d'élargissements différents sur un domaine abstrait. De plus, l'utilisation d'intervalles est un exemple de domaine abstrait pouvant être utilisé pour effectuer l'analyse, mais il en existe d'autres plus ou moins complexes. Il s'agit souvent de choisir et faire des compromis entre précision de l'analyse et temps de calcul.

L'analyse par interprétation abstraite est disponible dans plusieurs outils. Polyspace [Pol99] est un des premiers outils, développés après l'accident d'Ariane 501 [Lio+96], pour la vérification de code C, C++ et Ada. On peut également citer Astrée [Cou+05] pour le C et C++, notamment utilisé par Airbus, ou encore Eva [Bla+17] pour le C, un greffon de la plateforme Frama-C. Ces outils ont été également tous les trois utilisés par Électricité de France (EDF) [Our15] pour évaluer la sûreté de logiciels utilisés dans les centrales nucléaires. Eva est également utilisé par l'Agence nationale de sécurité des systèmes d'information (ANSSI) pour son parseur X.509 [Eba+19]. Enfin, Facebook a développé Infer [inf], un outil d'analyse statique utilisant, entre autre, des techniques d'interprétation abstraite [Dis+06] pour faire de l'analyse d'alias.

Dans cette thèse, nous utiliserons Eva comme outil d'analyse statique par interprétation abstraite pour la détection d'objectifs polluants (Chapitre 6).

2.1.6 Exécution symbolique

L'*exécution symbolique* [Kin76] est une technique permettant d'utiliser des valeurs symboliques (au lieu des valeurs concrètes) dans l'objectif d'explorer les chemins d'un programme, de créer des contraintes (*prédicat de chemin*) pour activer chacun des chemins d'exécution possibles à parcourir, et d'utiliser un solveur de contraintes pour générer ensuite des tests satisfaisant ces contraintes. Cette méthode est donc très utile pour générer des suites de tests avec une grande couverture de code du programme. Plusieurs outils ont été mis en place, comme EFFIGY [Cla76], SELECT [Boy+75] sur des programmes écrits en un sous-ensemble de LISP, ou encore DISSECT [How77] pour analyser des programmes en Fortran.

Plus récemment, des techniques ont été mises en place pour combiner l'exécution symbolique et l'exécution concrète d'un programme : l'exécution sym-

bolique dynamique (DSE, aussi appelée *test concolique*) [Sen07 ; Maj+07]. L'idée est d'instrumenter le programme pour observer le chemin pris pendant l'exécution concrète et, à partir de cette observation, guider l'exécution symbolique pour générer des contraintes activant d'autres chemins, et ainsi de suite. On peut ici citer les outils CUTE et jCUTE [Sen+05 ; Sen+06] pour les programmes C et Java, ou encore DART [God+05] et KLEE [Cad+08] également sur les programmes écrits en C. Enfin, Microsoft utilise l'outil SAGE [God+12] pour trouver des bugs et des failles de sécurité dans ses applications.

L'exécution symbolique peut être vue à la fois comme une technique statique, car l'exécution concrète du programme n'est pas indispensable, ou bien, combinée au test, comme une technique dynamique. Si l'exécution symbolique arrive à explorer tous les chemins d'un programme en montrant l'impossibilité de violer une propriété donnée, cette exploration fournit une preuve de correction pour cette propriété. En pratique, pour les programmes réels, le nombre de chemins peut être trop grand (voire infini) et les solveurs peuvent ne pas réussir à résoudre les contraintes nécessaires dans un délai attendu, ce qui explique pourquoi cette technique (ou sa variante DSE) est souvent utilisée pour le test avec une exploration partielle des chemins. Dans ce dernier cas, on réduit les chemins explorés, par exemple, par une longueur maximale ou un nombre maximal d'itérations de boucles.

Dans cette thèse, nous utiliserons l'exécution symbolique dynamique via l'outil PathCrawler que nous présenterons en détail dans la Section 2.5.

2.2 Injection de fautes

L'*injection de fautes* est un ensemble de techniques consistant à attaquer un système afin d'altérer son comportement au niveau matériel ou logiciel. Il existe beaucoup de techniques et d'outils différents [Hsu+97 ; Zia+04], ayant chacune leurs avantages et leurs inconvénients.

L'injection de fautes basée sur le matériel (Hardware-based fault injection) est la technique à laquelle nous nous sommes intéressés dans une partie de cette thèse. Elle nécessite l'accès physique au matériel du système que l'on souhaite attaquer et consiste à provoquer une faute dans le but d'altérer l'exécution normale du système visé [Arl+90 ; Kar+94 ; Bar+12 ; Bha+14]. Les systèmes ciblés par ce type d'attaques sont principalement les systèmes embarqués. Ces attaques peuvent être réalisées par différents moyens, comme des impulsions laser, des radiations d'ions lourds, des perturbations de l'alimentation ou de l'horloge, etc. Un des objectifs recherchés par ce type d'attaques est de contourner certaines vérifications critiques dans le code (comme une authentification

utilisateur, une vérification d'intégrité ou de versions, etc.) pour atteindre une zone protégée dans le code qui donne accès à des informations sensibles ou des ressources matérielles. Cette technique rend possible l'accès à des endroits inaccessibles via les autres méthodes, mais nécessite, par contre, une installation avec du matériel précis, parfois coûteuse, afin de réaliser les expérimentations. Par exemple, l'outil VoltPillager [Che+21], qui va contrôler la tension électrique du CPU en injectant des messages dans le SVID (Serial Voltage Identification), interface permettant au CPU d'annoncer la tension actuellement requise par ce dernier. VoltPillager est utilisé dans l'objectif de corrompre l'exécution du micro-logiciel Intel SGX, servant à sécuriser des données en cours d'utilisation, dans le but de récupérer les clés d'algorithmes cryptographiques.

L'injection de fautes basée sur le logiciel (Software-based fault injection) consiste à altérer le fonctionnement d'un logiciel directement, pendant la compilation ou pendant son exécution. On a pour cela besoin de modifier le logiciel, à l'aide d'un outil que l'on vient brancher entre le logiciel cible et le système d'exploitation, directement dans le logiciel cible ou directement dans le système d'exploitation, si c'est ce dernier que l'on souhaite altérer. Contrairement aux attaques matérielles qui nécessitent un accès physique, ici les attaques peuvent être effectuées à distance. On peut citer par exemple l'outil Plundervolt [Mur+20]. Cet outil a un objectif similaire à VoltPillager, mais, cette fois-ci, l'attaque n'est pas matérielle, mais bien logicielle. Au lieu d'avoir tout un ensemble matériel d'outils à brancher sur le système, ici on se place directement dans le logiciel. Et il existe beaucoup d'autres outils comme Xception [Car+98], EXFI [Ben+98] ou FERRARI [Kan+95] utilisant ce type d'injection de fautes logicielle afin de tester ou exploiter des systèmes critiques.

Il est également possible de simuler un système en créant des modèles et d'injecter ensuite des fautes dans ce modèle afin d'observer son comportement (Simulation-based fault injection). Cette technique a l'avantage de ne pas être intrusive contrairement aux techniques précédentes et permet de réaliser un très grand nombre de tests, en modélisant beaucoup de types de fautes et de méthodes d'injections possibles. En revanche, cette technique nécessite beaucoup de développement, les expériences sont potentiellement longues et les modèles doivent être très proches du système simulé pour que les résultats soient précis. L'outil MEFISTO [Jen+94] et d'autres variantes comme MEFISTO-C [Fol+98] utilisent le langage de modélisation VHDL [Rim+93] pour faire ces simulations de fautes.

Pour se prémunir des attaques, les concepteurs de logiciels vont ajouter ce que l'on appelle des *contre-mesures* dans le logiciel. Ces contre-mesures dépendent du type de fautes et de techniques d'injection que l'on souhaite éviter,

et il en existe donc plusieurs types. Les contre-mesures à base de redondance de code sont sans doute les plus simples à mettre en place pour se prémunir de l'injection de fautes matérielles. On va, dans le logiciel, dupliquer certains morceaux du code considérés comme critiques afin d'augmenter les chances que ce morceau soit exécuté au moins une fois normalement (c'est-à-dire sans faute). Ces contre-mesures ne sont pas triviales à vérifier, mais pourtant cruciales pour assurer une résistance contre le modèle de fautes considéré. Plusieurs approches sont utilisées pour évaluer l'efficacité des contre-mesures d'un système donné.

La méthode la plus naïve consiste simplement à injecter des fautes dans le système afin de permettre aux ingénieurs de validation de détecter des vulnérabilités ou d'être confiants sur le fait que le système est suffisamment résistant aux attaques. On peut alors utiliser les techniques basées sur le logiciel ou le matériel pour reproduire des attaques potentielles sur le périphérique ciblé. Ce type de techniques a l'avantage de prendre en compte le système cible réel, mais ne peut pas garantir que le système pourra résister à des attaques similaires en situation réelle, même légèrement différente (par exemple une différence de force du signal, de la fréquence, de la durée ou lu nombre d'essais).

Une autre approche consiste à chercher des attaques potentielles au niveau du logiciel, en simulant un ensemble de fautes possibles dans le code et en essayant d'identifier des attaques grâce à la génération de tests comme l'outil Lazart [Pot+14] ou de prouver leur absence en utilisant les méthodes formelles [Chr+13 ; Hey+19 ; Lac+21].

Même si leurs résultats font plusieurs hypothèses (le modèle de fautes considéré, la méthode de simulation, le compilateur, etc.), les approches au niveau logiciel fournissent un complément utile à l'évaluation physique des systèmes : elles sont moins chères, peuvent être totalement automatiques et peuvent rigoureusement prendre en compte toutes les fautes possibles par rapport à la simulation de fautes choisies.

Les travaux effectués pendant cette thèse poursuivent les efforts précédents [Pot+14 ; Chr+13 ; Hey+19 ; Lac+21] dans cette direction. On considère un modèle de faute simple qui permet à un attaquant d'inverser n'importe quel sous-ensemble d'au plus k instructions conditionnelles dans le code. Ce modèle, appelé *l'inversion de test*, est vu comme un modèle de faute très utile dans un rapport conjoint des autorités françaises de certification et d'évaluation [ANS+20, Sec. 16.4].

2.3 Formalisation des critères de couverture avec les Labels et Hyperlabels

Afin de s'assurer qu'on teste un système dans un ensemble de configurations suffisamment variées, plusieurs *critères de couverture* (ou *critères de test*) ont été mis en place dans le but de guider et simplifier le test tout en améliorant son efficacité. Les critères de couverture permettent de spécifier de manière abstraite un ensemble de points structurels ou de situations que l'on souhaite exécuter lors des tests. Ces critères peuvent être concrétisés, par exemple, directement intégrés dans le code, pour créer des objectifs de test. Ces objectifs pourront ensuite être utilisés par d'autres techniques comme la génération de tests, afin de générer une suite respectant au mieux possible ces critères. Dans cette section, nous allons présenter une manière de représenter les critères de couverture de code avec les labels et les hyperlabels. Les exemples seront en C, mais restent valides pour la plupart des langages de programmation impératifs.

Les labels et hyperlabels ont été introduits dans [Bar+14b ; Mar+17a]. Ils représentent une manière simple et formelle de décrire les critères de couverture de code.

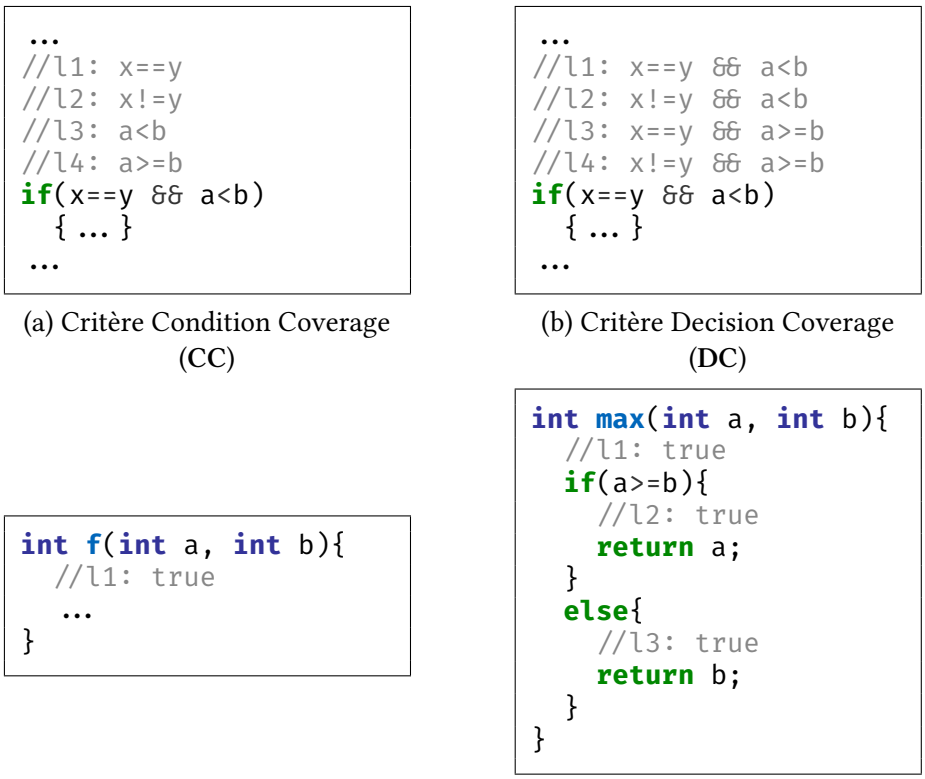
2.3.1 Labels

Les labels sont la forme la plus basique des objectifs de test, mais permettent déjà de représenter la majorité des critères de couverture de code les plus courants.

Définition 1 (Labels). Étant donné un programme P, un *label* ℓ est une paire (loc,φ) où loc est une position dans P et φ un prédicat.

La notion de couverture d'un label est simple : si on atteint loc pendant une exécution du programme dans un état où φ s'évalue à *vrai*, alors ce label sera considéré comme *couvert*. La Figure 2.1 donne plusieurs exemples de critères sous forme de labels.

Exemple 1. Dans l'exemple de la Figure 2.1, chaque label est nommé avec un identifiant unique et est à une position précise dans le code correspondant à sa loc. Le prédicat qui suit correspond donc à φ dans notre définition des labels. La Figure 2.1a illustre le critère de couverture de conditions CC (Condition Coverage), pour lequel on souhaite couvrir chaque condi-



(a) Critère Condition Coverage (**CC**)

(b) Critère Decision Coverage (**DC**)

(c) Critère Function Coverage (**FC**)

(d) Critère Statement Coverage (**STMT**)

FIGURE 2.1 : Encodage d'objectifs de test standards avec des labels

tion atomique de la décision et sa négation. Le critère de couverture de décisions (appelée aussi couverture de branches) **DC** (Decision Coverage, ou Branch Coverage) de la Figure 2.1b est assez proche : ici on veut obtenir chaque combinaison possible d'évaluation des conditions atomiques. La couverture de fonctions **FC** (Function Coverage) (Figure 2.1c) est très simple : on veut s'assurer que toutes les fonctions sont exécutées au moins une fois lors des tests. Pour cela, il suffit d'ajouter un label en début de fonction, avec un prédicat toujours vrai : ici on ne s'intéresse en fait qu'à l'atteignabilité du label. Enfin, le critère de couverture d'instructions **STMT** (Statement Coverage) (Figure 2.1d), comme son nom l'indique, a pour objectif de vérifier si on atteint bien toutes les instructions lors des tests. De la même manière que pour **FC**, les prédicats sont ici toujours vrais, on ajoute un label devant chaque instruction.

Remarque 1. Puisque le mot *label* est utilisé dans cette thèse à la fois pour un objectif de test et (plus rarement) pour un label d'instruction de programme C vers lequel on peut faire un saut, nous dirons toujours "label C" pour distinguer le second cas.

Comme nous l'avons vu, ces critères permettent d'exprimer beaucoup d'objectifs de tests, plus précisément, tous les objectifs qui peuvent s'exprimer sous la forme d'un prédicat en un seul point du programme. Cependant, ils sont insuffisants pour exprimer des objectifs plus complexes. On aimerait pouvoir exprimer des relations entre labels, voire des objectifs qui nécessitent de comparer les résultats de plusieurs exécutions, et créer des objectifs qui ne se réduisent pas à évaluer un prédicat en un seul point du programme. C'est pour cela qu'ont été introduits les hyperlabels.

2.3.2 Hyperlabels

Les hyperlabels ont été introduits dans [Mar+17a] avec le langage HTOL (Hyperlabel Test Objective Language), un langage formel pour les spécifier. Ce langage ajoute cinq nouvelles constructions, les *liaisons* (ou bindings), les *séquences*, les *gardes*, les *conjonctions* et les *disjonctions*.

Définition 2 (Hyperlabels). Un *hyperlabel* est un objectif de test défini à partir de labels avec les constructions suivantes :

- Les liaisons $\ell \triangleright \{v_1 \longmapsto e_1; \ldots\}$ stockent dans des *méta-variables* $v_1, \ldots$ les valeurs qu'avaient les expressions $e_1, \ldots$ à chaque fois que le label ℓ est couvert ;
- Les séquences $\ell_1 \xrightarrow{\varphi} \ell_2$ représentent un objectif dans lequel on souhaite avoir un cas de test pour lequel les deux labels sont couverts séquentiellement, au sens où dans la trace d'exécution on trouve un appel de fonction où les deux labels sont atteints (dans l'ordre de la séquence), et le prédicat φ est vérifié sur tout le chemin entre les deux labels ;
- Une conjonction $h_1 \cdot h_2$ crée un objectif pour lequel il faut couvrir h_1 et h_2. Ces deux hyperlabels peuvent être couverts par des tests différents ;
- Une disjonction $h_1 + h_2$, de la même manière que pour les conjonctions, crée un objectif dans lequel il faut couvrir au moins un hyper-

```
int f(){
  if( ... ){g();}
  if( ... ){g();}
}
```

(a) Code de base

```
int f(){
  if( ... ){/*l1: true*/ g();}
  if( ... ){/*l2: true*/ g();}
}
```

(b) Code avec labels

```
// Hyperlabels
// id, hyperlabel, coverage
h1, <l1+l2|>
```

(c) Hyperlabels

FIGURE 2.2 : Création des objectifs de test pour le critère **FCC**

label parmi ceux de la disjonction ;

— Une garde $\langle h \mid \psi \rangle$ permet d'exprimer une contrainte ψ sur les meta-variables utilisées par les liaisons dans h. La garde est couverte si h est couvert avec des exécutions où les méta-variables prennent des valeurs telles que ψ est satisfaite.

Ces constructions peuvent être combinées entre elles pour former des objectifs de test complexes dont nous allons maintenant donner quelques exemples.

Remarque 2. Les hyperlabels permettent de définir des objectifs sur plusieurs traces d'exécutions par l'intermédiaire des liaisons et peuvent donc exprimer des hyperpropriétés sur des ensembles de tests.

Exemple 2 (Critère FCC). Le critère **FCC** (Function Call Coverage), ou couverture d'appels de fonctions, est probablement un des critères les plus simples pour illustrer les hyperlabels. Pour ce critère, on s'intéresse au fait de couvrir au moins un appel de chaque fonction présente dans la fonction appelante. On aura ici besoin uniquement des disjonctions.

La Figure 2.2 montre un exemple dans lequel la fonction `g` est appelée deux fois par la fonction `f`, avec les deux labels et un hyperlabel correspondant au critère **FCC**.

Pour cet exemple, on souhaite couvrir l'hyperlabel h_1, c'est-à-dire au

```
...
if(x==y && a<b)
  { ... };
...
```

(a) Code de base

```
//l1:   x==y && a<b , c1A ← x==y, c2A ← a<b
//l2: !(x==y && a<b), c1B ← x==y, c2B ← a<b
...
if(x==y && a<b)
  { ... };
...
```

(b) Labels et liaisons

```
// Hyperlabels
// id, hyperlabel, coverage
h1, < l1.l2 | c1A != c1B && c2A == c2B>
h2, < l1.l2 | c1A == c1B && c2A != c2B>
```

(c) Hyperlabels

FIGURE 2.3 : Création des objectifs de test pour le critère **MCDC** fort

moins un des deux labels.

$$\ell_1 \triangleq (loc_1, \texttt{true})$$
$$\ell_2 \triangleq (loc_2, \texttt{true})$$
$$h_1 \triangleq \ell_1 + \ell_2$$

Exemple 3 (Critère MCDC). Le critère **MCDC** (Modified Condition / Decision Coverage) a pour objectif de montrer que chaque condition atomique dans une décision peut influencer seule la décision globale. Autrement dit, on veut trouver deux tests où la valeur de vérité de cette condition atomique change et fait changer la décision, alors que la valeur des autres conditions atomiques ne change pas.

Considérons l'exemple de la Figure 2.3. Ici, **MCDC** a pour objectif de montrer que chacune des deux conditions atomiques $c_1 \triangleq$ `x==y` et $c_2 \triangleq$ `a<b` peut influencer seule la décision globale $d \triangleq c_1 \wedge c_2$. Pour chacune des conditions, cela revient à essayer de couvrir la décision d et sa négation en changeant uniquement la valeur de la condition atomique

c_1 (respectivement c_2) sans changer le reste.

On peut exprimer ce critère avec deux hyperlabels h_1 et h_2 utilisant des labels, des liaisons, des gardes et des conjonctions de la façon suivante :

$$\begin{aligned}
\ell_1 &\triangleq (loc_1, d) \triangleright \{c_1 \longmapsto \texttt{x==y;}\, c_2 \longmapsto \texttt{a<b;}\} \\
\ell_2 &\triangleq (loc_1, \neg d) \triangleright \{c_1' \longmapsto \texttt{x==y;}\, c_2' \longmapsto \texttt{a<b;}\} \\
h_1 &\triangleq \langle \ell_1 \cdot \ell_2 \mid c_1 \neq c_1' \wedge c_2 = c_2' \rangle \\
h_2 &\triangleq \langle \ell_1 \cdot \ell_2 \mid c_1 = c_1' \wedge c_2 \neq c_2' \rangle
\end{aligned}$$

La Figure 2.3 illustre cet exemple. On a deux labels ℓ_1 et ℓ_2 qui testent les deux branches et qui contiennent également des méta-variables comme présentées dans les liaisons. Chacune d'elles recevra la valeur de la condition atomique correspondante lors de l'exécution. Enfin, on crée nos hyperlabels h_1 et h_2. Pour chaque hyperlabel, on veut couvrir ℓ_1 et ℓ_2 (conjonction) en ayant des valeurs pour nos conditions atomiques qui respectent les différentes gardes comme présenté ci-dessus. Si ces deux hyperlabels sont couverts grâce à une ou plusieurs traces d'exécution, alors c'est que le critère **MCDC** fort est couvert pour cette décision.

Exemple 4 (Critère All-uses). Le critère **All-uses** Coverage, couverture de toutes les utilisations, a pour objectif, pour chaque définition d'une variable, de couvrir toutes les utilisations de cette variable. On va pour cela utiliser les séquences pour représenter ce qu'on appelle des paires définition-utilisation (def-use).

```
int a = 42;
//l1 : true
if( ... ) {
  //l2 : true
  return a;
}
else {
  //l3 : true
  return a + 1;
}
```

Dans cet exemple, on a une variable a avec une définition qui est utilisée deux fois. On peut alors formuler le critère **All-uses** de la façon sui-

vante :

$$\begin{aligned}
\ell_1 &\triangleq (loc_1, \texttt{true}) \\
\ell_2 &\triangleq (loc_2, \texttt{true}) \\
\ell_3 &\triangleq (loc_3, \texttt{true}) \\
h_1 &\triangleq \ell_1 \xrightarrow{\varphi} \ell_2 \\
h_2 &\triangleq \ell_1 \xrightarrow{\varphi} \ell_3 \\
h_3 &\triangleq h_1 \cdot h_2
\end{aligned}$$

On retrouve nos trois labels, qui sont utilisés pour former deux séquences (une par paire def-use) qui ont toutes les deux comme contrainte que la variable a ne doit pas être redéfinie entre les deux labels. On appelle φ cette contrainte. Enfin, un dernier hyperlabel h_3 nous demande ensuite de couvrir ces deux séquences à l'aide d'une conjonction pour réaliser l'objectif du critère **All-uses**.

Exemple 5 (Critère All-defs). Le critère **All-defs** Coverage, couverture de toutes les définitions, est très similaire à **All-uses**. Cette fois-ci, au lieu de s'intéresser à toutes les utilisations, on cherche à en couvrir au moins une, c'est-à-dire qu'on veut utiliser chaque définition au moins une fois lors de l'exécution des tests.

Si on reprend l'exemple utilisé pour **All-uses**, avec 3 labels, on obtient l'hyperlabel suivant :

$$h_3 \triangleq h_1 + h_2$$

Les autres labels et hyperlabels sont identiques à **All-uses**, mais h_3 est cette fois-ci une disjonction au lieu d'une conjonction. On aura alors besoin de couvrir h_1 ou h_2 pour satisfaire le critère **All-defs**.

Le langage HTOL nous permet donc d'exprimer des critères de couverture complexes, sur une ou plusieurs traces d'exécution. Cependant, cette complexité nécessite des techniques plus avancées pour réaliser d'autres services de test comme la génération de tests ou la détection d'objectifs polluants. Nous verrons dans les chapitres suivants les travaux que nous avons menés pour gérer certains de ces critères.

2.4 Frama-C, une plateforme de vérification de programmes C

Dans le cadre de cette thèse, nous avons basé l'implémentation de nos travaux sur Frama-C [Kir+15 ; Bau+21a], une plateforme développée au CEA List en OCaml permettant d'analyser du code C.

2.4.1 Noyau de la plateforme

Le noyau de Frama-C (kernel en anglais) est la partie centrale de la plateforme et prend en charge plusieurs services pour l'utilisateur.

Parseur et arbre de syntaxe abstraite. Frama-C permet de parser du code C et par la même occasion de vérifier qu'il est bien formé. Le programme est ensuite normalisé par la plateforme et traduit en CIL [Nec+02], un langage intermédiaire, sous forme d'arbre de syntaxe abstraite (AST en anglais). Cette normalisation implique principalement :

- l'unification des retours de la fonction, on ne garde qu'une seule instruction de retour et on remplace les anciens retours par des `goto`. Par exemple `if(x >= y) return x; else return y;` deviendra

```
int res;
if(x >= y) {res = x; goto ret_lbl;}
else {res = y; goto ret_lbl;}
ret_lbl : return res;
```

- la suppression des effets de bord potentiels dans les expressions. Par exemple `if(f(x))...` deviendra `int tmp = f(x); if(tmp)...` pour éviter le cas où `f` comporte des effets de bord ;
- la transformation de toutes les instructions de boucles en `while(1)`. Par exemple la boucle `for(i = 0; i < 10; i++)...` sera transformée en `i = 0; while(1){if(!(i < 10)) break; ... ;i++};`
- la suppression des opérateurs logiques, où on transforme par exemple `if(A && B)...` en `if(A){if(B)...}`.

D'autres transformations sont effectuées, mais on ne va pas toutes les détailler ici, celles-ci sont les principales ayant un impact potentiel sur nos techniques, notamment les critères de test. Certaines sont désactivables, comme la suppression des opérateurs booléens, ce qui nous arrange pour les critères logiques comme **DC** ou **MCDC**. Une fois le code normalisé, on peut explorer cet AST, à l'aide de visiteurs, pour effectuer des manipulations et transformations de code, ajouter des annotations ACSL (un langage de spécification) ou encore réaliser des analyses.

```c
int find_min(int* t, int size) {
  int min, i_min;
  i_min = 0;
  min = t[0];
  for (int i = 1; i < size; i++) {
    if (t[i] < min) {
      i_min = i;
      min = t[i];
    }
  }
  return i_min;
}
```

(a) Exemple de recherche du minimum d'un tableau en C

```c
/*@ requires size > 0;
    requires \valid(t + (0 .. size - 1));
    ensures 0 <= \result < \old(size);
    ensures \forall integer j; 0 <= j < \old(size)==>
      *(\old(t) + \result) <= *(\old(t) + j);
    assigns \nothing; */
int find_min(int *t, int size){
  int min, i_min;
  i_min = 0;
  min = t[0];
  int i = 1;
  /*@ loop invariant 0 <= i <= size && 0 ⩽ i_min < size;
      loop invariant \forall integer j;
        0 <= j < i ==> min <= t[j];
      loop invariant t[i_min] == min;
      loop assigns min, i_min, i;
      loop variant size - i; */
  for (int i = 1; i < size; i++) {
    if (t[i] < min) {
      i_min = i;
      min = t[i];
    }
  }
  return i_min;
}
```

(b) Annotations ACSL de cette fonction

FIGURE 2.4 : Exemple d'utilisation de ACSL pour spécifier une fonction C

ACSL. La plateforme utilise ACSL (ANSI/ISO C Specification Langage) [Bau+21b], un langage de spécification qui permet d'ajouter des annotations sous forme de commentaires spéciaux dans le code, pour annoter le programme avec des contrats, des invariants de boucle ou encore des assertions. Ces annotations sont présentes dans l'AST et utilisables par les différents greffons et analyses pour vérifier le code.

La Figure 2.4 illustre un exemple de code C, avec une fonction de recherche de l'index du minimum d'un tableau, et de sa spécification en utilisant ACSL. Le contrat de la fonction requiert en précondition que le tableau ne soit pas de taille nulle (ligne 1) et qu'il soit valide de déréférencer l'ensemble des cases du tableau (ligne 2). Les postconditions demandent à ce que l'index retourné comme résultat soit bien dans le tableau (ligne 3), et qu'il pointe effectivement vers le minimum du tableau (ligne 4). Enfin, la clause assigns spécifie que cette fonction ne modifie aucune variable globale (ligne 6). Ce contrat est correct, mais ne suffira pas aux analyses pour vérifier la fonction. En effet, les boucles sont dures à vérifier de manière automatique, et c'est pourquoi ACSL permet de spécifier également les boucles. Ici, on a donc trois invariants, qui doivent être valides avant et après chaque itération de la boucle, qui vérifient que les index utilisés sont toujours dans les bornes du tableau (ligne 12), que toutes les cases déjà visitées sont inférieures ou égales au minimum (ligne 13), et que l'index `i_min` correspond effectivement au minimum courant trouvé dans le tableau à chaque instant (ligne 15). On indique également quelles variables sont modifiées par la boucle (ligne 16), et enfin le variant de la boucle (ligne 17). Ce variant est une expression utilisant des variables modifiées par la boucle pour déterminer une borne supérieure au nombre d'itérations restantes de la boucle. Le variant est toujours supérieur ou égal à 0 pendant l'exécution de la boucle et décroît strictement à chaque itération.

Toutes ces annotations permettent, à l'aide par exemple de greffons vérification déductive, de prouver que cette fonction est correcte par rapport à sa spécification.

API. Pour terminer, la plateforme Frama-C met à disposition de l'utilisateur une interface de programmation d'application (API, pour Application Programming Interface) permettant d'implémenter des greffons et de réaliser ses propres analyses. La Figure 2.5 illustre l'écosystème autour de Frama-C avec les greffons principaux disponibles, en open-source ou non. Les greffons en rouge sont ceux qui ont été utilisés pendant cette thèse. Frama-C est livrée par défaut avec plusieurs d'entre eux, notamment Wp et Eva que nous allons présenter dans cette section.

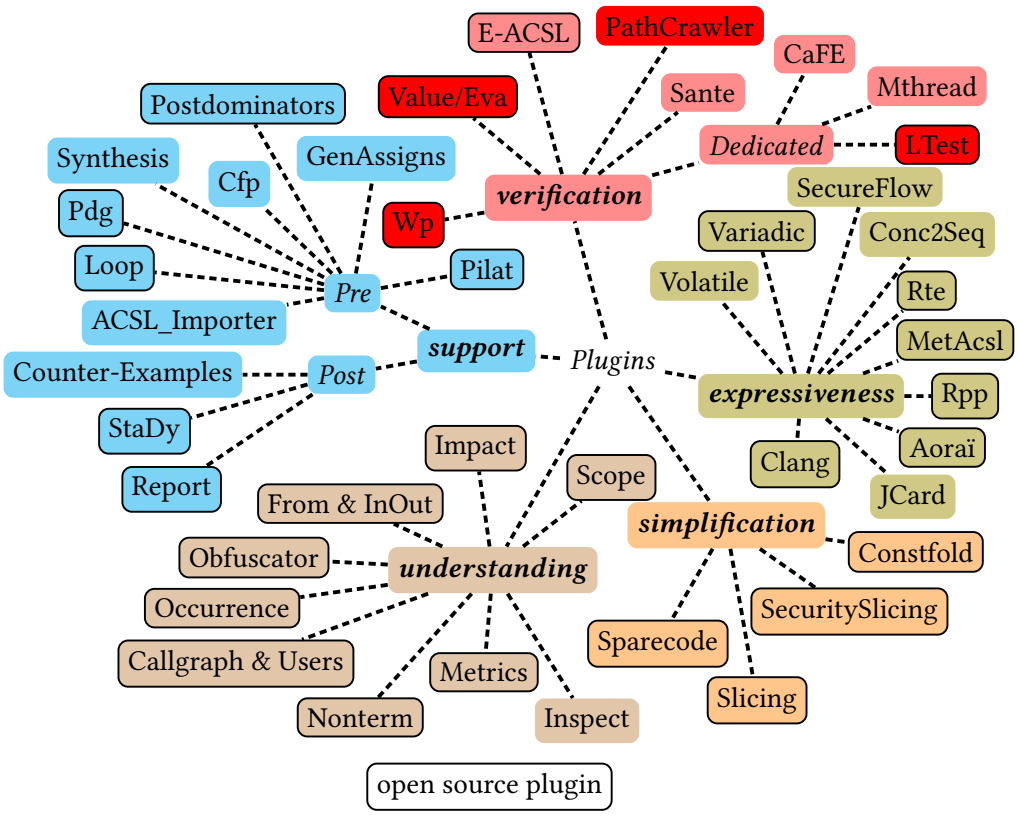


Figure 2.5 : Greffons de Frama-C

2.4.2 Analyse de flot de données

On a présenté dans la Section 2.1.4 le concept d'analyse de flot de données pour propager des informations dans le graphe de flot de contrôle. Frama-C intègre dans son noyau un module permettant de réaliser des analyses de flot de données en propageant un état contenant les informations que l'on souhaite utiliser. Le module demande à l'utilisateur de spécifier plusieurs informations nécessaires à l'analyse, notamment :

— l'état initial avant de lancer l'analyse ;

— les opérations à effectuer, avec l'état courant, en fonction du nœud (instruction) visité ;

— les opérations de jointure (merge) d'états, c'est-à-dire par exemple l'opération à effectuer après la visite de deux branches pour savoir quoi garder des états de chaque branche.

On peut alors lancer l'analyse de flot de données, et le résultat obtenu contient alors le graphe de flot de contrôle avec, pour chaque nœud, l'état cal-

```
1  int x, y, sum, res;
2  // D²x = D²y = D²sum = D²res = {Uninit}
3  if(getchar()=='*'){
4    x = 0; y = 0;
5  // D⁵x = {0}, D⁵y = {0}
6  }else{
7    x = 1; y = 1;
8  // D⁸x = {1}, D⁸y = {1}
9  }
10 // D¹⁰x = {0,1}, D¹⁰y = {0,1}
11 sum = x + y;
12 // D¹²sum = {0,1,2}
13 //@ assert sum != 0;
14 res = 10/sum;
```

(a) Vraie alarme de division par zéro

```
int x, y, sum, res;
// D²x = D²y = D²sum = D²res = {Uninit}
if(getchar()=='*'){
  x = 0; y = 1;
// D⁵x = {0}, D⁵y = {1}
}else{
  x = 1; y = 0;
// D⁸x = {1}, D⁸y = {0}
}
// D¹⁰x = {0,1}, D¹⁰y = {0,1}
sum = x + y;
// D¹²sum = {0,1,2}
//@ assert sum != 0;
res = 10/sum;
```

(b) Fausse alarme de division par zéro

FIGURE 2.6 : EVA illustré sur deux exemples de code C

culé obtenu lorsqu'on atteint ce nœud. Il est ensuite possible de parcourir ce graphe et d'utiliser les informations recueillies pour d'autres analyses.

2.4.3 Interprétation abstraite avec EVA

EVA est l'un des greffons principaux de FRAMA-C. Il permet de réaliser des analyses en utilisant l'interprétation abstraite, avec beaucoup de domaines abstraits et d'options pour paramétrer l'analyse, notamment pour obtenir un équilibre satisfaisant entre précision et temps de calcul. Afin de comprendre son fonctionnement, nous allons détailler un exemple simple d'exécution de l'analyse sur une fonction C.

La Figure 2.6, tirée de l'article [Bau+21a], donne deux versions d'un exemple de code C. On note $\mathcal{D}^l_{\mathrm{v}}$ une sur-approximation de l'ensemble des valeurs possibles de v à la ligne l. Dans ce code, on récupère un caractère donné par l'utilisateur et on va modifier `x` et `y` en fonction de ce dernier. On termine sur une division, qui sera la source de notre erreur potentielle. Au début de la Figure 2.6a, nos variables ne sont pas initialisées, ce qui est représenté par le singleton $\{\mathrm{Uninit}\}$ à la ligne 2. On modifie ensuite cet ensemble en fonction des valeurs assignées dans les branches (ligne 5 et 8). En sortant des deux branches, on effectue une jointure (merge) des ensembles de chaque variable. On a donc pour `x` et `y` deux valeurs possibles : 0 et 1. On les additionne ensuite et on obtient alors $\mathcal{D}^{12}_{\mathrm{sum}} = \{0, 1, 2\}$ pour la variable `sum`. On voit ici que c'est une sur-approximation, car en réalité il est impossible d'obtenir 1 à ce stade pour la variable `sum`. On essaie ensuite de vérifier l'assertion, et on échouera, car

`sum` peut valoir 0 ici, menant ensuite à une division par zéro. L'analyse émettra alors une alarme pour nous avertir de cette erreur potentielle. Si on prend maintenant la Figure 2.6b, les seules différences se trouvent sur les lignes 4 et 7. Cette fois on donne à `x` et `y` des valeurs différentes. La suite de l'analyse est similaire, on trouve le même ensemble de valeurs pour `sum`. Comme dans l'autre exemple, on note ici qu'en réalité `sum` ne peut valoir que 1 à la ligne 12, mais la sur-approximation due à la jointure nous fait perdre l'information que `x` et `y` sont différentes. L'analyse émet donc une alarme sur l'assertion qui selon elle peut être violée, mais c'est en réalité une fausse alarme, car comme on vient de voir, `sum` ne peut pas valoir 0 ici.

Il est possible de raffiner l'analyse, de lui donner des options pour être plus précise et éviter certaines fausses alarmes. En pratique, l'utilisateur lance l'analyse avec des paramètres choisis, observe le résultat puis modifie les paramètres pour rendre l'analyse la plus précise possible, souvent au détriment du temps d'exécution. Par exemple, il est possible ici de ne pas joindre les états abstraits de chaque branche et de continuer l'analyse en propageant deux états distincts. Dans notre exemple, on aurait alors pour `sum`, au niveau de l'assertion, deux états abstraits. Dans la Figure 2.6a, ces états seraient $\mathcal{D}^{12}_{\texttt{sum}} = \{0\}$ et $\mathcal{D}^{12}_{\texttt{sum}} = \{2\}$, avec une alarme pour 0, et plus de précision (on est capable de s'apercevoir que l'état concret où `sum` vaut 1 n'est pas accessible/atteignable.). Dans la Figure 2.6b, on aurait deux fois $\mathcal{D}^{12}_{\texttt{sum}} = \{1\}$ et on éviterait ainsi la fausse alarme.

2.4.4 Calcul de plus faible précondition avec Wp

Wp est un greffon de vérification déductive de Frama-C basé sur le calcul de plus faible précondition (weakest precondition en anglais). Il succède à Caveat et permet de vérifier que des programmes C sont bien conformes à des spécifications écrites en ACSL. L'analyse est paramétrée par un modèle mémoire choisi parmi un ensemble de modèles supportés, et génère des obligations de preuves. Ces obligations sont ensuite données à Qed [Cor14], un module de Wp, qui va les simplifier, puis Wp construit la formule Why3 correspondante et utilise ensuite l'API de Why3 pour faire appel aux prouveurs (ou solveurs SMT) automatiques comme Alt-Ergo [Con+18], Z3 [Mou+08] ou CVC4 [Bar+11].

Si on reprend la Figure 2.4b, Wp génère un total de 14 objectifs de preuves pour l'ensemble du programme. Parmi ces objectifs, 10 seront automatiquement prouvés après simplification de Qed, et 4 seront envoyés et vérifiés par Alt-Ergo. La fonction est donc correcte par rapport aux spécifications ACSL.

```
...
label(1,x==y,"CC");
label(2,x!=y,"CC");
label(3,a<b,"CC");
label(4,a>=b,"CC");
if(x==y && a<b)
{ ... }
...
```

(a) Critère Condition Coverage (**CC**)

```
...
label(1,x==y && a<b,"DC");
label(2,x!=y && a<b,"DC");
label(3,x==y && a>=b,"DC");
label(4,x!=y && a>=b,"DC");
if(x==y && a<b)
{ ... }
...
```

(b) Critère Decision Coverage (**DC**)

```
int f(int a, int b){
  // 1 pour true en C
  label(1,1,"FC");
   ...
}
```

(c) Critère Function Coverage (**FC**)

```
int max(int a, int b){
  label(1,1,"STMT");
  if(a>=b){
    label(2,1,"STMT");
    return a;
  }
  else{
    label(3,1,"STMT");
    return b;
  }
}
```

(d) Critère Statement Coverage (**STMT**)

FIGURE 2.7 : Création des objectifs par LANNOTATE selon plusieurs critères

2.4.5 Services de test avec LTEST

LTEST [1] [Mar+17b] est un ensemble d'outils basés sur le test permettant de réaliser de manière automatique différentes étapes dans le processus de tests d'un logiciel.

LANNOTATE. Ce greffon permet de créer des objectifs de test sous forme de labels et d'hyperlabels que nous venons de présenter. L'utilisation de ces briques de base pour former les critères permet à LANNOTATE d'être très générique et de supporter un grand nombre de critères. On a vu dans la Section 2.1.2 qu'il existait beaucoup d'outils dans la littérature ne supportant qu'un nombre restreint de critères. L'objectif de LANNOTATE est de permettre l'ajout du support de nouveaux critères facilement tout en restant compatible avec les autres outils de LTEST. Les critères de flot de données n'étaient pas supportés correctement, et un des objectifs de cette thèse était donc d'améliorer le support et les performances de ces critères. La Figure 2.7 reprend la Figure 2.1, avec cette fois-ci les labels tels que créés par LANNOTATE. Contrairement à la présentation sim-

1. disponible sur https://git.frama-c.com/pub/ltest

```
int f(){
  if( ... ){label(1,1); g();}
  if( ... ){label(2,1); g();}
}
```

(a) Code avec labels

```
1) <l1+l2|; ;>
```

(b) Hyperlabels

FIGURE 2.8 : Création des labels et hyperlabels par LANNOTATE pour le critère FCC

plifiée avec des commentaires dans nos exemples précédents, ils apparaissent sous forme d'appel de fonction (qui est en réalité une macro sans effet) utilisable par les autres outils de LTEST. Les arguments incluent un identifiant, le prédicat et, de manière optionnelle, le critère pour lequel ce label a été généré. Cet outil va également créer une base de données, sous forme d'un simple fichier texte, contenant pour chaque label son identifiant, sa position dans le code et également son statut (couvert, non couvert, infaisable). De même, si on reprend la Figure 2.2, on obtient avec LANNOTATE le résultat de la Figure 2.8. Les hyperlabels sont stockés sous forme de texte dans une autre base de données.

LUNCOV. Cet outil permet d'analyser un fichier C contenant des labels et hyperlabels (obtenus via LANNOTATE) pour détecter des objectifs infaisables. On va pour cela remplacer les labels par des assertions. Par exemple, pour prouver que le label `label(1,1,x<0);` est infaisable, on le remplace par l'assertion ACSL `/*@ assert !(x<0); */` et on essaie de la prouver avec l'analyse statique fournie par d'autres greffons de FRAMA-C (WP et EVA). En fonction du résultat de l'analyse, on va mettre à jour dans la base de données des labels le statut de chaque label. Ce statut pourra ensuite être utilisé par d'autres outils pour par exemple ignorer les labels infaisables lors de la génération de test. En suivant les corrections apportées à LANNOTATE, nous avons pendant cette thèse ajusté LUNCOV pour supporter les critères de flot de données et ainsi détecter correctement les objectifs polluants.

LREPLAY. Ce troisième greffon de LTEST est utilisé pour mesurer la couverture des labels et hyperlabels d'une suite de tests. Étant donné un programme C annoté par LANNOTATE et une suite de tests, LREPLAY va exécuter les tests et mesurer pendant l'exécution la couverture des objectifs. Les bases de données sont ensuite mises à jour en fonction des résultats, puis l'outil fournit un compte rendu donnant le nombre d'objectifs couverts pour chaque critère ainsi que les statistiques globales de couverture.

PathCrawler. Enfin, PathCrawler est l'outil de génération de test utilisant l'exécution symbolique dynamique (introduite dans la Section 2.1.6) pour générer des tests dans l'objectif de couvrir tous les labels. Nous détaillons son fonctionnement dans la Section 2.5

2.5 Génération de tests avec PathCrawler

La génération de tests est un service essentiel pour le test. On peut tester un programme avec une suite de tests réalisée manuellement, mais utiliser des outils de génération de tests permet de tester beaucoup plus automatiquement un programme en visant certains points clés, en essayant d'exécuter tous les chemins possibles et en essayant de maximiser la couverture d'un (ou plusieurs) critère(s) donné(s). Dans cette section, on va détailler l'exécution symbolique dynamique (DSE) utilisée en particulier par l'outil PathCrawler [Wil+04 ; Wil+05 ; Kos08 ; Bot+09 ; Bar+21].

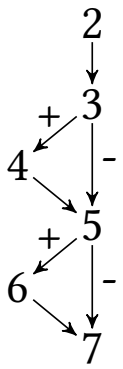
2.5.1 Fonctionnement de PathCrawler

PathCrawler est un greffon de Frama-C [Kir+15] développé au CEA List, dédié à la génération de tests pour des programmes et fonctions C. Il se base sur la technique d'exécution symbolique dynamique mentionnée en Section 2.1.6 pour tenter de générer des tests couvrant tous les chemins possibles (critère **All-paths**), en bornant éventuellement le nombre de tours de boucle à considérer.

Ce greffon est composé de deux modules principaux. Le premier utilise l'API de Frama-C et CIL [Nec+02] pour parser le code C et l'instrumenter pour qu'une exécution du programme affiche le chemin effectivement pris. L'utilisateur peut ensuite spécifier certains paramètres, par exemple donner des bornes aux entrées du programme, avant de lancer le second module, la génération de tests, implémenté en Prolog. Ce module utilise la résolution de contraintes du solveur COLIBRI [Bob+17] développé au CEA List pour générer des tests couvrant tous les chemins, c'est-à-dire pour le critère **All-paths**. Pour expliquer le fonctionnement de PathCrawler, nous allons suivre la présentation de l'algorithme dans l'article [Kos08], avec un exemple similaire.

La Figure 2.9a illustre une fonction qui étant donné un tableau de 3 éléments, renvoie l'élément le plus grand. On peut voir sur la Figure 2.9b le graphe de flot de contrôle simple de cette fonction avec deux tests conditionnels pour un total de quatre chemins possibles. Les nœuds correspondent aux numéros de lignes du programme. On va maintenant regarder comment PathCrawler procède pour couvrir ces quatre chemins.

```
1  int max3(int a[3]){
2    int max = a[0];
3    if(max < a[1])
4      max = a[1];
5    if(max < a[2])
6      max = a[2];
7    return max;
8  }
```



(a) Code en C de `max3` (b) Son graphe de flot de contrôle

Figure 2.9 : Code et CFG de `max3`, qui retourne le maximum dans un tableau de 3 éléments

La Figure 2.10 détaille l'ensemble des étapes de génération en profondeur d'abord de tous les chemins de la fonction `max3` de la Figure 2.9. La colonne Mem représente l'état mémoire de l'exécution symbolique, la colonne Constr. montre les contraintes actuelles données par l'exécution symbolique du chemin courant qui est noté CurPath en dessous des deux colonnes. Enfin sur la droite, on note le cas de test généré avec la valeur attribuée à chaque variable en respectant les contraintes, ainsi que le chemin exécuté par ce cas de test. Les + et − sur CurPath et PathEnd indiquent respectivement s'il faut prendre la branche positive ou négative du test. Les notations snd et fst nous indiquent sur CurPath si la branche contraire de cette condition a déjà été visitée ou non dans le chemin partiel jusqu'à ce nœud. Par exemple, $2, 3^{+}_{\text{fst}}, 4, 5^{+}_{\text{snd}}$ donne dans la Figure 2.9b le chemin partiel jusqu'au nœud 5, avec les deux tests à vrai. La mention fst nous indique que le chemin partiel $2, 3^{-}$ n'a pas été visité avant et snd au contraire signale que le chemin partiel $2, 3^{+}, 4, 5^{-}$ a déjà été visité. Enfin, ⟨precond⟩ représente les préconditions (contraintes sur les variables logiques) de la fonction avant la génération, paramétrables par l'utilisateur. Par exemple ici, on pourrait vouloir contraindre les valeurs des entrées :

$$X_0 \in [0, 255],\ X_1 \in [0, 255],\ X_2 \in [0, 255].$$

On déroule ensuite l'algorithme donné dans l'article pour réaliser la génération par profondeur d'abord.

Étape 1. On initialise l'algorithme, chaque variable d'entrée se voit assigner une variable symbolique, le chemin courant est vide, et les contraintes sont égales aux préconditions de `max3` ;

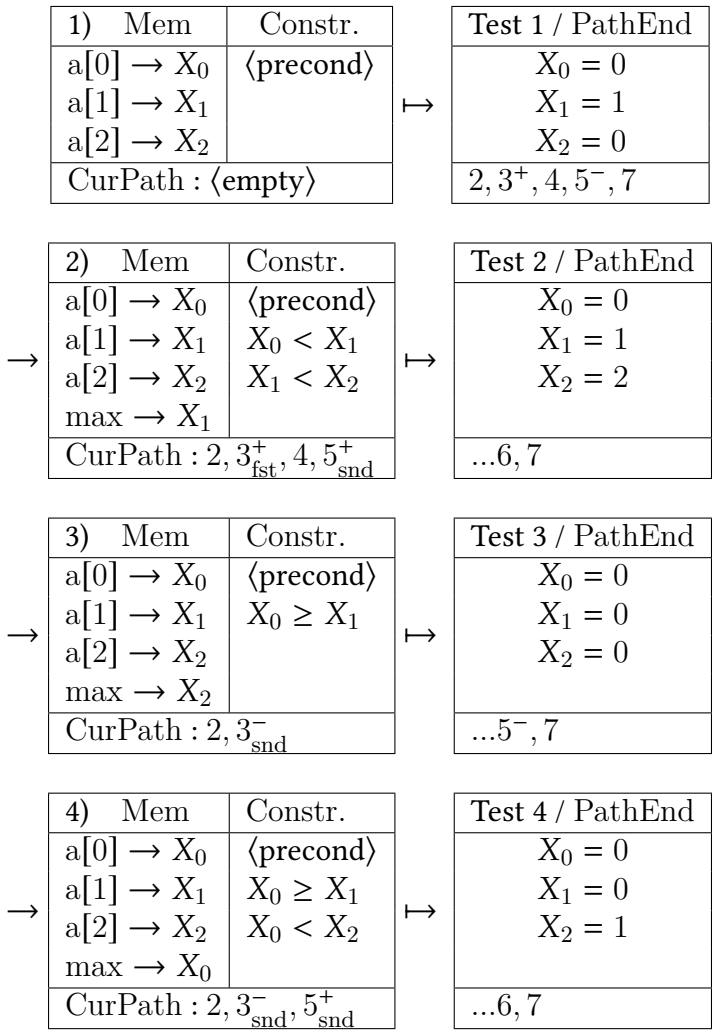


FIGURE 2.10 : Génération par profondeur d'abord des tests du critère **All-paths** pour la fonction `max3` de la Figure 2.9

Étape 2. PathEnd est initialisé comme étant vide. On exécute symboliquement le chemin courant, c'est-à-dire qu'on met à jour les contraintes et l'état mémoire en fonction de CurPath. Selon le résultat, on continue avec l'**étape 5** en cas d'échec, c'est-à-dire infaisabilité, des contraintes, et l'**étape 3** sinon ;

Étape 3. Le solveur de contraintes (ici COLIBRI) est appelé pour générer un cas de test satisfaisant les contraintes, et assigne à chaque variable d'entrée une valeur concrète. Comme à l'étape précédente, en cas d'échec on procède avec l'**étape 5**, sinon on poursuit avec l'**étape 4** ;

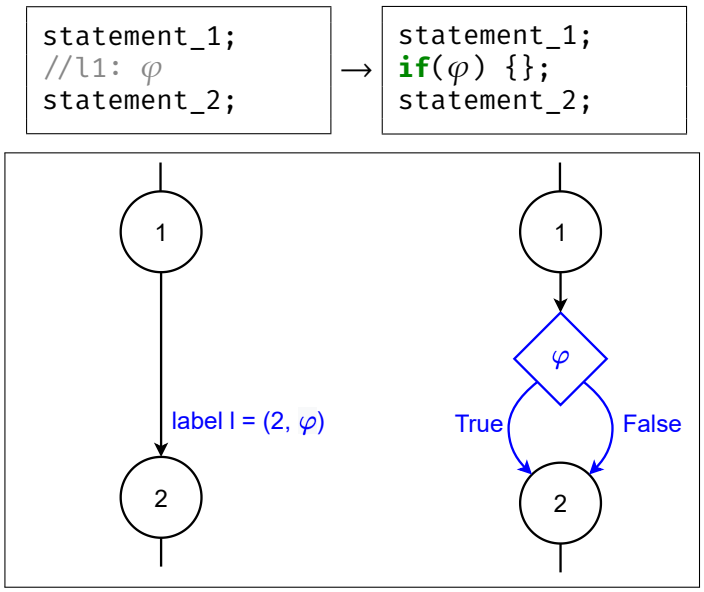


Figure 2.11 : Instrumentation directe d'un label

Étape 4. On exécute ici le cas de test et on observe le chemin exécuté. Le début du chemin doit correspondre à CurPath. La suite du chemin est stocké dans PathEnd ;

Étape 5. On appelle AllPath la concaténation de CurPath et PathEnd. Si AllPath ne contient plus de branche non visitée (donc que tous les nœuds sont tagués avec snd), alors on s'arrête. Dans le cas contraire, on cherche le dernier nœud θ (le plus profond dans le chemin) encore à $\theta^{\pm}_{\text{fst}}$. On affecte alors à CurPath le sous-chemin allant jusqu'à θ (exclu) auquel on ajoute donc sa branche inverse, à savoir $\theta^{\mp}_{\text{snd}}$. On retourne ensuite à l'**étape 2**.

On répète ces étapes jusqu'à l'exploration complète du graphe de flot de contrôle. Si on reprend l'exemple de `max3` et de la Figure 2.10, voici comment se déroule la génération. Tout d'abord, on initialise avec **étape 1** notre algorithme. À ce stade CurPath est donc vide et nos contraintes ne contiennent que les préconditions, ici nous allons considérer qu'il n'y en a pas, les **étapes 2** et **3** sont donc triviales. On passe à l'**étape 4**. Comme il n'y a pas de contraintes, on génère un cas de test aléatoire et on l'exécute pour obtenir PathEnd. On effectue ensuite l'**étape 5** pour obtenir AllPath et on cherche le dernier nœud à fst. Ici, c'est le 5, on l'inverse donc et on met à jour le chemin partiel CurPath comme il se doit, puis on redémarre à l'**étape 2**. Cette fois-ci, il y aura deux contraintes à respecter ($X_0 < X_1$ et $X_1 < X_2$) pour pouvoir exécuter CurPath, on génère un cas de test respectant ces contraintes et ainsi de suite. Ce sera

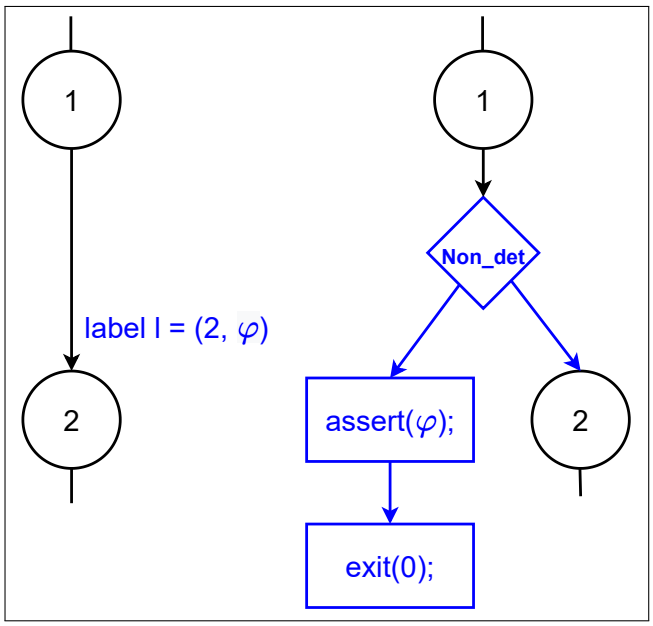


FIGURE 2.12 : Instrumentation compacte d'un label

ensuite au test du nœud 3 d'être inversé, puis de nouveau celui du nœud 5. Après ce dernier test généré, on constate que toutes les conditionnelles sont à l'état snd, il n'y a donc plus rien à visiter et la génération s'arrête.

2.5.2 PATHCRAWLER et les labels

La méthode de PATHCRAWLER que l'on vient de détailler fonctionne parfaitement sur un programme classique, mais ne prend pas directement en compte les labels. En effet, l'algorithme concerne le critère **All-paths** et ne vise donc pas spécialement à couvrir ces derniers. Or ce qui nous intéresse justement, c'est d'essayer de couvrir en priorité les labels afin de générer une suite de tests la plus petite possible maximisant leur couverture. Plusieurs modifications ont été apportées pour prendre en compte les labels [Bar+14b] et nous allons en détailler certaines dans cette section.

Une solution naïve pour intégrer le support des labels est simplement de les transformer en condition C. Comme la génération essaie de couvrir tous les chemins via le critère **All-paths**, elle essaiera donc de couvrir les deux branches du prédicat du label et donc le label sera couvert par un cas de test (si un cas de test le couvrant existe). La Figure 2.11 montre à quoi ressemblerait cette instrumentation.

Mais cette solution introduit quelque chose qui va gêner la génération :

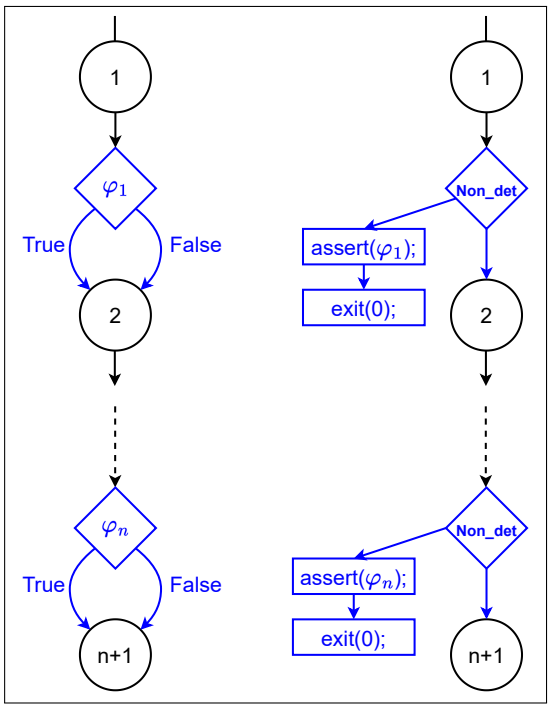


FIGURE 2.13 : Comparaison des instrumentations directe et compacte

l'explosion de chemins. En effet, chaque label est traduit en condition, ce qui crée deux branches et double le nombre de chemins initiaux. On voit bien dans l'exemple de cette figure que nous avons un seul chemin au départ, mais que le label, une fois traduit, fait passer ce nombre à deux. Avec deux labels, nous serions passés de 1 à 4 et ainsi de suite. On a donc une complexité exponentielle, en le nombre de labels, du nombre de chemins à couvrir par notre générateur de tests. Pour remédier à cette explosion, une première optimisation a été apportée : l'instrumentation *compacte*.

Pour un label $\ell \triangleq (loc, \varphi)$, l'instrumentation compacte assure les propriétés suivantes :

- la contrainte φ ne doit servir qu'à couvrir le label et ne doit pas être propagée dans la suite de l'exécution symbolique.
- la négation $\neg\varphi$ est inutile et ne doit donc pas être forcée.

La Figure 2.12 montre cette optimisation. La plupart des moteurs d'exécution symbolique vont soit fournir une fonctionnalité pour réaliser un choix non déterministe, soit permettre de le simuler, par exemple en utilisant en entrée supplémentaire un tableau symbolique de booléens. Derrière ce point de choix non déterministe, on ajoute une assertion exprimant la condition pour couvrir le label puis termine l'exécution.

```
statement_k;
if(nondet){
  assert(p);
  exit(0);
}
statement_k+1;
```

↓

```
statement_k;
if(nondet && !is_covered(label_state_db,l)){
  assert(p);
  set_covered(label_state_db,l,1);
  exit(0);
}
statement_k+1;
```

FIGURE 2.14 : Suppression des labels couverts

On a donc deux chemins possibles, un qui essaie de couvrir le label puis s'arrête et l'autre qui ignore le label et continue son exécution normale. Grâce au choix non déterministe, on ne garde pas $\neg\varphi$ dans la suite de l'exécution.

Si on compare maintenant les deux approches, avec la Figure 2.13, en instrumentation directe on a donc 2^n chemins et avec l'instrumentation compacte $n + 1$ chemins.

Enfin, une dernière optimisation a été apportée appelée *suppression itérative des labels* (IDL, pour Iterative deletion of labels en anglais). L'objectif est simplement de supprimer les labels au fur et à mesure qu'ils sont couverts pour ne pas essayer de les recouvrir dans une autre exécution. Pour ça l'instrumentation ajoute une table `label_state_db` contenant pour chaque label son état : 0 pour non couvert, 1 pour couvert. On met cette table à jour à chaque couverture de label et on ne cherche à couvrir que les labels qui sont encore à 0 dans la table, comme montré dans la Figure 2.14.

Toutes ces transformations permettent de traiter les labels, mais pas les hyperlabels. Les critères utilisant par exemple des séquences, comme les critères de flot de données, ne sont donc pas supportés par PATHCRAWLER. Nous verrons dans les Chapitres 3 et 6 les solutions que nous avons proposées pour ce problème.

La technique de génération de tests pour les labels réalisée dans PATHCRAWLER a été évaluée et utilisée avec succès dans plusieurs projets par un partenaire industriel MERCE (Mitsubishi Electric R&D Centre Europe) [Bar+18].

Chapitre 3

Langage Mini-C et génération d'objectifs de flot de données

Sommaire

Plusieurs notions vont être utilisées dans les prochains chapitres autour des techniques de test et des contributions de cette thèse. Afin de les clarifier, nous allons introduire dans ce chapitre un langage impératif simple, sur lequel nous allons formaliser ces notions et présenter les algorithmes que nous utiliserons par la suite. Nous nous concentrerons ici sur le flot de données et comment le représenter dans le cadre des techniques de test, en parlant principalement des définitions et utilisations des variables du programme. La Section 3.1 introduira les briques du langage Mini-C, puis nous détaillerons dans la Section 3.2 des notions et définitions sur le flot des données et la validité d'un graphe. Enfin, la Section 3.3 définira des fonctions permettant d'insérer des nœuds dans un graphe, mais également de générer des objectifs de test.

3.1 Introduction d'un petit langage : Mini-C

Le langage Mini-C est un petit langage reprenant en partie le fonctionnement du C, mais de manière simplifiée. Il nous permettra d'exprimer nos contributions à partir de définitions et algorithmes précis.

3.1.1 Briques de base du langage

On va définir les expressions et instructions du langage permettant de modéliser sur des exemples simples les labels et hyperlabels vus dans le Chapitre 2. On commence par définir les opérateurs binaires et unaires, contenant des opérateurs arithmétiques et logiques classiques.

Définition 3 (Mini-C : opérateurs binaires). Opérateurs binaires :

- opérateur arithmétiques : `+` `-` `*` `/`
- opérateur logiques : `||` `&&` `<` `<=` `>` `>=` `==` `!=`

L'ensemble de tous ces opérateurs est noté $\mathbb{B}$.

Définition 4 (Mini-C : opérateurs unaires). Opérateurs unaires :

- opérateur arithmétique : `-`
- opérateur logique : `!`

L'ensemble de tous ces opérateurs est noté $\mathbb{U}$.

À partir de ces opérateurs, on peut maintenant définir ce qu'est une expression de notre langage.

Définition 5 (Mini-C : expressions). Les *expressions* du langage Mini-C reprennent un sous-ensemble des expressions C classiques :

- `n` , une constante entière ;
- `x` , une chaîne de caractères qui représente le nom d'une variable ;
- $\texttt{e}_1 \bullet \texttt{e}_2$, une opération binaire entre deux expressions où $\bullet$ est dans $\mathbb{B}$ (voir la Définition 3) ;
- $\circ$ `e` , une opération unaire sur une expression où $\circ$ est dans $\mathbb{U}$ (voir la Définition 4).

L'ensemble de toutes ces expressions est noté $\mathbb{E}$. On notera toujours `e` ou e_i une expression appartenant à $\mathbb{E}$. L'ensemble des variables sera noté $\mathbb{X}$.

Exemple 6. Les expressions suivantes sont correctes, en supposant que a, b et c sont des variables locales définies et initialisées :

- `a + b`
- `-42 <= (b * c)`
- `!(a >= 0) && (b || c)`

Remarque 3. Ici, les expressions sont pures, c'est-à dire qu'elles ne contiennent pas d'effets de bord, contrairement au langage C. Les opérations logiques fonctionnent comme en C et renvoient `0` (faux) ou `1` (vrai). De même, toutes les valeurs différentes de `0` sont considérées comme vraies. Enfin, les variables locales doivent être déclarées pour pouvoir être utilisées (voir la Définition 6).

Nos expressions ne contiennent pas de notions plus compliquées comme les pointeurs, ou même les appels de fonction que nous avons choisi de modéliser comme étant des instructions (voir la Définition 6) principalement pour éviter les effets de bord dans les expressions. Elles sont toutes de type entier, qui est donc le seul type représentable dans notre langage.

Maintenant que nos expressions sont définies, on peut introduire les instructions du langage. Les instructions du langage Mini-C seront associées aux nœuds du graphe de flot de contrôle défini ci-dessous (Définition 14). Un exemple pour une fonction sera donné après les définitions dans la Figure 3.1. Les instructions constituent un sous-ensemble des instructions C classiques auxquelles on ajoute quelques instructions spéciales.

Définition 6 (Mini-C : instructions). Une instruction du langage Mini-C est l'une des instructions suivantes :

- `Begin` sert principalement à rendre explicite le point d'entrée d'une fonction dans le graphe de flot de contrôle (voir la Définition 14) ;
- `return x;` sort de la fonction courante. Cette instruction renvoie la valeur de la variable `x`. On ne considère qu'une seule instruction de ce type par fonction. Comme notre langage ne possède qu'un seul

type `int`, toutes les fonctions renvoient un entier ;

- `int x = e;` est une déclaration et initialisation de la variable locale `x`. Elle est déclarée jusqu'à la fin de la fonction ;
- `x = e;` est une affectation (ou définition) qui stocke la valeur de l'expression `e` dans la variable `x` si elle est déclarée ;
- `x = foo(`e_1`, `e_2`,...,`e_n`);` est un appel de la fonction `foo` avec en arguments une liste d'expressions qui peut être vide. Ces arguments sont ajoutés dans les paramètres formels correspondants de la fonction qui seront donc déclarés et initialisés au même titre que l'instruction `int x = e;` (voir la Définition 14 de notre graphe et la Remarque 8). Un appel de fonction est toujours sous la forme d'une affectation, ici sur la variable `x`, avec la valeur de retour de la fonction. La fonction appelée doit être présente dans l'ensemble $\mathbb{F}$ des fonctions du programme (voir la Définition 9) ;
- `if(e)` est un saut conditionnel qui dépend de la valeur de l'expression `e`.

L'ensemble de toutes ces instructions est noté $\mathbb{I}$.

Remarque 4. La contrainte de n'avoir qu'un seul nœud return par fonction n'induit pas de perte de généralité, car il est toujours possible de transformer une fonction avec plusieurs points de sortie en une fonction avec un point de sortie unique, avec des `goto` et labels C par exemple.

En plus de ces instructions, on ajoute deux nouveaux types d'objets qui nous serons très utiles pour décrire les objectifs : les labels et les checks. Une annotation est un objet qui nous servira à représenter les objectifs de test et les propriétés à vérifier afin d'appliquer les différentes techniques que nous présenterons plus en détail dans les chapitres suivants.

Définition 7 (Annotations). On définit deux types d'annotations :

- les labels, représentés par `label(id,e)`,
- les checks, représentés par `check(id,e)`,

où `id` est un identifiant unique et `e` une expression de notre langage. L'ensemble des annotations est noté $\mathbb{A}$.

Remarque 5. L'identifiant unique est en pratique un entier. Dans le cadre de la thèse, on s'autorisera à utiliser des symboles mathématiques pour améliorer la lisibilité des exemples.

Remarque 6. La forme `label(id,e)` est en théorie équivalente à la forme "commentaire" des labels vue dans la Section 2.3 du Chapitre 2. À partir de maintenant, la forme de la Définition 7 sera privilégiée pour mieux correspondre à la réalité des labels de nos travaux.

Comme nous avons vu dans le Chapitre 2, les labels sont utilisés pour représenter des objectifs. Ce ne sont pas des instructions classiques et c'est pour cela qu'ils ne sont pas formellement représentés comme tels. Nous verrons dans la Définition 14 comment ils seront intégrés dans nos fonctions.

Définition 8 (Fonction). Une fonction est simplement une liste d'instructions qui commence par `Begin` et finit par `return`, avec une liste de paramètres d'entrées. Cette liste sera obtenu à partir du prototype de la fonction, par exemple `int f(int a, int b);`. On représentera nos fonctions directement par des graphes (voir Définition 14).

Définition 9 (Programme). Un programme complet P est défini par un ensemble non-vide de fonctions $\mathbb{F}$ contenant toutes les fonctions de P. De plus, P doit contenir une fonction nommée `main`, faisant office de point d'entrée du programme.

3.1.2 Fonctions utilitaires

Nos instructions utilisent pour beaucoup d'entre elles des expressions. On aura besoin plus tard de pouvoir récupérer ces expressions pour générer nos objectifs. Pour ce faire, on introduit une nouvelle fonction. Étant donné une instruction i, $\mathrm{exprs}(i)$ donne la liste des expressions principales contenus dans l'instruction i.

Définition 10 (exprs : liste des expressions). $\mathrm{exprs} : \mathbb{I} \to \mathbb{E}$ est la fonction des instructions vers l'ensemble d'expressions, définie par les cas sui-

vants :

$$\begin{aligned}
\text{exprs}(\texttt{Begin}) &= \varnothing \\
\text{exprs}(\texttt{int x = e;}) &= \{\texttt{e}\} \\
\text{exprs}(\texttt{x = e;}) &= \{\texttt{e}\} \\
\text{exprs}(\texttt{return x;}) &= \{\texttt{x}\} \\
\text{exprs}(\texttt{if(e)}) &= \{\texttt{e}\} \\
\text{exprs}(\texttt{x = foo(}e_1\texttt{, }e_2\texttt{,...,}e_n\texttt{);}) &= \{e_1, e_2, \ldots, e_n\}
\end{aligned}$$

Remarque 7. On ne parcourt pas récursivement nos expressions pour obtenir les sous-expressions, on ne s'intéresse qu'au premier niveau, les expressions *principales*.

Maintenant que l'on peut récupérer facilement nos expressions, on a besoin également de pouvoir récupérer les variables locales et formelles qui sont présentes (utilisées) dans une expression. Pour ça on introduit une nouvelle fonction qui étant donné une expression e renvoie l'ensemble des variables utilisées dans cette expression.

Définition 11 (lvals : variables dans une expression). $\text{lvals} : \mathbb{E} \to \mathbb{X}$ est la fonction des expressions vers l'ensemble de variables, définie par les cas suivants :

$$\begin{aligned}
\text{lvals}(\texttt{n}) &= \varnothing \\
\text{lvals}(\texttt{x}) &= \{\texttt{x}\} \\
\text{lvals}(\texttt{e}_1 \bullet \texttt{e}_2) &= \text{lvals}(\texttt{e}_1) \cup \text{lvals}(\texttt{e}_2) \\
\text{lvals}(\circ\ \texttt{e}_1) &= \text{lvals}(\texttt{e}_1)
\end{aligned}$$

Exemple 7. Voici quelques exemples de l'utilisation de cette fonction sur des expressions simples de MINI-C :

- $\text{lvals}(\texttt{4 * 2 == 2 * 4}) = \varnothing$
- $\text{lvals}(\texttt{a * a + 1}) = \{\texttt{a}\}$
- $\text{lvals}(\texttt{-42 <= (b * c)}) = \{\texttt{b}; \texttt{c}\}$
- $\text{lvals}(\texttt{!(a >= 0) \&\& (b || c)}) = \{\texttt{a}; \texttt{b}; \texttt{c}\}$

Pour terminer, on introduit deux fonctions qui seront utilisées lors de la génération d'objectifs (voir la Section 3.3) :

Définition 12 (Nouvel identifiant libre). On note `new_id()` la fonction qui retourne un nouvel identifiant unique que nous utiliserons pour la création des labels.

Définition 13 (Création d'une variable locale). Afin de créer une nouvelle variable locale manuellement, on utilisera la fonction `make_fresh_var(`f`,id,x)` qui crée une variable, l'ajoute dans les locales d'une fonction (voir la Section 3.1.3) et retourne son nom qui est composé d'un préfixe, d'un nom de variable et d'un identifiant unique (donnés en paramètres).

3.1.3 Graphe de flot de contrôle

Effectuer des opérations et modifications directement sur le code peut être fastidieux et non trivial à formaliser. Afin de faciliter notre formalisation, nous allons maintenant introduire les graphes de flot de contrôle. Un graphe correspondra au code d'une fonction, nous pourrons ensuite utiliser le graphe pour effectuer des opérations comme l'insertion d'instructions ou leur modification.

Définition 14 (Graphe de flot de contrôle). À une *fonction* f de Mini-C on associe son *graphe de flot de contrôle* (V_f, E_f, L_f, F_f), un graphe dirigé avec les propriétés suivantes :

- V_f et E_f représentent les nœuds (ou sommets) et arêtes du graphe de la fonction f ;
- L_f représente l'ensemble des variables locales définies dans la fonction f ;
- F_f représente l'ensemble des paramètres formels, c'est-à-dire les arguments, de f ;
- Chaque sommet est représenté par un identifiant et une instruction accessible via les fonctions id et instr :

$$\mathrm{id} : V_f \to \mathbb{N}$$
$$\mathrm{instr} : V_f \to \mathbb{I}$$

où $\forall v \in V_f$, $\mathrm{id}(v)$ est un entier et $\mathrm{instr}(v)$ est une instruction dans l'ensemble $\mathbb{I}$ (voir la Définition 6). Chaque sommet possède également un ensemble d'annotations au sens de la Définition 7 accessible via la fonction annots :

$$\mathrm{annots} : V_f \longrightarrow 2^{\mathbb{A}} ;$$

- Chaque fonction possède un unique point d'entrée `Begin` et un unique point de sortie utilisant **`return`**.

  $$\exists!(v_1, v_2) \in V_f^2, \quad \mathrm{instr}(v_1) = \texttt{Begin} \ \wedge \ \mathrm{instr}(v_2) = \texttt{return x;}$$

  Comme ces deux sommets sont uniques dans une fonction, on les appellera $\mathrm{begin}(V_f)$ et $\mathrm{end}(V_f)$;
- $\mathrm{begin}(V_f)$ est le seul nœud qui n'a pas de prédécesseur ;
- $\mathrm{end}(V_f)$ est le seul nœud qui n'a pas de successeur ;
- L'instruction **`if`**`(e)` est la seule instruction ayant deux successeurs, tous les autres (excepté $\mathrm{end}(V_f)$) en ont un ;
- Tous les nœuds sont accessibles depuis $\mathrm{begin}(V_f)$;
- Toutes les variables utilisées dans les expressions sont dans L_f ou F_f.

Un nœud dans une fonction est aussi appelé une *instruction*.

Remarque 8. Comme nous l'avons précisé, lors d'un appel de fonction, les arguments seront ajoutés aux paramètres formels F_f de f et donc considérés comme étant déclarés et initialisés dans f. Ainsi, pour un appel de la fonction `f(`$\texttt{e}_1$`,`$\texttt{e}_2$`)` avec comme prototype **`int f(int a, int b);`**, les variables `a` et `b` seront incluses dans $F_{\texttt{f}}$ et auront comme valeurs $\texttt{e}_1$ et $\texttt{e}_2$. On considère que toutes ces définitions sont attachées au nœud $\mathrm{begin}(V_{\texttt{f}})$.

Remarque 9. Les annotations sont attachées à un nœud dans le graphe, mais elles seront matérialisées dans le code en les plaçant avant le nœud en question, avec une exception pour $\mathrm{begin}(V_f)$ où elles seront placées après le nœud.

Remarque 10. Bien qu'aucune instruction particulière ne soit pas prévue pour les boucles, notre langage permet d'en faire via l'instruction `if`, où un successeur d'un nœud contenant cette instruction peut se trouver *avant* ce dernier dans le graphe.

La Figure 3.1 montre un exemple de graphe de flot de contrôle Mini-C avec sa traduction en code C équivalent. Pour simplifier, l'ensemble des annotations d'un nœud ne sera affiché que lorsqu'il est non vide. Nous utiliserons pour la suite la forme code traduit en C pour simplifier la lecture des exemples. Les graphes seront disponibles en annexe.

Cette Définition 14 nous donne les règles pour obtenir un graphe structurellement valide (sa forme) mais il manque également des règles pour valider le contenu en lui-même. Pour cela, nous allons avoir besoin de quelques définitions supplémentaires sur des propriétés de flot de données du graphe.

3.2 Flot de données

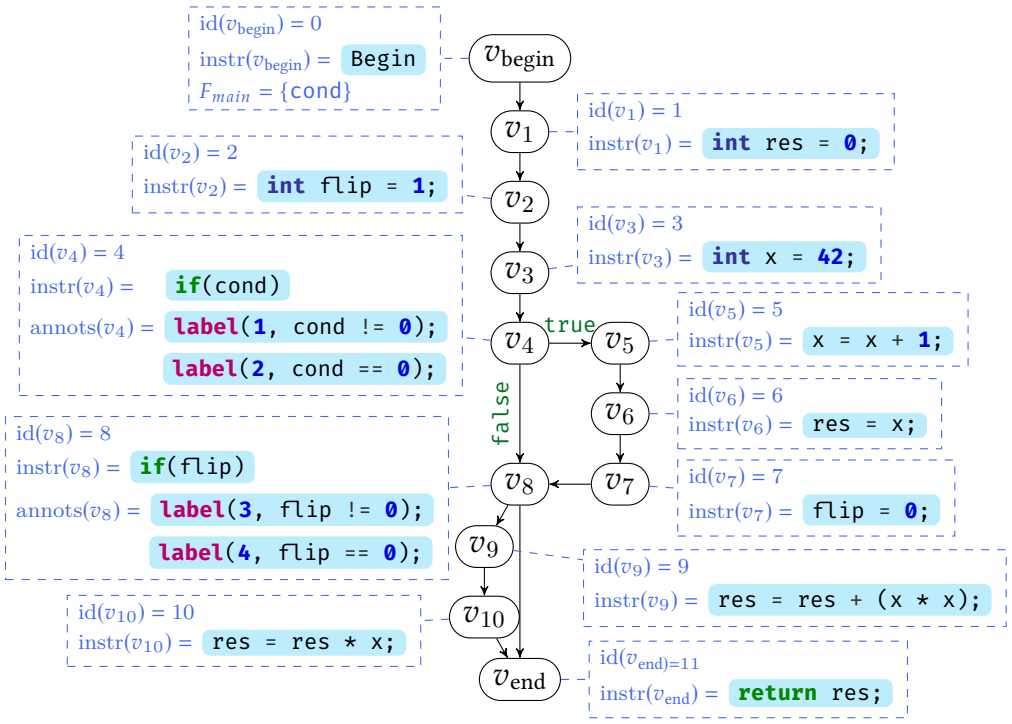
Nos travaux portent en partie sur les critères de couverture de code de flot de données, qui s'appuient sur les définitions et utilisations de variables pour définir des objectifs. On va donc définir deux ensembles permettant d'accéder aux nœuds qui sont des définitions et/ou des utilisations. Étant donné une variable `x` d'une fonction f, $D_f^{\mathtt{x}}$ donne l'ensemble des nœuds qui définissent la variable `x` dans f.

Définition 15 (Ensemble des définitions).

$$\begin{aligned} D_f^{\mathtt{x}} \triangleq \{v \in V_f \mid \mathrm{instr}(v) = \ & \texttt{x = e;} \vee \\ \mathrm{instr}(v) = \ & \texttt{int x = e;} \vee \\ (\mathrm{instr}(v) = \ & \texttt{Begin} \wedge \mathtt{x} \in F_f)\} \end{aligned}$$

On ne s'occupe pas de la valeur de `e` ici.

Remarque 11. Les formels F_f de la fonction f (voir la Remarque 8) sont donc également incluses dans nos ensembles de définitions. Ainsi, si la




```c
int main(int cond) {
  int res = 0;
  int flip = 1;
  int x = 42;
  label(1,cond != 0);
  label(2,cond == 0);
  if (cond) {
    x = x + 1;
    res = x;
    flip = 0;
  }
  label(3,flip != 0);
  label(4,flip == 0);
  if (flip) {
    res = res + x * x;
    res = res * x;
  }
  return res;
}
```

FIGURE 3.1 : Graphe de flot de contrôle MINI-C (a) et son équivalent en C (b)

fonction f possède un paramètre formel nommé x dans F_f, on aura alors :

$$\text{begin}(V_f) \in D_f^{\mathsf{x}}$$

$\text{begin}(V_f)$ est donc un cas particulier de nœud qui peut regrouper plusieurs définitions.

De manière similaire, on note U_f^{x} l'ensemble des nœuds qui utilisent la variable x dans f.

Définition 16 (Ensemble des utilisations).

$$U_f^{\mathsf{x}} \triangleq \{v \in V_f \mid \text{instr}(v) = i,\ \mathsf{x} \in \bigcup_{\mathsf{e} \in \text{exprs}(i)} \text{lvals}(\mathsf{e})\}$$

Exemple 8. À partir de la Figure 3.1, on a par exemple :

- $\text{begin}(V_{\texttt{main}}) \in D_{\texttt{main}}^{\texttt{cond}}$ pour l'unique argument de la fonction.
- $D_{\texttt{main}}^{\texttt{flip}} = \{v_2, v_7\}$ pour l'ensemble des nœuds qui définissent `flip`.
- $v_5 \in D_{\texttt{main}}^{\mathsf{x}}$ et $v_5 \in U_{\texttt{main}}^{\mathsf{x}}$ la définition et l'utilisation de x au nœud v_5.

En plus des définitions et utilisations, les critères de flot de données ont besoin aussi de la notion de chemin dans notre graphe. Pour cela, on définit un chemin comme étant la liste des nœuds dans l'ordre du parcours du chemin.

Définition 17 (Un chemin dans le graphe). Étant donné un graphe de flot de contrôle, on dit que $(v_1, v_2, \ldots, v_n)$ est un *chemin* dans le graphe si $n \geq 2$ et si

$$(v_1, v_2, \ldots, v_n) \triangleq \forall i \in \{1, \ldots, n-1\},\ (v_i, v_{i+1}) \in E_f.$$

Un chemin ne peut pas être constitué de moins de deux nœuds. Tous les nœuds d'un chemin sont dans la même fonction, on ne s'occupe pas des chemins inter-fonctions. On peut alors maintenant récupérer l'ensemble des chemins entre deux nœuds :

Définition 18 (Tous les chemins). Étant donné un nœud de départ v et un nœud d'arrivée v', la fonction $\text{paths}(f, v, v')$ renvoie la liste de tous les chemins existants entre v et v' dans f.

Et à partir de ces définitions, on est maintenant capable d'exprimer des notions indispensables aux critères de flot de données, à savoir la non-redéfinition d'une variable sur un ou plusieurs chemins.

Définition 19 (Chemin sans redéfinition).

$$\begin{aligned}\text{DC}_f(\mathsf{x}, v, v') \triangleq\ & \{(v_1, \ldots, v_n) \in \text{paths}(f, v, v') \mid \\ & \forall k \in \{2, \ldots, n-1\}, v_k \notin D_f^{\mathsf{x}}\}\end{aligned}$$

$$\begin{aligned}\overline{\text{DC}}_f(\mathsf{x}, v, v') \triangleq\ & \{(v_1, \ldots, v_n) \in \text{paths}(f, v, v') \mid \\ & \forall k \in \{1, \ldots, n-1\}, v_k \notin D_f^{\mathsf{x}}\}\end{aligned}$$

$\text{DC}_f(\mathsf{x}, v, v')$, pour def-clear (sans définition), représente l'ensemble des chemins "def-clear", sans définition de la variable x entre v et v'. Les bornes sont donc exclues, on ne prend en compte que les nœuds au milieu du chemin. De manière similaire, $\overline{\text{DC}}_f(\mathsf{x}, v, v')$ est une variante de la définition précédente, où cette fois-ci le premier nœud ne doit également pas définir x.

Grâce aux définitions précédentes, on peut maintenant définir un graphe valide.

Définition 20 (Graphe valide). Pour être valide, un graphe doit dans un premier temps respecter la Définition 14 et donc avoir une structure correcte. Ensuite, les expressions doivent être valides et donc utiliser uniquement des variables locales déclarées et initialisées :

$$\forall \mathsf{x} \in L_f, \forall v \in U_f^{\mathsf{x}}, \overline{\text{DC}}_f(\mathsf{x}, \text{begin}(V_f), v) = \varnothing$$

Enfin, de la même manière que pour les expressions dans les instructions, celles présentes dans les annotations doivent également être valides :

$$\begin{aligned}\forall v \in V_f,\ \forall \mathsf{e},\ & \texttt{label(id,e)} \in \text{annots}(v) \vee \texttt{check(id,e)} \in \text{annots}(v) \\ & \implies \forall \mathsf{x} \in \text{lvals}(\mathsf{e}) \cap L_f,\ \overline{\text{DC}}_f(\mathsf{x}, \text{begin}(V_f), v) = \varnothing\end{aligned}$$

```c
int f(int x, int y){
  int res = 0;
  if(x == y || x < y)
    res = 0;
  else
    res = 1;
  return res;
}
```

(a) Avant instrumentation avec des labels

```c
int f(int x, int y){
  int res = 0;
  label(1, x == y);
  label(2, x != y);
  label(3, x < y);
  label(4, x >= y);
  if(x == y || x < y)
    res = 0;
  else
    res = 1;
  return res;
}
```

(b) Après instrumentation avec des labels

FIGURE 3.2 : Annotation d'une fonction C pour le critère CC

Pour la suite, nous considérerons que tous les graphes utilisés sont valides.

Puisque les paramètres formels de la fonction f sont nécessairement déclarés et initialisés dans $\text{begin}(V_f)$ (cf. Remarque 8), il est suffisant de vérifier les conditions uniquement pour les variables locales (contenues dans L_f).

3.3 Conception de techniques de modélisation et de génération d'objectifs

3.3.1 Modélisation et insertion d'objectifs dans le graphe

On a vu dans le Chapitre 2 plusieurs types de critères permettant d'exprimer des objectifs de test, dans le but de générer et/ou mesurer l'efficacité d'une suite de tests. Nous allons montrer ici comment, à partir d'un graphe, nous allons insérer des nœuds pour représenter ces objectifs et ainsi effectuer l'étape que l'on appelle *annotation d'objectifs de test.*

La Figure 3.2 illustre le résultat de la création d'objectifs pour le critère CC (couverture de conditions) vu dans la Figure 2.1 (les graphes correspondants sont disponibles dans les Figures A.2 et A.3 en annexe). On a inséré quatre labels dans la fonction, au dessus de l'instruction `if(x == y || x < y)`. Même si les labels ne sont pas des instructions dans le graphe, ils sont représentés comme telles dans la version code d'une fonction.

Pour les critères simples, utilisant uniquement des labels, il suffit d'insérer dans notre fonction les instructions correspondantes aux labels qu'on sou-

haite créer. Pour ça nous avons besoin de définir l'opération d'insertion dans le graphe.

Algorithme 1 Insertion d'une instruction dans le graphe

```
1 : procedure insert_before(f, i, v_j)          ▷ Insère i avant v_j dans f
2 :     v_i ← create_node(i)                    ▷ Création d'un nœud contenant i
3 :     V_f ← V_f ∪ {v_i}                       ▷ Ajout du nœud dans le graphe
4 :     for all (v_1, v_2) ∈ E_f do
5 :         if v_2 = v_j then                   ▷ Pour chaque arête allant vers v_j
6 :             E_f ← E_f \ {(v_1, v_2)}        ▷ On supprime cette arête
7 :             E_f ← E_f ∪ {(v_1, v_i)}        ▷ Puis on en crée une nouvelle vers v_i
8 :         end if
9 :     end for
10 :    E_f ← E_f ∪ {(v_i, v_j)}                ▷ On relie v_i et v_j
11 : end procedure
12 :
13 : procedure insert_after(f, i, v_j)          ▷ Insère i après v_j dans f
14 :    v_i ← create_node(i)
15 :    V_f ← V_f ∪ {v_i}
16 :    for all (v_1, v_2) ∈ E_f do
17 :        if v_1 = v_j then                   ▷ Similaire à insert_before
18 :            E_f ← E_f \ {(v_1, v_2)}
19 :            E_f ← E_f ∪ {(v_i, v_2)}
20 :        end if
21 :    end for
22 :    E_f ← E_f ∪ {(v_j, v_i)}
23 : end procedure
```

L'Algorithme 1 illustre le fonctionnement de l'insertion. On crée un nouveau nœud v_i qu'on ajoute dans V_f, puis on redirige les arêtes liées à v_j dans E_f vers v_i. Enfin on ajoute une nouvelle arête entre v_i et v_j. Les arêtes modifiées et ajoutées dépendent de l'emplacement du nouveau nœud par rapport à v_j.

> **Remarque 12.** En théorie ici, on pourrait avoir des problèmes de cohérence de notre graphe si on essayait par exemple d'insérer un nœud avant $\mathrm{begin}(V_f)$ ou après $\mathrm{end}(V_f)$ (**`return`**), ou après un **`if`** (parce que par définition une condition est la seule instruction qui peut avoir plusieurs successeurs). En pratique ces cas ne pourront pas arriver, parce qu'on n'in-

sérera une instruction avant un nœud uniquement si c'est une utilisation (donc pas $\mathrm{begin}(V_f)$) et après uniquement si c'est une déclaration (donc pas $\mathrm{end}(V_f)$ ou **if**).

L'Algorithme 2 montre simplement comment créer et ajouter un nouveau label dans un nœud donné. Cette fonction sera utilisée pour tous les critères que nous verrons ensuite.

Comme nous l'avons vu dans la Section 2.3.2 avec la Définition 2, les hyperlabels permettent d'exprimer des critères de couverture avancés comme les critères de flot de données. Les séquences relient plusieurs labels dans le programme avec une contrainte sur le chemin à emprunter. Si on couvre tous les labels d'une séquence dans l'ordre sans enfreindre la contrainte, alors l'objectif qu'elle représente est couvert. Si on reprend l'exemple des critères de flot de données, on veut créer des séquences reliant les définitions et utilisations de chaque variable du programme entre elles, aussi appelées paires def-use. Dans la suite, nous ne parlerons que de séquences au sens flot de données, donc de paires def-use.

Algorithme 2 Insertion d'un label dans un nœud

```
1 : procedure add_label(v, id, e)          ▷ Insère un label dans annots(v)
2 :     annots(v) ← annots(v) ∪ {label(id,e)}
3 : end procedure
```

Définition 21 (paire def-use modélisée par une séquence). Étant donné une définition au nœud v_i de la variable x et v_j une utilisation de cette même variable dans la fonction f (c'est-à-dire, $v_i \in D_f^{\mathsf{x}}$ et $v_j \in U_f^{\mathsf{x}}$), la paire (v_i, v_j) est une paire def-use. Cette paire def-use est couverte par un test si ce test exécute un chemin qui passe par v_i et plus tard par v_j sans redéfinir x entre ces deux nœuds. La paire def-use (v_i, v_j) est donc modélisée par la séquence $v_i \xrightarrow{\mathrm{dc}(\mathsf{x})} v_j$ où la contrainte $\mathrm{dc}(\mathsf{x})$ signifie que la partie du chemin comprise (strictement) entre v_i et v_j ne doit pas définir x, et donc qu'on souhaite couvrir un chemin p allant de v_i à v_j tel que $p \in \mathrm{DC}_f(\mathsf{x}, v_i, v_j)$ (voir la Définition 19).

Remarque 13. Ici, on utilise directement des nœuds du graphe pour définir notre séquence, plutôt que des labels, pour mieux représenter la réalité de notre approche qui n'utilise pas ces derniers de la même manière que

la définition originale des séquences. Dans le cas des paires def-use classiques, qui sont un type de séquences particulières, on ne s'intéresse qu'à l'atteignabilité, c'est-à-dire la partie loc du label et nous n'avons donc pas besoin de prédicat.

Définition 22 (paire def-use avec des prédicats modélisée par une séquence). Étant donné une définition au nœud v_i de la variable x et v_j une utilisation de cette même variable dans la fonction f (c'est-à-dire, $v_i \in D_f^{\mathsf{x}}$ et $v_j \in U_f^{\mathsf{x}}$), des prédicats φ_s (pour start) et φ_e (pour end) à satisfaire, respectivement, au nœud v_i et au nœud v_j, la paire (v_i, v_j) avec les prédicats φ_s et φ_e est modélisée par la séquence $(v_i, \varphi_s) \xrightarrow{\mathrm{dc}(\mathsf{x})} (v_j, \varphi_e)$ où la contrainte $\mathrm{dc}(\mathsf{x})$ signifie que la partie du chemin comprise (strictement) entre v_i et v_j ne doit pas définir x.

Remarque 14. La notation sous la forme (v_i, φ_s) se rapproche de celle des labels (voir la Définition 1), avec le nœud v_i jouant le rôle de loc et φ_s pour le prédicat ϕ. Même si nous n'utilisons pas directement des labels ici, on remarque qu'en théorie, c'est équivalent.

Ici, φ_s et φ_e sont des prédicats qui doivent être évalués à vrai (respectivement aux nœuds v_i et v_j) pendant l'exécution des tests pour considérer la séquence comme étant couverte, et font donc offices de contraintes supplémentaires. Nous allons maintenant décrire une technique de modélisation à l'aide d'une instrumentation qui permet de déterminer lors de l'exécution du programme si une séquence est couverte. Nous présentons cette technique dans une version plus générale, pour des paires def-use avec des prédicats (qui seront remplacer par vrai pour certains critères simples). La première étape est de modéliser le fait d'avoir exécuté v_i dans un état qui respecte φ_s sans redéfinir x depuis. L'Algorithme 3 illustre la fonction qui insère les contraintes φ_s et $\mathrm{dc}(\mathsf{x})$ de notre séquence dans f pour un nœud source v_i.

Ici, on n'introduit pas de label pour le début de notre séquence, mais on utilise simplement une variable de statut pour vérifier le prédicat φ_s, représenté par la valeur $start$. Plus précisément, la variable $status$ sera vraie si et seulement si l'exécution a traversé le nœud v_i avec un état qui vérifie le prédicat φ_s, et n'a pas rencontré une redéfinition de x depuis.

Pour rappel, on souhaite couvrir un chemin sans redéfinition entre deux nœuds de notre graphe. Avec la variable de statut donnée en argument, on va

Algorithme 3 Insertion de la contrainte $\mathrm{dc}(\mathtt{x})$ et φ_s

```
1 : procedure insert_cstr(f, x, status, v_i, start)
2 :     if v_i ≠ begin(V_f) then            ▷ Si le nœud n'est pas le point d'entrée
3 :         insert_after(int status = 0;, begin(V_f))
4 :         insert_after(status = start;, v_i)
5 :     else
6 :         insert_after(int status = start;, begin(V_f))
7 :     end if
8 :     for all v_k ∈ D_f^x do              ▷ Chaque redéfinition réinitialise la séquence
9 :         if v_k ≠ v_i ∧ v_k ≠ begin(V_f) then
10 :            insert_after(status = 0;, v_k)
11 :        end if
12 :    end for
13 : end procedure
```

pouvoir modéliser les contraintes $\mathrm{dc}(\mathtt{x})$ et φ_s dans la fonction. On commence par déclarer cette variable de statut et l'initialiser en insérant une instruction, avec comme valeur pour l'initialisation **0** ou *start*, selon si le nœud de la définition est également le premier nœud dans le graphe (voir la Remarque 11). En fait l'initialisation directe à *start* dans le cas où $v_i = \mathrm{begin}(V_f)$ est une version courte pour ne pas avoir à initialiser *status* à **0**, puis à l'affecter juste après, comme on fait pour les autres définitions. Si le nœud v_i n'est pas le nœud $\mathrm{begin}(V_f)$, alors on insère l'instruction qui affecte *status* à *start* après v_i.

Ensuite, on va mettre la variable de statut à **0** pour tous les nœuds v_k dans V_f étant des définitions de la variable observée et différents de v_i et de $\mathrm{begin}(V_f)$ en insérant après v_k l'instruction *status* = **0**;. Cette remise à 0 nous sert à représenter la violation de la contrainte : la variable x est redéfinie.

> **Remarque 15.** L'ordre de ces deux étapes n'a pas d'importance, puisqu'on ne casse une séquence pour une définition donnée que si celle-ci est différente de v_i. Sans cette contrainte, l'ordre serait important puisqu'il faudrait s'assurer qu'on n'interrompt pas une séquence qui vient de démarrer.

Pour simplifier la description des exemples sur les critères de flot de données, nous allons utiliser des labels C pour représenter des nœuds de notre graphe. Nous parlerons de ces labels C plutôt que des lignes du code. Par exemple dans la Figure 3.3, le nœud **v2** fait référence à la ligne 3 dans le code d'origine et la ligne 5 après l'insertion.

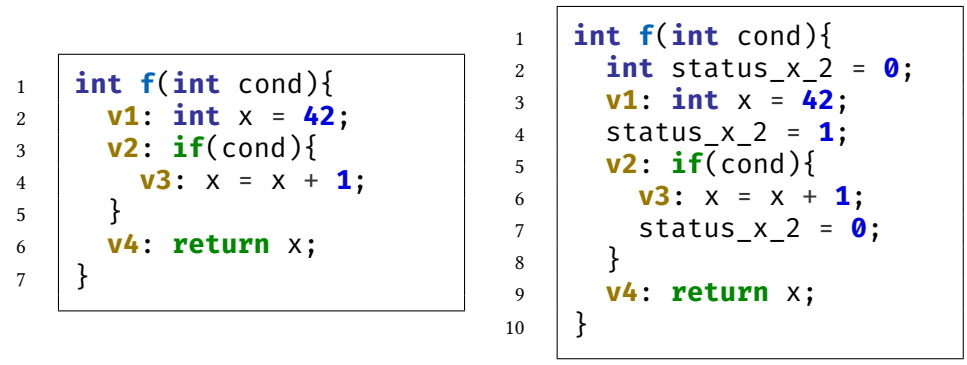


FIGURE 3.3 : Ajout de la contrainte `insert_cstr(`f`,x,status_x_2,`**`v1`**`,`**`1`**`)`

Exemple 9. La Figure 3.3 illustre l'insertion de la contrainte (à l'aide de la fonction `insert_cstr` de l'Algorithme 3) sur la variable `x` et la définition au nœud **`v1`**. On crée la variable `status_x_2` initialisée à **`0`**, puis on lui affecte **`1`** sous la définition **`v1`** et **`0`** au-dessous de toutes les autres (c'est-à-dire uniquement **`v3`** ici). Si la variable `status_x_2` est évaluée à **`1`** au nœud **`v4`**, c'est qu'on a exécuté la définition **`v1`** et on a ensuite parcouru un chemin sans redéfinition de `x` jusqu'à **`v4`**, donc qu'on n'est pas rentré dans la condition.

Remarque 16. Les paires def-use de base sont un cas particulier de séquences où les prédicats φ_s et φ_e sont définies à *vrai*, car on veut simplement les atteindre, ils sont donc automatiquement vérifiés. C'est pourquoi ici on affecte simplement **`1`** à notre variable de status. Nous verrons dans le Chapitre 5 des cas de séquences où les prédicats φ_s et φ_e ne sont pas trivialement vrais.

Grâce à cette contrainte, nous sommes maintenant capable d'insérer une paire def-use dans notre graphe.

L'Algorithme 4 illustre la fonction qui insère la paire $(v_i, \varphi_s) \xrightarrow{\text{dc(x)}} (v_j, \varphi_e)$ dans f avec φ_s et φ_e valant respectivement *start* et *end*. On commence par récupérer un nouvel identifiant **`id`** et créer la variable *status*. Avec `make_fresh_var` on crée et récupère le nom d'une nouvelle variable (voir la Définition 13). Maintenant que **`id`** et *status* sont prêtes, on peut insérer la paire correspondante. On crée nos prédicats *start* et *end* pour les contraintes φ_s et φ_e, on insère toutes les contraintes, puis on introduit un label dans le nœud comme pour l'exemple précédent sur le critère CC qui devra vérifier que la contrainte *cstr* est vérifiée, ainsi que *end*.

Algorithme 4 Insertion d'une paire def-use

```
1 : procedure insert_du(f, x, v_i, v_j)
2 :     id ← new_id()                          ▷ Création d'un nouvel identifiant
3 :     status ← make_fresh_var(f,id,x)        ▷ Création d'une variable
4 :     start ← 1                              ▷ Le prédicat de départ est toujours vrai
5 :     cstr ← status == 1                     ▷ Condition de couverture de la séquence
6 :     end ← 1                                ▷ Le prédicat de fin est toujours vrai
7 :     insert_cstr(f, x, status, v_i, start)  ▷ Ajout de la contrainte
8 :     add_label(v_j, id, cstr && end)        ▷ Et du label
9 : end procedure
```

```
int f(int cond){
  v1: int x = 42;
  v2: if(cond){
    v3: x = x + 1;
  }
  v4: return x;
}
```

```
int f(int cond){
  int status_x_2 = 0;
  v1: int x = 42;
  status_x_2 = 1;
  v2: if(cond){
    label(2, status_x_2 == 1 && 1);
    v3: x = x + 1;
    status_x_2 = 0;
  }
  v4: return x;
}
```

FIGURE 3.4 : Insertion de la paire (**v1**, **1**) $\xrightarrow{\text{dc(x)}}$ (**v3**, **1**) (φ_s et φ_e valent **1**)

Remarque 17. De même que la Remarque 16, le prédicat φ_e est simplement **1** ici, puisqu'on cherche uniquement à vérifier qu'il est atteint.

Exemple 10. De la même manière que pour l'exemple précédent, on insère dans la Figure 3.4 notre objectif complet (à l'aide de la fonction `insert_du` de l'Algorithme 4) pour la paire (**v1**, **1**) $\xrightarrow{\text{dc(x)}}$ (**v3**, **1**). On ajoute la contrainte comme dans l'Exemple 9, puis le label au nœud **v3**. On suppose ici que l'identifiant unique **id** retourné est égale à 2 et le préfixe utilisé est `status` (d'où le nom `status_x_2` pour la variable de statut).

Pour simplifier, on omettra la partie `&& 1`, qui représente le prédicat de fin φ_e, lorsqu'il est toujours vrai, ce qui est en fait le cas pour les séquences de flot de données classiques.

3.3.2 Génération d'objectifs

On est maintenant capable de générer tous les objectifs d'une fonction donnée en créant tous les couples de séquences de flot de données possibles dans le programme. On va donc insérer nos séquences dans le graphe, en utilisant la fonction `insert_du` de l'Algorithme 4. L'Algorithme 5 décrit cette opération.

Algorithme 5 Création de toutes les paires def-use

```
1 : procedure alldu(f)
2 :     L ← L_f ∪ F_f                    ▷ Ignorer les variables créées par cet algorithme
3 :     for all x ∈ L do                                  ▷ Pour chaque variable
4 :         for all v_j ∈ U_f^x do                        ▷ Pour chaque utilisation
5 :             for all v_i ∈ D_f^x do                    ▷ Pour chaque définition
6 :                 insert_du(f, x, v_i, v_j)             ▷ Création d'une paire
7 :             end for
8 :         end for
9 :     end for
10 : end procedure
```

Pour chaque variable `x` appartenant aux locales L_f ou aux formelles F_f de la fonction f (avant les ajouts de variables supplémentaires via `insert_du`) et chaque couple de nœuds (v_i, v_j) où v_i est une définition de `x` et v_j une utilisation de `x`, on crée un objectif (ou séquence) de flot de données.

De cette façon, on crée donc toutes les paires def-use possibles dans notre graphe de flot de contrôle. Si on reprend la Figure 3.4 et qu'on ajoute cette fois tous les objectifs, on obtient le code de la Figure 3.5 (dont les graphes illustrés par les Figures A.5 et A.6 sont disponibles en annexe). On a alors quatre séquences pour la variable `x`, car il y a deux définitions et deux utilisations de `x` dans cette fonction, et une séquence pour la variable `cond`. On a donc introduit une variable de statut par séquence et on a affecté ces variables comme décrit dans l'Algorithme 3. On peut ensuite appliquer cette fonction à toutes les fonctions f dans $\mathbb{F}$ de notre programme P.

Cependant, on peut facilement constater que cette méthode n'a pas l'air très efficace. En effet, la séquence 3 entre la définition de **v3** et l'utilisation également sur **v3** ne semble pas couvrable, il faudrait au moins une boucle pour pouvoir la réaliser. Nous verrons ce problème en détail dans le Chapitre 4.

```
1  int f(int cond){
2    v1: int x = 42;
3    v2: if(cond){
4      v3: x = x + 1;
5    }
6    v4: return x;
7  }
```

(a) Avant instrumentation avec des séquences

```
1  int f(int cond){
2    int status_cond_1 = 1;
3    int status_x_2 = 0;
4    int status_x_3 = 0;
5    int status_x_4 = 0;
6    int status_x_5 = 0;
7    v1: int x = 42;
8    status_x_2 = 1;
9    status_x_3 = 0;
10   status_x_4 = 1;
11   status_x_5 = 0;
12   label(1,status_cond_1 == 1);
13   v2: if(cond){
14     label(2,status_x_2 == 1);
15     label(3,status_x_3 == 1);
16     v3: x = x + 1;
17     status_x_2 = 0;
18     status_x_3 = 1;
19     status_x_4 = 0;
20     status_x_5 = 1;
21   }
22   label(4,status_x_4 == 1);
23   label(5,status_x_5 == 1);
24   v4: return x;
25 }
```

(b) Après instrumentation avec des séquences

FIGURE 3.5 : Annotation d'une fonction avec des séquences de flot de données

Chapitre 4

Objectifs polluants pour les critères de flot de données

Sommaire

On a vu dans le chapitre précédent une formalisation permettant d'exprimer des propriétés sur le flot de données et également qu'il était possible de générer toutes les séquences de flot de données pour une fonction Mini-C avec l'Algorithme 5.

Pour générer des objectifs de flot de données, il suffit de créer une paire définition-utilisation pour chaque définition et utilisation d'une même variable du programme. Cependant, cette façon de procéder comporte des désavantages assez évidents. En effet, la définition complexe de ces critères de flot de données, mêlant atteignabilité et chemin sans redéfinition, rend la détection des objectifs polluants difficile : cette méthode simple va générer beaucoup d'objectifs polluants.

La Figure 3.5 du Chapitre 3 montrait un exemple en apparence simple dans lequel on voulait créer des objectifs de flot de données. En utilisant une approche naïve on créait 5 objectifs : un objectif pour l'unique paire def-use de la variable `cond` et quatre objectifs pour la variable `x` (il y a deux définitions et deux utilisations, donc quatre couples possibles). Cependant, la paire allant

```
int main(int cond) {
  int res = 0;
  int flip = 1;
  v1: int x = 42;
  if (cond) {
    v2: x = x + 1;
    v3: res = x;
    flip = 0;
  }
  if (flip) {
    v4: res = res + x * x;
    v5: res = res * x;
  }
  return res;
}
```

FIGURE 4.1 : Exemple d'une fonction C simple

de la définition au nœud **v3** vers l'utilisation à cette même ligne n'était pas réalisable, car aucun chemin ne permettait de remonter dans le code.

Il existe d'autres cas de séquences du même type ne pouvant pas être couvertes. Plus généralement, si la contrainte $\mathrm{dc}(x)$ (voir la Définition 19) n'est pas réalisable : c'est-à-dire qu'il n'existe pas de chemin sans redéfinition entre un couple de nœuds (v_i,v_j) avec $v_i \in D_f^x$ et $v_j \in U_f^x$. Une autre possibilité est qu'il existe des chemins dans le graphe de flot de contrôle, mais qu'ils ne peuvent pas être exécutés (par exemple de **v2** vers **v4** dans la Figure 4.1). De plus cette méthode va créer un très grand nombre d'objectifs. Dès lors, on peut se demander s'il est possible d'en générer moins tout en testant aussi exhaustivement le programme ? Il est crucial d'éviter de perdre du temps pendant la génération de tests (en essayant de couvrir un objectif impossible) et de mesurer correctement la couverture (en ignorant ces objectifs dans le nombre total d'objectifs).

Dans ce chapitre, nous allons présenter trois techniques permettant de détecter ces objectifs dits *polluants*. Dans un premier temps, nous verrons comment utiliser une analyse de flot de données (Section 4.1) pour détecter les objectifs *non-applicables* (Section 4.1.1). Ensuite, nous enrichirons cette analyse pour localiser des objectifs *équivalents* (Section 4.1.2). Enfin, nous montrerons comment prouver, à l'aide d'outils d'analyse statique, que certains objectifs sont *infaisables* (Section 4.2).

4.1 Analyse de flot de données

L'analyse de flot de données [Kil73] paraît logiquement être pertinente quand on parle de critères de flot de données. On va simplement parcourir le programme statiquement et observer son comportement. On propage des informations le long des arêtes de notre graphe de flot de contrôle, visitant chaque état, pour accumuler des informations et ainsi détecter des objectifs polluants. Cette analyse sera décrite plus en détail dans la Section 6.1.1 du Chapitre 6.

4.1.1 Objectifs non-applicables

Les objectifs non-applicables [Fra+88] regroupent l'ensemble des objectifs pour lesquels il n'existe aucun chemin entre une définition et une utilisation d'une variable sans redéfinition de cette même variable.

Exemple 11. La séquence de la Figure 3.5 allant de la ligne de **v3** vers elle-même est dans ce cas. Aucun chemin ne permet d'aller de sa définition vers son utilisation. Dans la Figure 4.1, la définition de x au nœud **v1** ne peut pas atteindre l'utilisation au nœud **v3**, car il n'existe qu'un unique chemin p entre **v1** et **v3** et ce chemin n'appartient pas à $\mathrm{DC}_f(\mathsf{x}, \mathbf{v1}, \mathbf{v3})$: il passe par une redéfinition de notre variable au nœud **v2**. Il n'existe donc pas de chemin entre cette définition et cette utilisation de x sans redéfinition : cet objectif est polluant et non-applicable.

Formellement, on peut définir les objectifs non-applicables de la manière suivante :

Définition 23 (Non-applicable). Parmi les paires def-use dans une fonction f pour une variable x, on dit que la paire pour un couple de nœuds (v_i, v_j) et la variable x est *non-applicables*, et on note $\mathrm{NA}_f(\mathsf{x}, v_i, v_j)$ si

$$v_i \in D_f^{\mathsf{x}} \wedge v_j \in U_f^{\mathsf{x}} \wedge \mathrm{DC}_f(\mathsf{x}, v_i, v_j) = \varnothing.$$

Autrement dit, dans une paire définition-utilisation non-applicable d'une même variable x de la fonction f représentées par deux nœuds v_i et v_j, l'ensemble des chemins sans redéfinition entre ces deux nœuds est vide.

Pour appliquer la Définition 23 dans notre analyse de flot de données et éviter de générer les objectifs non-applicables, on va l'enrichir avec une nouvelle fonction.

Définition 24 (Peut atteindre). Avec la fonction $\mathrm{defmayreach}_f(\mathsf{x}, v_j)$, on obtient l'ensemble des définitions de x dans f qui peuvent atteindre v_j par un chemin sans redéfinition.

$$\mathrm{defmayreach}_f(\mathsf{x}, v_j) \triangleq \{ v_i \in D_f^{\mathsf{x}} \mid \mathrm{DC}_f(\mathsf{x}, v_i, v_j) \neq \emptyset \}$$

Remarque 18. Les valeurs de $\mathrm{defmayreach}$ sont obtenues à l'aide d'une analyse de flot de données classique. On propage le long du graphe de flot de contrôle notre ensemble de définitions pouvant atteindre chaque point de ce graphe, d'un nœud à ses successeurs. Lorsqu'on traverse une définition v de la variable x (qui est donc l'affectation d'une expression ou du résultat d'un appel de fonction), cette définition cache toutes les définitions précédentes, et on propage l'ensemble singleton $\{v\}$ vers le successeur du nœud v. Enfin, lorsqu'un nœud possède plusieurs prédécesseurs, on fait l'union des ensembles propagés depuis ses prédécesseurs.

Pour générer un objectif on procède donc de la manière suivante : au lieu de générer un objectif pour chaque paire définition-utilisation d'une fonction comme dans l'Algorithme 5, on va d'abord filtrer pour ne garder que les couples tels que la définition est dans l'ensemble $\mathrm{defmayreach}$ de cette utilisation. On peut donc mettre à jour notre algorithme de création des objectifs.

Algorithme 6 Création de toutes les paires def-use sans non-applicables

```
1 : procedure alldu_NA_filter(f)
2 :     L ← L_f ∪ F_f
3 :     for all x ∈ L do
4 :         for all v_j ∈ U_f^x do
5 :             for all v_i ∈ defmayreach_f(x, v_j) do    ▷ On filtre les définitions
6 :                 insert_du(f, x, v_i, v_j)
7 :             end for
8 :         end for
9 :     end for
10 : end procedure
```

La contrainte de la séquence $\mathrm{dc}(\mathsf{x})$ est évidemment gardée, car ce n'est pas parce qu'on peut atteindre un nœud par un chemin sans redéfinition que ce sera effectivement le cas. On a maintenant besoin d'exécuter un tel chemin pour

couvrir notre objectif. Avec cette version de notre algorithme, on ne génère donc pas les objectifs non-applicables.

> **Exemple 12.** Dans notre exemple de la Figure 4.1, en regardant ce que contient notre ensemble $\mathrm{defmayreach}_{\texttt{main}}(\texttt{x}, \texttt{v3})$, nous verrons qu'il n'y a qu'une seule définition, celle de **v2**, puisqu'elle a écrasé celle de **v1** et donc nous saurons qu'il ne faut pas générer d'objectif allant de **v1** à **v3**.

Cet algorithme est évalué sur plusieurs exemples de code C dans le Chapitre 6 sous le nom M_{NA}.

4.1.2 Objectifs équivalents

Les objectifs équivalents ne sont pas polluants dans le même sens que les autres : ils sont polluants car redondants. Par conséquent, si deux objectifs sont équivalents, ils seront toujours couverts ou non ensemble et toujours non-applicables (voir Section 4.1.1) ou infaisables (voir Section 4.2) ou non ensemble. Dans le cas des objectifs infaisables, lancer la détection sur deux objectifs équivalents est 2 fois plus long alors que les deux résultats seront similaires, et il suffit donc d'en faire un seul pour connaître l'autre. Dans le cas de la mesure de couverture, les objectifs équivalents vont avoir tendance à dissimuler des problèmes de couverture en faussant le score obtenu. Prenons un exemple simple de code avec la Figure 4.2 dans lequel on veut obtenir un taux de couverture d'au moins 80%. Imaginons maintenant qu'on ait 1 objectif normal dans la branche vraie de la condition et 9 objectifs équivalents dans la branche fausse (**else**). Si une suite de test exécute toujours notre fonction `f` avec `cond == 0`, alors on obtiendra un taux de couverture de 90%, ce qui satisfait la limite de 80% qu'on s'était fixée. Or, si on retire les objectifs équivalents pour n'en garder qu'un, alors cette fois la couverture ne sera plus que de 50%, ce qui ne satisfait plus notre critère, et on détecte alors que la suite est incomplète.

Pour trouver ces objectifs, on va s'intéresser d'abord uniquement aux utilisations de nos variables, la partie *destination* des séquences. On distingue deux types d'objectifs équivalents. Le premier est trivial : si une variable `x` est utilisée plus d'une fois dans la même expression, en faisant l'hypothèse que cette expression ne contient pas d'effet de bord (voir la Remarque 3), la valeur de `x` sera la même pour chaque occurrence. On peut donc se concentrer sur une seule occurrence et ignorer les autres lors de la création de paires définitions-utilisations.

```c
int f(int cond) {
  if (cond) {
     ... // 1 objectif
  }
  else {
     ... // 9 obtectifs équivalents
  }
  return 0;
}
```

FIGURE 4.2 : Exemple d'une fonction C simple pour les équivalents

Exemple 13. Dans notre Figure 4.1, on constate au nœud **v4** que la variable x est utilisée deux fois dans la même expression. Ici il est donc trivial de constater que ces deux occurrences de x viendront de la même définition. Il n'est donc pas utile de tester deux fois et on va donc considérer ici une seule utilisation.

Notre formalisation avec la Définition 16 prend déjà ce cas en compte, car ne pas le prendre en compte rendrait la formalisation artificiellement plus compliquée : on ne garde dans notre ensemble U_f^x qu'une seule occurrence de chaque nœud, il n'est donc pas possible de différencier deux utilisations dans un même nœud. Nous détaillerons dans le Chapitre 6 comment nous gérons ce cas d'objectifs trivialement équivalents. Dans la suite de cette section, nous considérons donc que les objectifs équivalents ne concernent que les objectifs non trivialement équivalents.

Le deuxième type d'objectifs équivalents est logiquement plus complexe. Pour deux utilisations de la même variable $v_i \in U_f^x$ et $v_j \in U_f^x$ avec v_i différent de v_j, on cherche à vérifier les propriétés suivantes :

- Tous les chemins entre le point d'entrée du graphe de flot de contrôle $\text{begin}(V_f)$ et v_j passent par v_i ;
- Tous les chemins entre v_i et le point de sortie du graphe $\text{end}(V_f)$ passent par v_j ;
- aucun chemin entre v_i et v_j ne contient une définition de la variable x.

Exemple 14. Toujours dans la Figure 4.1, on s'intéresse cette fois aux nœuds **v4** et **v5** (en considérant que les utilisations trivialement équivalentes sont déjà prises en compte donc) avec deux utilisations de la variable x. On peut voir qu'on respecte bien les 3 propriétés précédentes : Tous les

chemins qui passent par **v5** passent par **v4** et inversement, et il n'y a pas de définition de x entre les deux. Ces deux utilisations sont donc équivalentes.

On peut définir les utilisations équivalentes de la manière suivante :

Définition 25 (Equivalents). Parmi les utilisations d'une variable x dans une fonction f, on dit que deux utilisations v_i et v_j sont *équivalentes*, et on note $\mathrm{EQ}_f(\mathsf{x}, v_i, v_j)$ si

$$\begin{aligned}
&v_i \neq v_j \ \wedge v_i \in U_f^{\mathsf{x}} \wedge v_j \in U_f^{\mathsf{x}} \wedge \\
&\forall p \in \mathrm{paths}(f, \mathrm{begin}(V_f), v_j),\ v_i \in p \wedge \\
&\forall q \in \mathrm{paths}(f, v_i, \mathrm{end}(V_f)),\ v_j \in q \wedge \\
&\forall r \in \mathrm{paths}(f, v_i, v_j),\ r \in \overline{\mathrm{DC}}_f(\mathsf{x}, v_i, v_j).
\end{aligned}$$

Cette définition correspond aux points évoqués plus tôt, on utilise $\overline{\mathrm{DC}}_f(\mathsf{x}, v_i, v_j)$ pour trouver l'ensemble des chemins entre v_i (inclus) et v_j qui ne définissent pas x.

Si on a $\mathrm{EQ}_f(\mathsf{x}, v_i, v_j)$, alors on est certain que si une de nos deux utilisations à v_i ou à v_j est exécutée, l'autre le sera également (avec la même définition de x). Inversement, si une utilisation n'est pas exécutée, cela implique que la deuxième ne le sera pas non plus.

De la même manière que pour les objectifs non-applicables, on va enrichir notre analyse en créant plusieurs fonctions. La première propriété que l'on doit avoir pour notre équivalence est en fait appelée *domination* [Pro59].

Définition 26 (Domination). La fonction $\mathrm{dominate}_f(\mathsf{x}, v_j)$ nous donne l'ensemble des nœuds qui utilisent x et qui *dominent* v_j.

$$\begin{aligned}
\mathrm{dominate}_f(\mathsf{x}, v_j) \triangleq\ & \{v_i \in U_f^{\mathsf{x}} \mid v_i \neq v_j \wedge \\
& \forall p \in \mathrm{paths}(f, \mathrm{begin}(V_f), v_j),\ v_i \in p\}
\end{aligned}$$

La deuxième propriété, appelée *post-domination*, peut être donnée similairement par la définition suivante :

Définition 27 (Post-domination). La fonction $\text{post-dominate}_f(\mathsf{x}, v_i)$ nous donne l'ensemble des nœuds qui utilisent x et qui *post-dominent* v_i.

$$\text{post-dominate}_f(\mathsf{x}, v_i) \triangleq \{v_j \in U_f^{\mathsf{x}} \mid v_j \neq v_i \wedge \forall p \in \text{paths}(f, v_i, \text{end}(V_f)),\ v_j \in p\}$$

Remarque 19. On constate que la post-domination peut également s'obtenir en inversant le graphe et en calculant la domination.

On définit alors $\text{usemustreach}_f(\mathsf{x}, v_i)$, similaire à la Définition 24 de defmayreach, qui donne l'ensemble des utilisations de x dans f qui dominent v_j par un chemin sans redéfinition.

Définition 28 (Doit atteindre).

$$\text{usemustreach}_f(\mathsf{x}, v_j) \triangleq \{ v_i \in \text{dominate}_f(\mathsf{x}, v_j) \mid \forall p \in \text{paths}(f, v_i, v_j), p \in \overline{\text{DC}}_f(\mathsf{x}, v_i, v_j)\}$$

Remarque 20. Le principe de propagation de usemustreach est similaire à celui de defmayreach présenté dans la Remarque 18. Contrairement à l'ensemble defmayreach qui va faire l'union des ensembles defmayreach propagés depuis chaque prédécesseur, ici on va au contraire faire une intersection et assurer ainsi la domination.

Pour générer un objectif on procède donc de la manière suivante : lorsqu'on traite l'utilisation d'une variable x au nœud v_j, on vérifie si cette utilisation est équivalente à une utilisation qui la précède. Si oui, on l'ignore, sinon, on crée un objectif.

L'Algorithme 7 nous donne l'algorithme de génération d'objectifs ignorant les éventuels objectifs équivalents. La fonction `Has_equiv_before` nous permet de vérifier pour une utilisation donnée si elle est équivalente à au moins une autre utilisation. On regarde ce que contient l'ensemble renvoyé par $\text{usemustreach}_f(\mathsf{x}, v_j)$. Pour chaque nœud v_k dans cet ensemble, on calcule la post-domination comme vu précédemment. Si v_j post-domine au moins un

Algorithme 7 Création de toutes les paires def-use sans équivalents

```
1 : function Has_equiv_before(f, x, v_j)
2 :     for all v_k ∈ usemustreach_f(x, v_j) do
3 :         if v_j ∈ post-dominate_f(x, v_k) then
4 :             return true
5 :         end if
6 :     end for
7 :     return false
8 : end function
9 :
10 : procedure alldu_EQ_filter(f)
11 :     L ← L_f ∪ F_f
12 :     for all x ∈ L do
13 :         for all v_j ∈ U_f^x do          ▷ Filtre, garde les uses sans équivalents
14 :             if ¬Has_equiv_before(f, x, v_j) then
15 :                 for all v_i ∈ D_f^x do
16 :                     insert_du(f, x, v_i, v_j)
17 :                 end for
18 :             end if
19 :         end for
20 :     end for
21 : end procedure
```

nœud v_k, alors on va ignorer l'utilisation de x au nœud v_j dans la génération d'objectifs, en considérant que les objectifs sur x au nœud v_j seraient équivalents à ceux sur v_k. Si ce n'est pas le cas, alors on va générer un objectif pour chaque définition de la même variable.

> **Exemple 15.** On a vu dans notre exemple de la Figure 4.1, que les nœuds **v4** et **v5** sont équivalents. En exécutant notre génération, **v4** sera dans $\text{usemustreach}_{\texttt{main}}(\texttt{x}, \textbf{v5})$, et de la même manière **v5** sera dans $\text{post-dominate}_{\texttt{main}}(\texttt{x}, \textbf{v4})$. La fonction `Has_equiv_before` renvoie donc $vrai$, on ne génère pas d'objectif pour l'utilisation de x au nœud **v5**.

Cet algorithme est évalué sur plusieurs exemples de code C dans le Chapitre 6 sous le nom M_{Eq}.

Enfin, la combinaison de nos deux techniques nous donne alors l'Algorithme 8. Cet algorithme est évalué sur plusieurs exemples de code C dans le Chapitre 6 sous le nom M_{NA_Eq}.

Algorithme 8 Création de toutes les paires def-use sans non-applicables ou équivalents

```
1 : procedure alldu_NA_EQ_filter(f)
2 :     L ← L_f ∪ F_f
3 :     for all x ∈ L do
4 :         for all v_j ∈ U_f^x do
5 :             if ¬Has_equiv_before(f, x, v_j) then
6 :                 for all v_i ∈ defmayreach_f(x, v_j) do
7 :                     insert_du(f, x, v_i, v_j)
8 :                 end for
9 :             end if
10 :        end for
11 :    end for
12 : end procedure
```

4.2 Analyse statique

La catégorie d'objectifs infaisables est un peu différente des deux autres. Ici on ne s'intéresse pas à l'existence de chemins ou l'équivalence entre plusieurs objectifs, mais à la faisabilité des chemins. Dans notre exemple de la Figure 4.1 se trouve des objectifs polluants que l'on ne peut pas trouver avec notre analyse de flot de données : on les appelle des objectifs *infaisables*. Imaginons qu'on veuille ici générer un objectif entre **v2** et **v4**, pour représenter une paire définition-utilisation entre ces deux nœuds. On s'aperçoit qu'il est impossible d'exécuter à la fois **v2** et **v4**, puisque l'exécution du bloc de la première conditionnelle va modifier notre variable `flip`, rendant la suivante fausse. On ne rentrera donc pas dans le bloc qui contient **v4**. L'analyse de flot de données ne s'intéresse qu'au flot dans le graphe de flot de contrôle et pas aux valeurs en elles-mêmes et ne peut donc pas détecter ce cas particulier. Cependant, en utilisant les méthodes d'analyses statiques présentées dans les Chapitres 1 et 2, on va être capable de détecter certains de ces cas.

Reprenons notre exemple de la Figure 4.1, et insérons la séquence de **v2** vers **v4**. La Figure 4.3 est le résultat obtenu après cette insertion. On souhaite maintenant utiliser l'analyse statique pour prouver que cet objectif n'est pas réalisable, c'est-à-dire qu'on ne peut pas atteindre le label avec `status_x_1 == 1`. Pour ce faire, on va appliquer une transformation de code définie dans l'algorithme Algorithme 9.

On inverse l'expression de chaque label utilisée pour réaliser l'objectif pour obtenir sa négation et on crée un **check** comme présenté dans le Chapitre 3.

```
int main(int cond) {
 int status_x_1 = 0;
 int res = 0;
 int flip = 1;
 v1: int x = 42;
 status_x_1 = 0;
 if (cond) {
  v2: x = x + 1;
  status_x_1 = 1;
  v3: res = x;
  flip = 0;
 }
 if (flip) {
  label(1,status_x_1 == 1);
  v4: res = res + x * x;
  v5: res = res * x;
 }
 return res;
}
```

FIGURE 4.3 : Insertion d'une séquence dans la Figure 4.1

```
int main(int cond) {
  int status_x_1 = 0;
  int res = 0;
  int flip = 1;
  v1: int x = 42;
  status_x_1 = 0;
  if (cond) {
    v2: x = x + 1;
    status_x_1 = 1;
    v3: res = x;
    flip = 0;
  }
  if (flip) {
    check(1,!(status_x_1 == 1));
    v4: res = res + x * x;
    v5: res = res * x;
  }
  return res;
}
```

FIGURE 4.4 : Conversion du `label` de la Figure 4.3 en `check`

Algorithme 9 Remplacer les labels par des checks

```
1 : procedure to_checks(f)
2 :     for all v ∈ V_f do
3 :         newA ← ∅                                  ▷ Création d'un ensemble vide
4 :         for all label(id,e) ∈ annots(v) do        ▷ Pour chaque label
5 :             newA ← newA ∪ {check(id,!e)}          ▷ Crée un check
6 :         end for
7 :         annots(v) ← newA                          ▷ Remplace les annotations
8 :     end for
9 : end procedure
```

On remplace ensuite l'ensemble des labels de chaque nœud par un ensemble de checks. Enfin, on va utiliser les outils d'analyse statique pour essayer de prouver que cette négation est toujours vraie. Si c'est le cas et si on peut le prouver, alors on est certain que notre label est infaisable, puisque sa négation est toujours vraie.

Pour notre exemple de la paire def-use (v_2, v_4) de la Figure 4.4, l'analyse statique peut effectivement prouver que le `check` correspondant est toujours vrai, donc le label est infaisable. Donc la paire def-use (v_2, v_4) est infaisable.

Chapitre 5

Nouveaux critères et génération de tests

Sommaire

Nous avons conceptualisé et formalisé dans les chapitres Chapitres 3 et 4 des techniques pour générer des objectifs de flot de données tout en essayant de filtrer ceux qui sont polluants. Nous avons également décrit dans l'état de l'art (Chapitre 2, Section 2.3) les notions que nous appelons des labels et hyperlabels pour matérialiser des critères de couverture de code.

Dans ce chapitre, nous allons proposer dans la Section 5.1 une méthode de génération de tests pour des critères de flot de données. Cette méthode peut s'appliquer à de nouveaux critères. Nous verrons dans la Section 5.2 un type de critères pas encore mentionné jusqu'à présent, les critères aux limites. Nous présenterons ensuite un nouveau critère qui combine le test aux limites et les

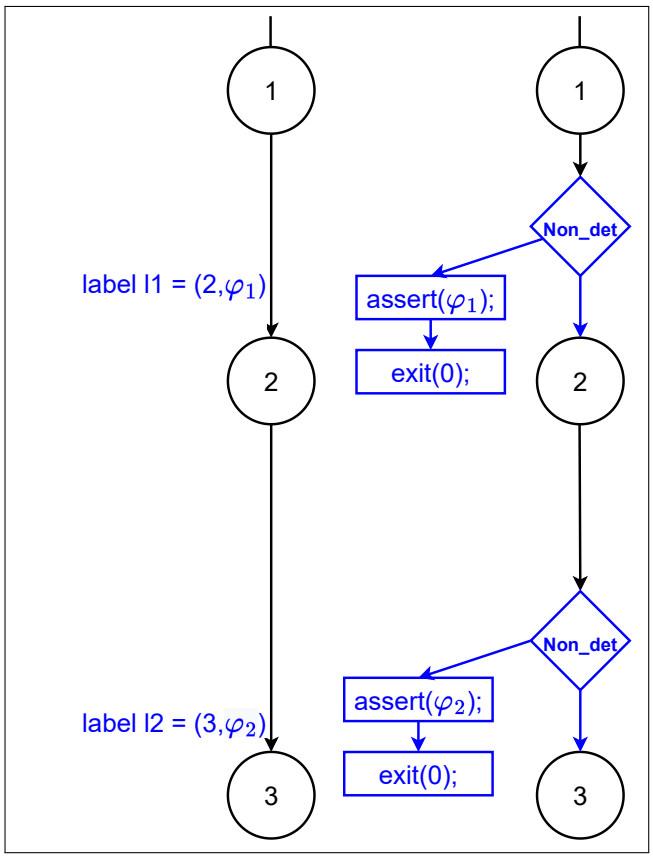


FIGURE 5.1 : Instrumentation compacte de deux labels

critères de flot de données. Enfin, nous introduirons le concept de visibilité dans la Section 5.3 avec une méthode pour l'appliquer aux critères classiques. Ces critères nous permettrons de démontrer, grâce à notre approche de modélisation de séquences lors de la génération de tests, la capacité de prendre en compte différents critères à base de séquences non triviales, ayant des prédicats dans le label de début et/ou dans le label de fin de la séquence.

5.1 Génération de tests pour les critères de flot de données

On a présenté dans le Chapitre 2, Section 2.5.2, une technique de génération de tests pour les labels, dans laquelle ils sont transformés pour créer un choix non déterministe permettant d'accéder au label ou non et où les labels sont ignorés une fois couverts.

Pendant cette thèse, nous avons également étudié la génération de tests

dans l'objectif de couvrir les critères de flot de données. Notre formalisation des paires def-use (introduite dans la Section 3.3) présente une forme de séquence utilisant une variable de statut permettant de traquer pendant l'exécution l'état de la séquence, si sa contrainte est maintenue ou non. Cette forme n'a pas été choisie par hasard. En effet, même s'il existe déjà des techniques permettant de générer des tests pour les labels classiques, les séquences utilisent en revanche plusieurs labels reliés entre eux avec une contrainte sur le chemin. Un problème apparaît donc avec l'optimisation d'instrumentation compacte, comme montré par la Figure 5.1.

Si on utilise des labels classiques pour représenter notre séquence comme dans le Chapitre 2, on obtient ce résultat lorsqu'on utilise notre technique de génération de tests. La contrainte permettant de couvrir notre premier label est jetée une fois qu'il est couvert, puis on passe à la suite pour essayer de couvrir le deuxième label. Mais pour couvrir une séquence, il faut que les deux labels soient couverts par *la même* exécution de la fonction, et donc on a besoin de garder le prédicat du premier label lorsqu'on essaie de couvrir le second. L'optimisation avec un choix non déterministe ne permet pas de garantir de couvrir les deux labels d'une séquence par le même test. Nous voyons que la technique de génération de tests pour les labels ne convient pas pour le support des séquences avec deux labels. C'est pour cette raison que nous avons introduit une version des séquences dans le Chapitre 3 qui n'utilise qu'un seul label, avec une variable de statut qui servira à stocker le résultat du premier label.

Cependant, un nouveau problème apparaît lorsqu'on souhaite ajouter un prédicat de départ φ_s, c'est-à-dire, un prédicat à satisfaire au point du premier label d'une séquence. Dans l'approche proposée dans la Section 3.3 (par les Algorithmes 3 et 4, où start représente le prédicat φ_s), ce prédicat est affecté à une variable de statut. Cette variable de statut sera intégrée à la condition du deuxième label de la séquence et sera donc testée au point de ce label. Cela n'est pas le plus efficace. En effet, avec l'ajout d'un prédicat de départ affecté à une variable de statut, un nouveau problème est apparu. Avoir simplement un prédicat affecté dans une variable va être traité, lors de la génération de tests, comme une décision, et transformé de la même façon que le prédicat du label de la Figure 2.11 avec l'instrumentation directe, comme le montre l'exemple de la Figure 5.2.

Dans cette figure, au lieu d'affecter `1` à la variable de statut d'une séquence au point de définition, on ajoute un prédicat (ici, `i == INT_MAX`). Ce dernier représente φ_s et donc le prédicat du label de départ de la séquence.

Après la transformation pour la génération de tests (illustrée par la partie droite de la Figure 5.2), une explosion de complexité similaire à celle des

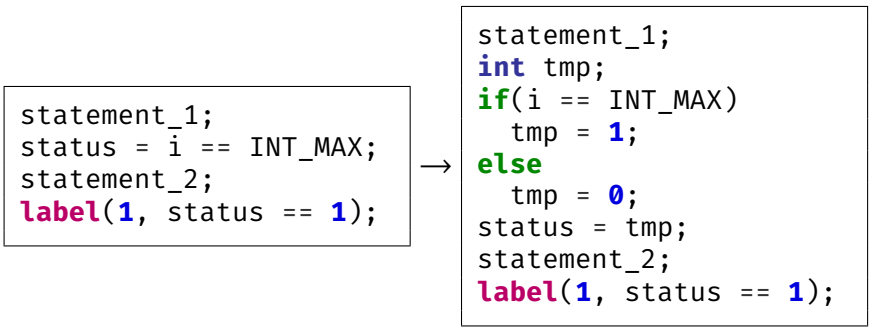


FIGURE 5.2 : Instrumentation pour une évaluation immédiate d'un prédicat

```
statement_1;
status = 1;
i_delay = i;
statement_2;
label(1, status == 1 && i_delay == INT_MAX);
```

FIGURE 5.3 : Instrumentation pour une évaluation retardée d'un prédicat

labels évitée grâce à l'instrumentation compacte (voir la Section 2.5.2) réapparaît, puisque le prédicat n'est pas directement instrumenté comme un label avec notre forme de séquence. Il y a donc deux solutions : soit réintroduire les labels de début de séquence dans la modélisation, en plus des labels de fin et des variables de statut, ce qui impliquerait de recoder en grande partie les techniques vues jusqu'à présent (détection de polluants, génération de tests pour les labels), soit trouver une alternative à cette forme. C'est cette seconde approche que nous avons suivie, en proposant d'utiliser l'évaluation *retardée* du prédicat.

Contrairement à l'exemple de la Figure 5.2 que nous qualifions d'évaluation *immédiate*, et dans laquelle le prédicat est évalué sur place à l'aide d'un **if**, l'évaluation retardée va repousser le traitement du prédicat au niveau de la fin de la séquence, c'est-à-dire cette fois dans le label. Le principe de cette approche est le suivant :

— sauvegarder (dans des variables auxiliaires) les valeurs des variables impliquées dans le prédicat φ_s au point de début de la séquence où ce prédicat doit être vérifié, sans l'évaluer à cet endroit ;

— évaluer ce prédicat à l'aide des valeurs sauvegardées au point de fin de la séquence, dans la condition du label inséré à cet endroit.

La Figure 5.3 présente cette seconde approche sur le même exemple.

Pour comparer les deux approches sur notre exemple, la Figure 5.4 présente les graphes de flot de contrôle correspondant aux différentes instrumentations

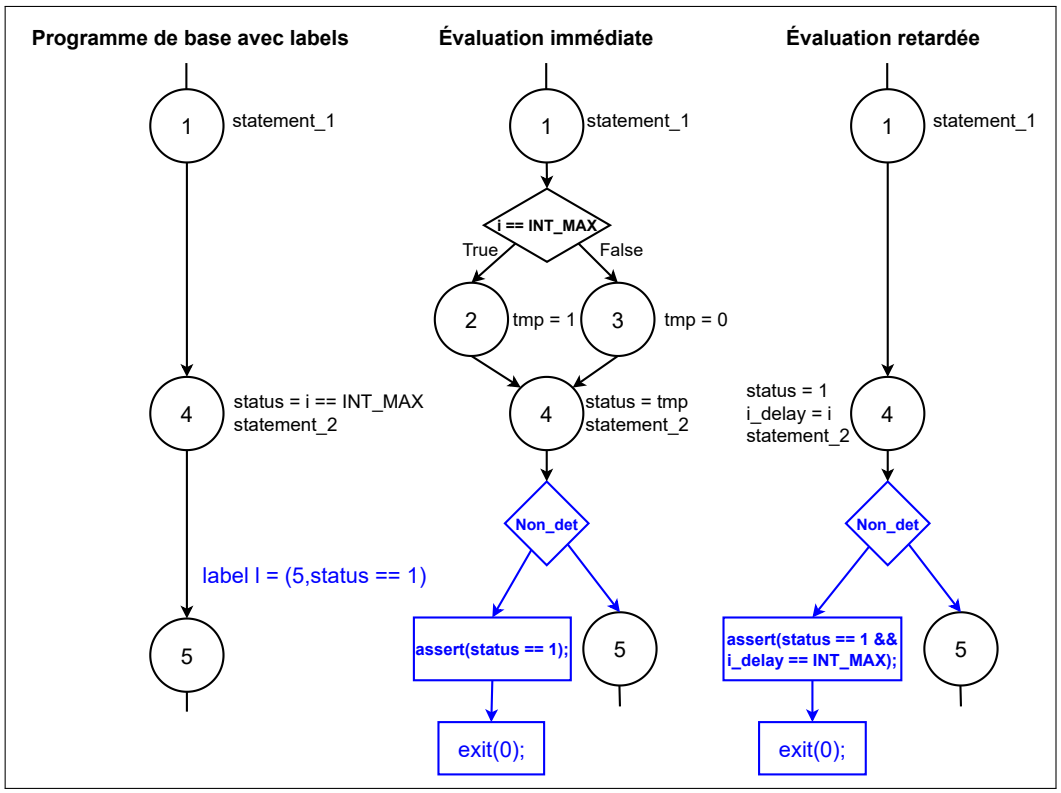


FIGURE 5.4 : Comparaison des deux approches

des Figures 5.2 et 5.3. Dans le cas où l'évaluation est immédiate, on duplique le nombre de chemins en passant par ce prédicat. De plus, la contrainte issue du prédicat φ_s est gardée dans le prédicat du chemin (et par conséquent, dans le store des contraintes à résoudre) lors de la poursuite de l'exploration des chemins du programme, même quand le choix non déterministe du label indique de poursuivre cet exploration en ignorant le label. Cela rend cette exploration moins efficace, notamment, pour couvrir d'autres objectifs que la suite du programme peut contenir. L'évaluation retardée quant à elle permet d'évaluer le prédicat φ_s en même temps que le prédicat de fin de séquence φ_e, comme mentionné dans le Chapitre 3, et donc de l'ignorer, comme pour les labels classiques, dans la suite de l'exploration.

Cette approche nous permet donc en théorie d'éviter l'explosion des chemins lorsqu'on utilise des séquences plus complexes que les critères de flot de données classiques. En particulier, cette instrumentation s'est avérée utile pour de nouveaux critères auxquels nous sommes intéressés et que nous décrivons dans la section suivante.

Pour plus de lisibilité, dans les exemples et les algorithmes des sections sui-

vantes de ce chapitre, nous allons utiliser l'instrumentation pour l'évaluation immédiate des séquences, car elle est un peu moins verbeuse que l'instrumentation pour l'évaluation retardée. En pratique, nous avons implémenté dans les outils et évalué les deux approches, comme nous le présenterons dans le Chapitre 6.

5.2 Test aux limites pour les séquences

Lorsqu'on teste un logiciel, il est en général impossible de tester toutes les entrées possibles pour une fonction donnée et c'est pour cela qu'on fait appel à des techniques plus avancées, utilisant des critères de couverture, afin de choisir ce que l'on considère comme étant pertinent et suffisant à tester.

Il est donc possible, par exemple, de ne tester que certaines valeurs en particulier que l'on juge plus intéressantes. En informatique, les nombres entiers sont généralement représentés en utilisant des types qui ont une taille spécifique qui peut dépendre de la machine utilisée, contrairement aux entiers mathématiques qui sont infinis. Par exemple pour les nombres entiers, le type `int` pour entier (integer en anglais) sur 32 bits représente les nombres allant de -2147483648 (entier minimum) à 2147483647 (entier maximum). Mais que ce passe-t-il si on essaie de dépasser ces limites ? Par exemple additionner 1 à l'entier maximal. C'est ce qu'on appelle en C un comportement non défini : le comportement dépend de l'architecture de la machine qui exécute le programme et du compilateur utilisé. On veut donc généralement empêcher l'apparition de ces comportements, car ils peuvent être source de bugs. Il pourrait alors être intéressant d'essayer de trouver les endroits où ce type de comportement peut se produire, pour essayer de s'en protéger.

C'est ici qu'intervient le *test aux limites*. On va essayer de créer des tests jouant avec les valeurs limites des variables (en fonction du type) pour vérifier que le programme se comporte correctement. C'est justifié par le fait que dans beaucoup de calculs, le dépassement arithmétique se produit pour les valeurs aux limites. Les critères de test aux limites sont des critères utilisant des objectifs allant dans ce sens. Dans le cadre de cette thèse, nous avons conceptualisé de tels critères. Avant de voir comment utiliser ces limites avec les séquences, nous allons introduire un nouveau critère de couverture.

5.2.1 Test aux limites des entrées et sorties

Input Output Boundary Coverage (**IOB**) est un critère plutôt simple : on crée des objectifs de test pour chaque entrée et sortie des fonctions aux limites des

Algorithme 10 Génération des objectifs de test pour le critère **IOB**

```
1 : procedure gen_min_max(v, e)
2 :     t ⟵ get_type(e)                    ▷ Retourne le type de l'expression e
3 :     min ⟵ get_min(t)                   ▷ Retourne le minimum du type t
4 :     max ⟵ get_max(t)                   ▷ Retourne le maximum du type t
5 :     id1 ⟵ new_id()
6 :     id2 ⟵ new_id()
7 :     add_label(v, id1, e == min)
8 :     add_label(v, id2, e == max)
9 : end procedure
10 : procedure gen_IOB(f)
11 :     for all f ∈ 𝔽 do
12 :         for all x ∈ F_f do                ▷ Pour chaque entrée de f
13 :             gen_min_max(begin(V_f), x)
14 :         end for
15 :         for all e ∈ exprs(end(V_f)) do    ▷ exprs renvoie ici un singleton
16 :             gen_min_max(end(V_f), e)
17 :         end for
18 :     end for
19 : end procedure
```

types correspondants. L'objectif est de vérifier le comportement desdites fonctions quand leurs paramètres sont égaux aux valeurs maximales et minimales possibles. Si on reprend le type entier sur 32 bits, alors on aura deux objectifs : un pour l'entier maximum et un pour l'entier minimum. On peut ensuite généraliser cela à toutes les fonctions f dans $\mathbb{F}$ de notre programme P.

Avec l'Algorithme 10, on crée nos objectifs de test, des labels, pour le critère **IOB**. Pour chaque fonction, on va générer deux objectifs par paramètre d'entrée pour comparer sa valeur avec celle des bornes de son type. De la même manière, on récupère la sortie de la fonction et on crée un objectif pour tester sa valeur aux bornes de son type. Dans notre petit langage, toutes les expressions sont de type entier et toutes les fonctions retournent un entier. Cet algorithme présente une idée générale, qui peut s'adapter à l'ajout de nouveaux types numériques.

La Figure 5.5 illustre un exemple d'annotation pour ce critère pour un programme C. On crée six labels pour les entrées (deux pour chaque paramètre) en fonction de leur type et deux labels pour l'unique sortie.

Ce critère est plutôt simple et permet donc lors de la génération de tests de créer des cas de test appelant une fonction avec les valeurs limites dans le

```c
int f(char a, int b, unsigned short c){
  b = a + b + c
  return b;
}
```

↓

```c
int f(char a, int b, unsigned short c){
  label(1,a == -128);
  label(2,a ==  127);
  label(3,b ==  2147483647);
  label(4,b == -2147483648);
  label(5,c == 0);
  label(6,c == 65535);
  b = a + b + c
  label(7,b ==  2147483647);
  label(8,b == -2147483648);
  return b;
}
```

FIGURE 5.5 : Critère IOB sur un exemple C simple

but de couvrir les labels. Cependant, on peut vouloir tester encore plus spécifiquement les limites pour trouver les comportements non définis et c'est ici qu'interviennent les critères de flot de données.

5.2.2 Test aux limites des critères de flot de données

Maintenant qu'on a vu comment fonctionnaient les critères de test aux limites, on va proposer un nouveau type de critère qui va combiner le test aux limites avec les critères de flot de données, et en particulier les séquences. On va considérer de nouveau les paires def-use qui sont une brique essentielle des critères de flot de données. On a vu dans le Chapitre 3 que la définition de φ_s pour les paires def-uses classiques était triviale, mais ici on va voir apparaître un cas plus complexe : φ_s sera utilisé pour imposer des valeurs particulières, à savoir, des valeurs égales aux minimums ou maximums de leurs types. Autrement dit, on impose de tester des paires def-use quand la valeur de définition est aux limites.

La Figure 5.6 illustre ce que donne un critère de flot de données tel que présenté dans les chapitres précédents, en ajoutant des contraintes imposant les valeurs aux limites des variables. On remarque dans un premier temps qu'aucune séquence ne va de **v3** vers lui-même, contrairement à la Figure 3.5 du Chapitre 3. Nous avons utilisé nos différentes techniques, avec l'Algorithme 8, pour

```
int f(int cond){
v1: int x = 42;
v2: if(cond){
  v3: x = x + 1;
}
v4: return x;
}
```

```
int f(int cond) {
int status_cond_1 = cond==(-2147483647-1);
int status_cond_2 = cond==2147483647;
int status_x_3 = 0;
int status_x_4 = 0;
int status_x_5 = 0;
int status_x_6 = 0;
int status_x_7 = 0;
int status_x_8 = 0;
v1: int x = 42;
status_x_3 = x == (-2147483647-1);
status_x_4 = x == 2147483647;
status_x_7 = x == (-2147483647-1);
status_x_8 = x == 2147483647;
label(1,status_cond_1 == 1);
label(2,status_cond_2 == 1);
v2: if (cond) {
  label(7,status_x_7 == 1);
  label(8,status_x_8 == 1);
  v3: x ++;
  status_x_3 = 0;
  status_x_4 = 0;
  status_x_7 = 0;
  status_x_8 = 0;
  status_x_5 = x == (-2147483647-1);
  status_x_6 = x == 2147483647;
}
label(3,status_x_3 == 1);
label(4,status_x_4 == 1);
label(5,status_x_5 == 1);
label(6,status_x_6 == 1);
v4: return x;
}
```

FIGURE 5.6 : Annotation d'une fonction avec des séquences de flot de données aux limites

supprimer les objectifs polluants et notamment ici un objectif non-applicable. En ignorant cette séquence, on avait dans l'exemple précédent quatre objectifs, qui ont été dédoublés et transformés en huit objectifs. Plus généralement, pour chaque séquence classique, on introduit deux séquences : une pour le maximum et une pour le minimum du type des variables. On compare notre variable à la borne concernée et le résultat vaudra donc `1` si la variable est égale à la borne, `0` sinon. Pour couvrir une séquence, il faut donc se situer sur une limite du type de la variable.

Algorithme 11 Création de toutes les paires def-use avec limites

```
1 : procedure alldu_bound_NA_EQ_filter(f)
2 :     L ← L_f ∪ F_f
3 :     for all x ∈ L do
4 :         for all v_j ∈ U_f^x do
5 :             if ¬Has_equiv_before (f, x, v_j) then
6 :                 for all v_i ∈ defmayreach_f(x, v_j) do
7 :                     t ← get_type(x)
8 :                     min ← get_min(t)
9 :                     max ← get_max(t)
10 :                    insert_du_bound(f, x, v_i, v_j, min)
11 :                    insert_du_bound(f, x, v_i, v_j, max)
12 :                end for
13 :            end if
14 :        end for
15 :    end for
16 : end procedure
```

Pour ce critère, on doit donc adapter nos Algorithmes 4 et 8 des Chapitres 3 et 4, ce qui nous donne les Algorithmes 11 et 12.

Cette fois-ci, au lieu de simplement insérer une paire def-use, on va, comme pour le critère **IOB**, récupérer le type de notre expression (ici la variable x) et créer une séquence pour chaque borne du type en question. La limite sera alors ajoutée au prédicat de départ φ_s introduit dans le Chapitre 3 (et représenté par la variable start dans les algorithmes) pour former la contrainte que doit respecter cette séquence pour être couverte. Chaque critère de flot de données portant sur des paires def-use (tels que **All-defs** ou **All-uses**) peut donc être adapté de cette façon pour le rendre plus précis. Grâce à notre méthode de génération des tests, on sera donc également capable de générer de tests spécifiques à la couverture des paires def-use aux limites.

5.3 Visibilité des objectifs de test dans les sorties

La visibilité, dans le domaine du test et des critères de couverture de code, représente l'observation de l'impact de la couverture d'un des objectifs de test sur la sortie d'une fonction sous test. Est-ce que la couverture d'un test est *visible* sur la sortie de la fonction, c'est-à-dire, différente par rapport aux sorties des autres tests ? Est-il possible de trouver une sortie différente pour chaque

Algorithme 12 Insertion d'une paire def-use avec un test de limite

```
1 : procedure insert_du_bound(f, x, v_i, v_j, bound)
2 :     id ← new_id()
3 :     status ← make_fresh_var(f, id, x)
4 :     start ← x == bound
5 :     cstr ← status == 1
6 :     end ← 1
7 :     insert_cstr(f, x, status, v_i, start)
8 :     add_label(v_j, id, cstr && end)
9 : end procedure
```

objectif de test ?

Suite à nos discussions, dans le cadre du projet ANR SATOCROSS, avec les ingénieurs du partenaire industriel MERCE, et à la demande de ce partenaire, nous avons défini un nouveau type d'objectifs de test ayant pour but de mettre en avant cette visibilité. En pratique, l'ingénieur vérification souhaite pouvoir sélectionner des cas de test pour les différents objectifs ayant des sorties différentes quand cela est possible. Cela peut être impossible dans le cas où le nombre de sorties possibles est inférieur au nombre d'objectifs par exemple. Nous sommes partis de critères de couverture existants, comme le critère **DC**, que nous modifions dans le but de mettre en avant cette visibilité. Notre méthode consiste en 5 étapes principales :

Étape 1 : Annotation des objectifs

Étape 2 : Création des séquences

Étape 3 : Génération de tests

Étape 4 : Mise à jour des objectifs

Étape 5 : Vérification des conditions d'arrêts

Les étapes 3 à 5 sont répétées jusqu'à satisfaire une condition d'arrêt que nous détaillerons plus tard.

5.3.1 Étape 1 : Annotation des objectifs

Cette étape reprend simplement ce que nous avons présenté dans la Section 2.3 du Chapitre 2. On veut annoter notre programme en ajoutant des objectifs correspondant à un critère de couverture de code fixé par l'utilisateur, par exemple le critère **DC**. La Figure 5.7 illustre cette annotation sur une fonction `f`. Cette fonction compte le nombre de bits à 1 parmi les trois premiers bits d'un nombre entier. On va donc tester la valeur de chacun des bits une par une

```c
int f(int x){
  int res = 0;
  if(x & 1) res++;
  if(x & 2) res++;
  if(x & 4) res++;
  return res;
}
```

(a) Code d'origine

```c
int f(int x) {
  int res = 0;
  //l1 :   x & 1
  //l2 : !(x & 1)
  if (x & 1) res++;
  //l3 :   x & 2
  //l4 : !(x & 2)
  if (x & 2) res++;
  //l5 :   x & 4
  //l6 : !(x & 4)
  if (x & 4) res++;
  return res;
}
```

$\ell_1 \triangleq (loc_1,$ `x & 1`$)$
$\ell_2 \triangleq (loc_2,$ `!(x & 1)`$)$
$\ell_3 \triangleq (loc_3,$ `x & 2`$)$
$\ell_4 \triangleq (loc_4,$ `!(x & 2)`$)$
$\ell_5 \triangleq (loc_5,$ `x & 4`$)$
$\ell_6 \triangleq (loc_6,$ `!(x & 4)`$)$

(b) Code obtenu après annotation et labels générés

FIGURE 5.7 : Étape 1 : Annotation des objectifs pour le critère **DC**

et incrémenter la valeur de retour. Pour cette fonction, on a trois conditions et donc deux objectifs **DC** par condition pour un total de six labels. On rappelle que les positions loc_i correspondent bien aux lignes dans le code correspondant aux trois décisions, même si elles ne sont pas détaillées ici.

> **Remarque 21.** Comme présenté dans notre formalisation (Chapitre 3), cette fonction ne contient qu'une seule instruction de retour `return res`, ce qui aura de l'importance pour les prochaines étapes. On peut toujours procéder à une transformation simple, à base de sauts (`goto` en C), pour se ramener à une fonction avec une seule instruction de retour.

5.3.2 Étape 2 : Création des séquences

On va, dans cette étape, ajouter pour chaque label ℓ_i de notre critère un nouveau label ℓ'_i, juste avant la sortie de la fonction dont le prédicat sera vrai. Ensuite, on crée une séquence allant du premier label vers celui que nous ve-

$\ell_1 \triangleq (loc_1,$ `x & 1`$)$
$\ell_2 \triangleq (loc_2,$ `!(x & 1)`$)$
$\ell_3 \triangleq (loc_3,$ `x & 2`$)$
$\ell_4 \triangleq (loc_4,$ `!(x & 2)`$)$
$\ell_5 \triangleq (loc_5,$ `x & 4`$)$
$\ell_6 \triangleq (loc_6,$ `!(x & 4)`$)$
$\ell'_1 \triangleq (loc_7,$ `true`$)$
$\ell'_2 \triangleq (loc_8,$ `true`$)$
$\ell'_3 \triangleq (loc_9,$ `true`$)$
$\ell'_4 \triangleq (loc_{10},$ `true`$)$
$\ell'_5 \triangleq (loc_{11},$ `true`$)$
$\ell'_6 \triangleq (loc_{12},$ `true`$)$
$h_1 \triangleq \ell_1 \xrightarrow{\texttt{true}} \ell'_1$
$h_2 \triangleq \ell_2 \xrightarrow{\texttt{true}} \ell'_2$
$h_3 \triangleq \ell_3 \xrightarrow{\texttt{true}} \ell'_3$
$h_4 \triangleq \ell_4 \xrightarrow{\texttt{true}} \ell'_4$
$h_5 \triangleq \ell_5 \xrightarrow{\texttt{true}} \ell'_5$
$h_6 \triangleq \ell_6 \xrightarrow{\texttt{true}} \ell'_6$

```
int f(int x) {
  int res = 0;
  //l1 :   x & 1
  //l2 : !(x & 1)
  if (x & 1) res++;
  //l3 :   x & 2
  //l4 : !(x & 2)
  if (x & 2) res++;
  //l5 :   x & 4
  //l6 : !(x & 4)
  if (x & 4) res++;
  //l1' : true
  //l2' : true
  //l3' : true
  //l4' : true
  //l5' : true
  //l6' : true
  return res;
}
```

FIGURE 5.8 : Étape 2 : création des séquences à partir de **DC**

nons de créer $\ell_i \xrightarrow{true} \ell'_i$. C'est pour cette étape que l'unicité du nœud de sortie est importante, car on veut relier chaque label à la sortie de la fonction et avoir plusieurs sorties complexifierait beaucoup cette procédure. La Figure 5.8 illustre cette étape avec $\varphi =$ `true` comme contrainte sur nos séquences, donc pas de contrainte et le prédicat de fin de nos labels également toujours vrai pour le moment.

5.3.3 Étape 3 : Génération de tests

Une fois nos séquences créées, on va utiliser la génération de tests dans le but de les couvrir conformément à ce que nous avons présenté dans la Section 5.1. Après une session de tests, on peut obtenir par exemple les résultats suivants :

entrée : x = 1 / sortie : res = 1 / couvre : h_1, h_4 et h_6

entrée : x = 6 / sortie : res = 2 / couvre : h_2, h_3 et h_5

> **Remarque 22.** Pour rappel, ici la génération essaie de créer des tests jusqu'à ce que les labels soient tous couverts. Lorsque toutes les séquences (et donc labels) sont couvertes, on arrête la génération, ce qui met fin à cette session de tests.

Ces tests couvrent bien les six séquences, on retient donc que les séquences h_1, h_4 et h_6 peuvent amener à retourner la valeur 1 et h_2, h_3 et h_5 à la sortie 2. On enregistre, pour chaque séquence, toutes les sorties obtenues par les tests qu'on vient de générer qui couvrent cette séquence. Le déroulement de cette étape et de la suite de notre algorithme est synthétisé dans la Figure 5.9.

> **Remarque 23.** On appelle sortie ici tout ce qui peut avoir un effet sur le reste du programme hors de cette fonction. La valeur de retour en fait évidemment partie, mais également les variables globales qui peuvent être modifiées par la fonction. Il n'y en a pas dans cet exemple pour le simplifier, mais la méthode s'applique également à ces variables.

5.3.4 Étape 4 : Mise à jour des objectifs

On va alors venir modifier nos objectifs, en particulier le label final de ces séquences (correspondant à φ_e dans notre formalisation), pour inclure le résultat de l'**étape 3** afin de l'éviter à l'avenir, c'est-à-dire que la sortie doit être différente de 1 lorsqu'on essaie de couvrir les séquences couvertes précédemment. Avec cette **étape 4**, on modifie donc les labels ℓ'_1, ℓ'_4 et ℓ'_6 correspondant aux séquences h_1, h_4 et h_6 et les labels ℓ'_2, ℓ'_3 et ℓ'_5 pour les séquences h_2, h_3 et h_5 comme suit :

$$\begin{aligned}
\ell'_1 &\triangleq (loc_7, \texttt{ res != 1})\\
\ell'_2 &\triangleq (loc_8, \texttt{ res != 2})\\
\ell'_3 &\triangleq (loc_9, \texttt{ res != 2})\\
\ell'_4 &\triangleq (loc_{10}, \texttt{res != 1})\\
\ell'_5 &\triangleq (loc_{11}, \texttt{res != 2})\\
\ell'_6 &\triangleq (loc_{12}, \texttt{res != 1})
\end{aligned}$$

En ajoutant cette contrainte, on demande en fait à la génération de tests d'essayer de couvrir les séquences en évitant les sorties produites par les tests qui les ont déjà couvertes. Si on essaie de couvrir à nouveau le label ℓ'_1 de la séquence h_1, on le fera en évitant la valeur de sortie 1.

5.3.5 Étape 5 : Vérification des conditions d'arrêts

Après avoir réalisé nos quatre premières étapes, on va alors déterminer si on doit poursuivre notre algorithme ou s'arrêter dès que l'on satisfait l'une des trois *conditions d'arrêts* :

1. Tous les labels sont infaisables ;
2. La génération de tests n'arrive plus à couvrir les labels ;
3. On atteint une limite d'itération pour les **étapes 3** à **5** fixées préalablement par l'utilisateur.

Si on ne satisfait aucune condition d'arrêt, on reprend l'algorithme à l'**étape 3** et on recommence.

On dispose de plusieurs manières pour déterminer qu'on a testé toutes les sorties possibles qui couvrent nos objectifs et arrêter notre algorithme.

La première consiste à utiliser les travaux présentés dans le Chapitre 4. Après avoir modifié nos labels avec l'**étape 4**, on vérifie s'ils sont devenus infaisables. On marque ainsi tous ceux qui le sont (s'il y en a). Si tous les labels sont marqués comme étant infaisables, il devient alors inutile de continuer et on satisfait notre première condition d'arrêt. Le défaut de cette approche étant évidemment qu'il peut être compliqué dans certains cas de prouver l'infaisabilité d'un objectif.

> **Remarque 24.** Lorsqu'on effectue l'**étape 3**, la génération de tests ignore tous les labels étant marqués comme infaisables, car il est inutile de dépenser des ressources pour essayer de les couvrir si on sait que c'est impossible.

Cependant, cette détection d'infaisable est en réalité facultative. Elle peut permettre de gagner du temps sur la génération de tests dans la suite en ignorant les labels infaisables et donc d'améliorer les performances de notre approche, mais lorsqu'un objectif est infaisable, la génération de tests n'arrivera simplement pas à le couvrir. Elle va tout faire pour essayer, en vain, en testant tous les chemins possibles ce qui peut fortement impacter les performances s'il y a beaucoup de chemins à explorer, mais à la fin la génération terminera sans avoir couvert cet objectif. Par conséquent, si tous les objectifs sont infaisables, la génération s'arrêtera sans avoir créé un seul test couvrant nos séquences.

On peut donc obtenir un résultat similaire grâce au deuxième critère d'arrêt, c'est-à-dire s'arrêter quand tous les labels sont infaisables, sans pour autant utiliser la détection d'objectifs polluants. N'ayant pas de tests à la fin de l'**étape**

3, l'**étape 4** n'apportera aucune modification et on considérera alors dans l'**étape 5** qu'on a atteint un point fixe grâce à notre deuxième condition d'arrêt.

Enfin, l'utilisateur peut lui-même définir une limite arbitraire sur le nombre d'itérations à effectuer à partir de laquelle il arrête l'algorithme.

On peut résumer l'**étape 5** de la manière suivante :

— Si l'**étape 3** n'a pas généré de tests, alors on s'arrête ;

— Sinon, si on a atteint la limite spécifiée par l'utilisateur, alors on s'arrête ;

— Sinon, si tous les labels sont infaisables après l'**étape 4**, alors on s'arrête ;

— Sinon on reprend à l'**étape 3**.

Si on répète nos 3 étapes, on peut par exemple obtenir les résultats de la Figure 5.9. Le symbole ✗ indique que l'objectif correspondant est prouvé infaisable.

On commence avec la première ligne, obtenue à l'issue de l'**étape 2** et nos labels toujours vrais. On applique ensuite à tour de rôle les **étapes 3**, **4** et **5**. La première session nous génère deux tests permettant de couvrir tous les labels. L'entrée avec x valant 1 couvre les séquences h_1, h_4 et h_6 et donc les labels ℓ'_1, ℓ'_4 et ℓ'_6, avec comme sortie de fonction la valeur 1. De même, l'entrée avec x valant 6 couvre h_2, h_3 et h_5 et les labels ℓ'_2, ℓ'_3 et ℓ'_5, avec comme sortie de fonction la valeur 2.

En partant de ce résultat, on modifie nos six labels ℓ'_i pour empêcher la génération de tests de couvrir ces labels avec des tests donnant les mêmes valeurs de retour. On a généré des tests à l'**étape 3** de la première itération et tous les labels ne sont pas infaisables après l'**étape 4** de cette même itération, donc l'**étape 5** n'arrête pas l'algorithme et on recommence avec la génération de tests, cette fois-ci avec les entrées 5, 2, 4 et 0. On procède de la même façon que pour les labels de l'étape précédente, créant ainsi des conjonctions pour interdire toutes les sorties obtenues jusqu'à présent.

Cet exemple contient au total quatre sorties différentes, allant de 0 à 3. La condition d'arrêt sera donc ici soit de trouver des sorties uniques à chaque séquence, soit de tester les quatre sorties possibles, soit de s'arrêter quand l'utilisateur le souhaite. Comme cet exemple est relativement court, on va donc pouvoir tester raisonnablement toutes les solutions.

Itér.	Étape 2
0	$\ell'_1 \triangleq (loc_7,$ `true`$)$ $\ell'_2 \triangleq (loc_8,$ `true`$)$ $\ell'_3 \triangleq (loc_9,$ `true`$)$ $\ell'_4 \triangleq (loc_{10},$ `true`$)$ $\ell'_5 \triangleq (loc_{11},$ `true`$)$ $\ell'_6 \triangleq (loc_{12},$ `true`$)$

Itér.	Étape 3					Étape 4	Étape 5
1	entrée	x = 1	x = 6			$\ell'_1 \triangleq (loc_7,$ `res!=1`$)$ $\ell'_2 \triangleq (loc_8,$ `res!=2`$)$ $\ell'_3 \triangleq (loc_9,$ `res!=2`$)$ $\ell'_4 \triangleq (loc_{10},$ `res!=1`$)$ $\ell'_5 \triangleq (loc_{11},$ `res!=2`$)$ $\ell'_6 \triangleq (loc_{12},$ `res!=1`$)$	
	sortie	res = 1	res = 2				
	couvre	h_1, h_4, h_6	h_2, h_3, h_5				
2	entrée	x = 5	x = 2	x = 4	x = 0	$\ell'_1 \triangleq (loc_7,$ `res!=1 && res!=2`$)$ $\ell'_2 \triangleq (loc_8,$ `res!=2 && res!=1 && res!=0`$)$ $\ell'_3 \triangleq (loc_9,$ `res!=2 && res!=1`$)$ $\ell'_4 \triangleq (loc_{10},$ `res!=1 && res!=2 && res!=0`$)$ $\ell'_5 \triangleq (loc_{11},$ `res!=2 && res!=1`$)$ $\ell'_6 \triangleq (loc_{12},$ `res!=1 && res!=0`$)$	✗ ✗
	sortie	res = 2	res = 1	res = 1	res = 0		
	couvre	h_1, h_4	h_2, h_3	h_2, h_5	h_2, h_4, h_6		
3	entrée	x = 7	x = 3			$\ell'_1 \triangleq (loc_7,$ `res!=1 && res!=2 && res!=3`$)$ $\ell'_2 \triangleq (loc_8,$ `res!=2 && res!=1 && res!=0`$)$ $\ell'_3 \triangleq (loc_9,$ `res!=2 && res!=1 && res!=3`$)$ $\ell'_4 \triangleq (loc_{10},$ `res!=1 && res!=2 && res!=0`$)$ $\ell'_5 \triangleq (loc_{11},$ `res!=2 && res!=1 && res!=3`$)$ $\ell'_6 \triangleq (loc_{12},$ `res!=1 && res!=0 && res!=2`$)$	✗ ✗ ✗ ✗ ✗ ✗
	sortie	res = 3	res = 2				
	couvre	h_1, h_3, h_5	h_6				

FIGURE 5.9 : Déroulement des étapes 3 à 5 à partir de l'exemple de la Figure 5.8

À la fin on obtient le résultat suivant :

$$\ell'_1 \triangleq (loc_7,\ \texttt{res != 1 \&\& res != 2 \&\& res != 3})$$
$$\ell'_2 \triangleq (loc_8,\ \texttt{res != 2 \&\& res != 1 \&\& res != 0})$$
$$\ell'_3 \triangleq (loc_9,\ \texttt{res != 2 \&\& res != 1 \&\& res != 3})$$
$$\ell'_4 \triangleq (loc_{10},\ \texttt{res != 1 \&\& res != 2 \&\& res != 0})$$
$$\ell'_5 \triangleq (loc_{11},\ \texttt{res != 2 \&\& res != 1 \&\& res != 3})$$
$$\ell'_6 \triangleq (loc_{12},\ \texttt{res != 1 \&\& res != 0 \&\& res != 2})$$

Ici, la détection d'infaisables permet de mettre fin à l'algorithme après sa troisième itération avec la première condition d'arrêt : tous les objectifs ont été prouvés comme étant infaisables. Si nous n'avions pas réussi à prouver cette infaisabilité, l'itération suivante de l'algorithme n'aurait produit aucun test et la méthode s'arrêterait de toute façon avec la deuxième condition d'arrêt.

5.3.6 Filtrage des tests

Une fois que la suite de tests est générée, il faut prendre une décision : est-ce qu'on veut tout garder, ou seulement certains tests ? Avec quelle stratégie veut-on sélectionner des tests ?

Une stratégie que nous avons envisagée suite aux discussions avec MERCE est de couvrir tous les ℓ_i et en même temps maximiser la visibilité, c'est-à-dire, garder un test par objectif avec une sortie différente autant que possible, ou si impossible, minimiser le nombre de sorties répétées entre les différents tests.

Dans notre exemple, à partir des huit tests générés, on pourrait alors garder les tests suivants :

— x = 1 pour ℓ_1 (avec res = 1),

— x = 2 pour ℓ_2 (avec res = 1),

— x = 7 pour ℓ_3 (avec res = 3),

— x = 5 pour ℓ_4 (avec res = 2),

— x = 6 pour ℓ_5 (avec res = 2),

— x = 0 pour ℓ_6 (avec res = 0),

afin de couvrir tous les ℓ_i et en même temps maximiser la visibilité. Par rapport aux résultats d'une seule session de génération (par exemple, à la première itération), on constate que la visibilité est fortement améliorée. Nous avons en effet deux objectifs (ℓ_3 et ℓ_6) couverts avec des sorties uniques, et parmi les autres au plus deux répétitions de sorties. Pour comparer, on voit que dans les résultats de la session de génération à la première itération, chaque sortie était répétée trois fois, donc la visibilité était plus faible.

Cette phase nécessite un algorithme de filtrage des tests générés afin de trouver un sous-ensemble de tests qui correspond aux besoins de l'ingénieur vérification. Nous nous sommes limités dans cette thèse à concevoir une technique permettant de générer des tests candidats qui augmentent la visibilité des objectifs dans les sorties. Nous n'avons pas travaillé sur le filtrage qui est donc laissé pour de futurs travaux.

Nous avons sélectionné intentionnellement un exemple où les sorties possibles appartiennent à un petit ensemble (ici, 4 valeurs de sorties possibles) afin de pouvoir illustrer une situation difficile où une suite de tests avec une sortie unique pour chaque test n'existe pas. Dans d'autres programmes (où les sorties possibles appartiennent à des ensembles plus grands), une suite de tests avec une sortie unique pour chaque test peut exister et pourrait être trouvée grâce à notre méthode. L'implémentation de cette technique ayant été réalisée tout à

la fin de la thèse, nous avons laissé son évaluation expérimentale approfondie pour de futurs travaux. Notamment, le partenaire industriel MERCE a exprimé un intérêt pour évaluer cette méthode sur des programmes industriels.

Chapitre 6

Implémentation et expérimentations

Sommaire

Les chapitres précédents introduisent la problématique des objectifs polluants et détaille des méthodes qui peuvent être utilisées pour essayer de les détecter, ainsi que des problématiques autour de la génération de tests sur des critères plus complexes. Les objectifs de flot de données sont généralement très nombreux dans un programme et l'efficacité des différentes techniques de test pour ces objectifs peut avoir un impact significatif sur le processus de test. L'objectif de ce chapitre est donc de mesurer cet impact.

Dans ce chapitre, nous commencerons par détailler dans la Section 6.1 les outils que nous avons utilisés et l'implémentation dans ces derniers permettant à la fois de créer des objectifs de test pour les critères de flot de données, mais également de détecter des objectifs polluants et de générer des tests. Puis, nous décrirons dans la Section 6.2 plusieurs expérimentations effectuées pour tester et comparer les méthodes que nous avons mises au point pour détecter les objectifs polluants. Enfin, nous montrerons dans la Section 6.3 comment nous avons procédé pour tester l'efficacité des différentes approches présentées dans le Chapitre 5 vis-à-vis de la génération de tests et de la détection d'objectifs polluants.

6.1 Les critères de couverture de code avec LTest

Nous avons présenté dans le Chapitre 2 LTest [1] [Mar+17b], un ensemble d'outils écrits en OCaml comme des greffons de Frama-C [Kir+15], permettant de réaliser de manière automatique certaines étapes dans le processus de test d'un logiciel. Les techniques que nous avons mises au point dans les chapitres sur les objectifs polluants (Chapitre 4) et la génération de tests (Chapitre 5) ont été implémentées dans ces différents outils.

6.1.1 LAnnotate et les critères de flot de données

LAnnotate est, pour rappel, responsable de la génération d'objectifs. Plusieurs critères de base y sont présents, comme la couverture de condition, de fonctions ou encore les mutations faibles. Cependant, les critères plus complexes comme les critères de flot de données étaient très mal supportés. Nous avons décrit dans le Chapitre 3 avec l'Algorithme 5 une méthode naïve pour créer ces objectifs entre chaque paire définition-utilisation. Cependant l'implémentation dans LAnnotate était un peu différente et finalement bien moins exhaustive et prévisible. Pour commencer, les séquences étaient instrumentées différemment. Au lieu d'avoir une variable de statut avec un label comme dans nos exemples, une séquence était composée de deux parties.

La Figure 6.1 donne un exemple de cette ancienne méthode. Un label de séquence contenait dans l'ordre un identifiant unique de la séquence, un identifiant spécifique à chaque partie d'une même séquence, la taille de la séquence et un identifiant de variable. Ici on a donc la séquence numéro 2, avec ses deux parties (1/2 et 2/2) représentant respectivement le début (la définition) et la fin (l'utilisation) de la paire, sur la variable x d'identifiant 42 (dans l'arbre de

1. disponible sur https://git.frama-c.com/pub/ltest

```
1  int f(int cond){
2    int status_x_2 = 0;
3    int x = 42;
4    label_sequence(2,1,2,42);
5    if(cond){
6      label_sequence(2,2,2,42);
7      x = x + 1;
8      label_sequence_condition(0,42);
9    }
10   return x;
11 }
```

```
int f(int cond){
  int status_x_2 = 0;
  int x = 42;
  status_x_2 = 1;
  if(cond){
    label(2, status_x_2 == 1);
    x = x + 1;
    status_x_2 = 0;
  }
  return x;
}
```

Figure 6.1 : Annotation des objectifs liés à des séquences par LAnnotate : version originale (à gauche), et version actuelle (à droite)

syntaxe abstraite). Enfin, au lieu d'avoir une variable de statut pour savoir si on atteint l'utilisation avec la bonne définition, on utilise une macro (appelée `label_sequence_condition` dans la Figure 6.1) qui signifie que toutes les séquences sur la variable d'identifiant 42 seront cassées à cet endroit. Par "cassées", on veut dire ici que la séquence a été interrompue par une redéfinition de la variable. Cette implémentation a le mérite d'être plus lisible lorsque le code contient beaucoup de séquences, puisque au lieu de gérer le statut de manière individuelle, on regroupe les séquences par variable. Une seule condition suffit alors, ici à la ligne 8. Cependant, avec cette méthode, on donne aux autres outils la charge de transformer ce code pour simuler les séquences. Par exemple, LUncov, que nous avons présenté dans la Section 2.4.5 du Chapitre 2, effectuait une transformation pour transformer l'ancienne version en un code similaire à la nouvelle version.

Maintenant qu'on connaît l'ancienne forme des séquences, on peut décrire leur génération. Pour fonctionner, grâce à l'API de Frama-C, on parcourt l'AST de chaque fonction avec un *visiteur* pour générer nos objectifs. Ici, au lieu de générer toutes les paires possibles, le greffon utilisait ce parcours, qu'il réalisait plusieurs fois. Le premier visiteur avait simplement pour but de compter le nombre d'utilisations et de définitions de chaque variable de chaque fonction. Une fois cette première passe faite, un deuxième parcours utilisant les résultats du premier était réalisé : pour chaque définition d rencontrée, on créait un objectif entre d et toutes les utilisations qui n'avaient pas encore été visitées dans l'AST.

Et on peut voir ici les limites de cette implémentation purement syntaxique. En utilisant le parcourt de l'AST comme unique analyse, on ne prend pas en compte certaines constructions comme les boucles. En effet, une boucle demanderait de remonter dans l'arbre et ce n'est pas prévu par les visiteurs. Une pre-

mière approche a été de faire un troisième parcours, entre les deux décrits plus tôt, pour prendre en compte les boucles. Cette technique semblait fonctionner, mais il était dur de prévoir tous les comportements possibles en fonction du code d'origine et c'est pourquoi nous avons voulu changer de méthode. De plus, comme nous l'avons évoqué, cette méthode créait, elle aussi, des objectifs polluants, d'où la volonté de l'améliorer.

Les analyses de flot de données, du Chapitre 4 sur les objectifs polluants, ont été intégrées à notre processus de création d'objectifs, accompagnées par le changement de forme des séquences. Le premier effet a été de grandement simplifier le support des boucles : au lieu d'utiliser un parcours d'AST pour visiter notre programme, on utilise directement le graphe de flot de contrôle. Ainsi on prend en compte les boucles et on évite par la même occasion de créer les objectifs nécessitant de remonter dans le graphe alors qu'il n'y a pas de chemin le permettant. Ces objectifs sont non-applicables, mais ne sont pas pris en compte dans nos mesures, puisque cette implémentation les retire par défaut. Il aurait été intéressant de mesurer la différence entre cette utilisation du graphe et une version naïve, mais nous n'avons pas cette dernière. Cette technique de création d'objectifs nous servira donc de base que nous appellerons alors objectifs candidats.

Ensuite, en partant de cette méthode plus propre et cohérente, nous introduisons nos deux techniques M_{NA} and M_{Eq} pour filtrer respectivement les non-applicables (ceux qu'il reste après le premier filtre inhérent à notre implémentation) et les équivalents.

> **Remarque 25.** Tout le langage C n'est pas supporté par LTest et LAnnotate. En l'occurrence, nous ne prenons pas en compte les pointeurs et les tableaux sont gérés de manière globale. Ainsi une affectation de la forme `*p = 42;` sera vu comme une définition de la variable `p` et non de la région mémoire pointée par `p`. De la même manière, une affectation de la forme `t[i] = 42;` sera vu comme une utilisation de la variable `i` et une affectation globale de la variable `t` plutôt que d'une affectation de la zone mémoire spécifique à l'offset `&t+i`.

6.1.2 LUncov pour la détection d'objectifs infaisables

En ce qui concerne les objectifs infaisables, c'est Eva [Bla+17] et Wp [Bau+22] qui seront utilisés grâce au greffon LUncov. Là aussi un travail a été réalisé pour, dans un premier temps, ajouter le support des critères de flot de données (et donc des séquences), puis dans un second temps pour adap-

ter le code à la nouvelle forme de séquences. M_{WP} et M_{VA} dénotent donc les techniques de détection d'objectifs infaisables utilisant Wp et Eva. Pour chaque label dans le code, on crée une annotation ACSL **check** comme nous l'avons montré dans la Section 4.2 sur les objectifs infaisables. LUncov va ensuite faire appel aux analyseurs pour voir s'ils parviennent à prouver cette annotation. Si c'est le cas, alors l'objectif correspondant est infaisable.

6.1.3 PathCrawler pour la génération de tests

On a déjà beaucoup mentionné PathCrawler dans le Chapitre 2, et notamment la Section 2.5, et c'est donc évidemment cet outil que nous utiliserons pour effectuer la génération de tests pour mesurer et comparer les approches que nous avons détaillées dans le Chapitre 5. Grâce à notre modélisation des séquences, nous avons pu utiliser la technique existante de génération de tests pour les labels et n'avons pas été amené à modifier l'outil PathCrawler de manière significative.

6.2 Expérimentations avec la détection d'objectifs polluants

6.2.1 Description des benchmarks

Afin de tester nos différentes approches de détection d'objectifs polluants, pour les comparer et évaluer leur efficacité, nous allons les appliquer sur de vrais exemples de programme C. Open Source Case Studies [2], ou OSCS, regroupe un ensemble de plusieurs applications et benchmarks qui sont utilisés pour tester Frama-C et Eva. Ces programmes sont assez hétérogènes : leurs tailles varient de quelques dizaines de lignes à plusieurs milliers de lignes et on y trouve aussi bien des applications de jeux, de benchmarks, de cryptographie ou tests de *Common Weakness Enumeration* ou CWE (base de vulnérabilités logicielles). L'objectif est de générer des paires def-use dans ces programmes et ainsi tester nos techniques de détection. Les résultats seront présentés dans le tableau de la Figure 6.2.

Les programmes sur lesquels nous avons choisi de faire notre évaluation sont les suivants :

— **cwe787.** Comme son nom l'indique, c'est une implémentation C de cette cwe [3] qui essaie d'écrire avant le début ou après la fin d'un buffer ;

2. https://git.frama-c.com/pub/open-source-case-studies
3. https://cwe.mitre.org/data/definitions/787.html

— **2048.** Cette application est une implémentation C dans un terminal [4] du célèbre jeu du même nom ;

— **papabench.** Ce programme [5][Nem+06] est un benchmark sous la forme d'une application, basé sur le code d'un drone (UAV) du projet Paparazzi ;

— **debie1.** Le benchmark debie1 [6] est basé sur le logiciel embarqué pour l'instrument satellite DEBIE-1 [Kui+01] ayant pour objectif de mesurer les impacts de petits débris spatiaux et micrométéorites ;

— **gzip.** GNU Gzip [7] est un programme C de compression et décompression populaire ;

— **itc-benchmarks.** Des benchmarks d'analyse statique [8] de Toyota ITC ;

— **monocypher.** Monocypher [9] est une librairie de cryptographie conçue pour être rapide, légère et simple à utiliser.

6.2.2 Protocole expérimental

Pour chaque programme C, on génère d'abord l'ensemble des paires def-use dites *candidates*, c'est-à-dire les paires créées en utilisant une méthode naïve comme vu dans la section précédente. On utilise pour ça LAnnotate sans analyse supplémentaire et on obtient alors les résultats de la troisième colonne du tableau de la Figure 6.2. Ensuite, on applique les différentes techniques à tour de rôle, d'abord séparément (colonnes M_{NA}–M_{WP}), puis en les combinant (les quatre dernières colonnes). Pour chaque méthode, nous mesurons le temps d'exécution, le nombre de séquences (ou paires def-use) détectées comme polluantes dans l'ensemble des objectifs candidats et le pourcentage que cela représente par rapport au nombre total de candidats. La dernière ligne donne un pourcentage moyen par technique. TO et MO dénotent un timeout (on décide d'arrêter l'exécution si elle prend trop de temps, ici on s'est fixé à une limite de 10 heures) et manque de mémoire (l'exécution termine, car l'espace mémoire utilisé est plus grand que celui disponible).

Ces expériences ont été réalisées sur un ordinateur muni d'un processeur AMD Ryzen 5 5600X et 16 Go de RAM.

Il est important de noter que M_{VA} nécessite un point d'entrée dans le programme et un contexte initial pour fonctionner et lancer son analyse. Pour cette

4. https://github.com/mevdschee/2048.c
5. https://github.com/t-crest/patmos-benchmarks/tree/master/PapaBench-0.4
6. http://www.tidorum.fi/debie1/
7. https://www.gnu.org/software/gzip/
8. https://github.com/regehr/itc-benchmarks
9. https://monocypher.org

raison, certains objectifs peuvent être détectés comme polluants par rapport à ce contexte ou point d'entrée. Au contraire, M_{WP} va bien traiter toutes les fonctions du programme, peu importe le point d'entrée. Cette différence peut expliquer en partie des écarts de performances.

L'analyse de valeurs requière une expertise pour trouver les paramètres optimaux pour améliorer les résultats de l'analyse : modifier ces paramètres peut avoir un impact sur le temps d'exécution et sur la précision de l'analyse. Comme notre objectif est d'être le plus automatique possible pour permettre à n'importe quel utilisateur d'utiliser l'analyse, nous allons garder les paramètres par défaut de nos outils.

Comme nous l'avons mentionné, M_{WP} ne nécessite pas de point d'entrée dans le programme pour fonctionner. Cependant, ajouter des contrats avec des préconditions et postconditions ainsi que des annotations de boucles peut permettre d'améliorer fortement les performances de l'analyse. De la même manière que pour M_{VA}, nous n'avons pas ajouté ces annotations dans nos expérimentations pour que les résultats représentent les résultats obtenus sans expertise supplémentaire de la part de l'utilisateur. Par conséquent, un utilisateur expert peut probablement améliorer les résultats mesurés.

De même, utiliser des greffons de Frama-C dédiés à la génération d'annotations ACSL pourrait également améliorer ces résultats. Cependant, cela demanderait une adaptation de notre implémentation et nous avons donc choisi de garder cette possibilité pour de futurs travaux.

Pour notre évaluation, nous cherchons à répondre aux questions de recherche suivantes :

RQ1 : Est-ce que l'analyse de flot de données avec M_{NA} et M_{Eq} est efficace pour détecter les objectifs de test non-applicables et équivalents ? Est-ce qu'elle passe correctement à l'échelle sur de vraies applications ?

RQ2 : Est-ce que l'analyse statique, comme l'analyse de valeur et de plus faible précondition $M_{\mathrm{VA}}, M_{\mathrm{WP}}$, est capable de détecter des objectifs polluants ? Est-ce qu'elle passe correctement à l'échelle sur de vraies applications ?

RQ3 : Est-il utile de combiner ces méthodes ? Quelles sont les avantages et inconvénients ? Quelle est la meilleure technique en pratique ?

6.2.3 Analyse des résultats

Le tableau de la Figure 6.2 illustre nos résultats que nous détaillerons ensuite.

Ex./ Size	Indicateur	Cand. obj.gen.	Détection d'objectifs polluants							
			M_{NA}	M_{Eq}	M_{VA}	M_{WP}	$M_{\mathrm{NA,Eq}}$	$M_{\mathrm{NA,Eq,VA}}$	$M_{\mathrm{NA,Eq,WP}}$	$M_{\mathrm{NA,Eq,VA,WP}}$
cwe787 34 loc	time	0.05s	0.04	0.04s	2.0s	25.4s	0.04s	0.5s	4.2s	4.1s
	#seq.	208	144	100	155	108	172	178	172	178
	%seq.		69.2%	48.0%	74.1%	51.9%	82.7%	85.6%	82.7%	85.6%
2048 376 loc	time	0.09s	0.08s	0.1s	12.1s	1m29	0.08s	7.3s	42.0s	44.7s
	#seq.	560	159	187	196	14	293	321	294	321
	%seq.		28.4%	33.4%	35.0%	2.5%	52.3%	57.3%	52.5%	57.3%
papa-bench 1399 loc	time	0.19s	0.19s	0.2s	2.2s	62.6s	0.2s	2.0s	38.54s	32.5s
	#seq.	392	47	107	124	23	146	202	146	202
	%seq.		12.0%	27.3%	31.6%	5.9%	37.2%	51.6%	37.2%	51.6%
debie1 5165 loc	time	0.3s	0.23s	0.28s	2m24s	7m59s	0.2s	1m54	3m17s	5m55s
	#seq.	2149	815	825	907	181	1272	1350	1280	1350
	%seq.		37.9%	38.4%	42.2%	8.4%	59.2%	62.8%	59.6%	62.8%
gzip 4790 loc	time	1.2s	0.6s	0.9s	4m42s	80m	0.5s	4m	31m17s	22m9s
	#seq.	7337	3711	1047	4901	1311	4744	5392	4747	5392
	%seq.		50.6%	14.3%	66.8%	17.9%	64.7%	73.5%	64.7%	73.5%
itc-benchm. 11825 loc	time	1.1s	1.0s	1.1s	1m11s	12m19s	1.0s	54.3s	8m50s	6m14s
	#seq.	2881	198	705	1359	65	840	1659	905	1659
	%seq.		6.9%	24.5%	47.2%	2.3%	29.2%	57.6%	31.4%	57.6%
Mono-cypher 1913 loc	time	33.2s	3.6s	11.5s	MO	TO	3.0s	25m22s	TO	TO
	#seq.	47332	39759	24214	–	–	44605	44672	44605	44672
	%seq.		84.0%	51.1%	–	–	94.2%	94.4%	94.2%	94.4%
Average	%seq.		41.3%	33.9%	49.5%	12.7%	59.9%	69.0%	60.3%	69.0%

FIGURE 6.2 : Objectifs polluants détectés par différentes techniques et leurs combinaisons.

Nous avons choisi d'évaluer deux métriques pour comparer nos analyses, le temps et le nombre d'objectifs polluants trouvés. Pour le moment nous ne nous intéressons qu'aux quatre premières colonnes du tableau, à savoir nos quatre techniques en elles-mêmes.

Tout d'abord, en ce qui concerne le nombre d'objectifs polluants détectés, les résultats sont résumés sous la forme d'un graphique dans la Figure 6.3 donnant pour chaque technique et chaque cas d'étude une barre représentant le pourcentage d'objectifs polluants détectés. On constate ici plusieurs choses. Premièrement, l'analyse de flot de données permet de détecter un assez grand nombre d'entre eux, mais varie beaucoup selon les exemples, allant de 6,9% sur *itc-benchmarks* à 84% sur *monocypher* pour M_{NA} et les non-applicables et de 14,3% sur *gzip* à 51,1% encore une fois sur *monocypher* pour M_{Eq} et les équivalents. L'écart de performances est plus faible pour cette dernière technique et la moyenne de détection est de 41.3% pour M_{NA} contre 33.9% pour M_{Eq}. M_{NA} reste donc plus efficace. Cette différence de détection d'un exemple à l'autre s'explique en grande partie par le type d'implémentation. En effet, certains patterns se retrouvent régulièrement dans les codes avec beaucoup de polluants non-applicables et équivalents, notamment une répétition d'instructions de définition et d'utilisation massive (avec des macros par exemple) dans certaines fonctions.

Pour prendre le cas de *monocypher*, sur l'ensemble des objectifs candidats (soit 47332), 39212 appartiennent à la même fonction `blake2b_compress`. Le nombre d'objectifs de flot de données candidats croît rapidement en faisant le

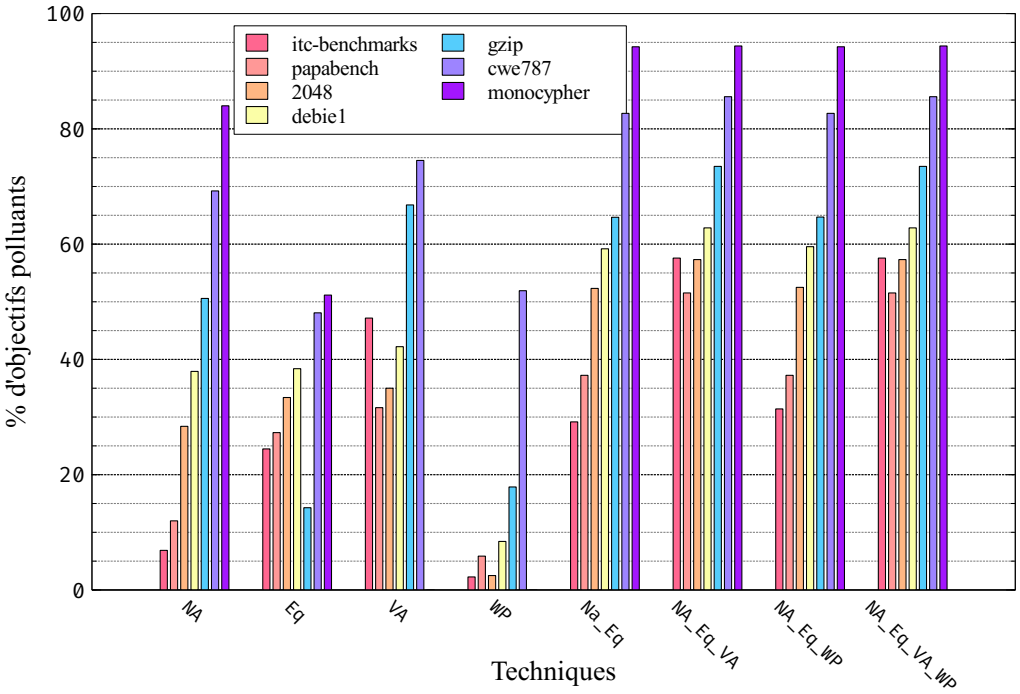


FIGURE 6.3 : Pourcentage d'objectifs polluants détectés

produit entre le nombre de définitions et d'utilisations. Si une fonction contient un très grand nombre de ce type d'instructions, alors le nombre d'objectifs explose. En effet, sur cette seule fonction de 39212 objectifs, 37440 (ou 95.5%) sont détectés comme étant non-applicables et 19699 (50%) équivalents. Au contraire, si un code introduit peu de variables et ne les redéfinit ou utilise que quelques fois, alors le nombre d'objectifs sera faible et donc celui des non-applicables ou équivalents aussi.

L'analyse M_{VA} quant à elle semble détecter plus d'objectifs polluants que l'analyse de flot de données, avec une exception sur *monocypher* ou le manque de mémoire ne permet pas, avec l'ordinateur utilisé pour faire ces benchmarks, d'évaluer ses performances. Nous avons mentionné dans la sous-section précédente que EVA (et donc M_{VA}) nécessite un point d'entrée dans le programme pour fonctionner. Le nombre d'objectifs polluants détectés contient donc des non-applicables, des infaisables vis-à-vis du point d'entrée et des infaisables dus au code lui-même. M_{VA} obtient une moyenne de détection de 49.5% en excluant donc *monocypher*. Pour ce dernier exemple, il est bon de noter que, si on exclut la fonction problématique `blake2b_compress` de l'analyse, EVA fonctionne correctement et est capable de trouver 2428 objectifs polluants sur les 8120 restants, soit 29.9%.

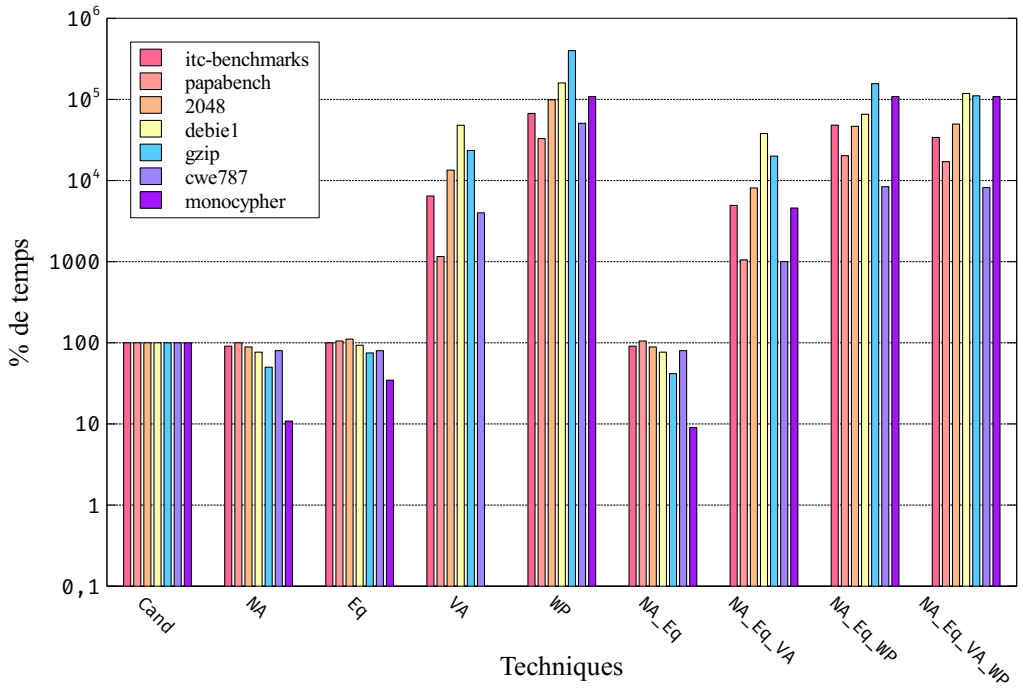


FIGURE 6.4 : Pourcentage de temps par rapport à la génération de candidats

Enfin, M_{WP} est la technique avec les moins bonnes performances. On détecte jusqu'à 51.9% d'objectifs polluants, mais sur la majorité des exemples cette valeur ne dépasse pas 10%, avec une moyenne de 12.7%. Ces faibles résultats peuvent s'expliquer facilement. Cette technique, utilisant WP, fonctionne de manière automatique comme les autres, sans que l'utilisateur ait besoin d'ajouter dans le code des contrats de fonctions et invariants de boucles à l'aide d'ACSL. Cependant, sans ces informations, les prouveurs automatiques sont souvent dans l'incapacité de déduire quoique ce soit. Si le code contient des objectifs infaisables, ils ne seront que rarement détectés par WP comparé à EVA.

En ce qui concerne le temps d'exécution des techniques, la Figure 6.4 donne une vue globale des performances par rapport à la génération de candidats. Les résultats obtenus sur les analyses de flot de données M_{NA} et M_{Eq} nous montrent que la génération de candidats est en règle générale plus lente que ces analyses, ce qui est d'autant plus visible sur le plus gros exemple *monocypher*. Ces améliorations s'expliquent facilement. L'analyse de flot de données passe très bien à l'échelle et ne coûte pas cher comparée aux performances des autres étapes de la création d'objectifs. Manipuler un AST peut prendre du temps, d'autant plus lorsque le nombre d'objectifs à ajouter dans ce dernier est grand. En évitant de créer certains objectifs polluants, on peut dans certains

cas gagner en performance. Le temps des deux techniques est en général très proches et varie principalement quand le nombre de polluants détectés est également très différent, comme ici sur *monocypher*. On est presque toujours sous la seconde avec un maximum de 11.5 secondes pour M_{Eq} sur *monocypher* : ces deux techniques sont très rapides en plus d'être efficaces.

Pour M_{VA}, on pouvait s'en douter, les performances sont un peu moins bonnes. Là où l'analyse de flot de données passe très bien à l'échelle, l'analyse de valeur a plus de mal. En effet, on constate que ses temps varient beaucoup, allant de 2 secondes sur *cwe787* à 4 minutes 42 secondes sur *gzip*. De plus, si on reprend les résultats de l'analyse sur *monocypher* en ignorant la fonction `blake2b_compress` comme plus tôt, l'exécution prend 2 heures et 10 minutes. Cette technique est donc très efficace et rapide sur des petits exemples, mais peut avoir plus de mal à passer à l'échelle.

Enfin, M_{WP} est de loin la technique la plus lente. Si on exclut le timeout au bout de 10 heures sur *monocypher*, on atteint quand même un maximum de 80 minutes pour *gzip* et un minimum de 25.4 secondes. Le temps d'exécution de cette technique dépend en grande partie du nombre d'objectifs à vérifier : pour chaque objectif on essaie de prouver son infaisabilité en générant un objectif de preuve et en lançant les prouveurs dessus.

Nous avons comparé nos techniques individuellement, mais il est également intéressant d'essayer de les combiner pour mesurer les performances et essayer si possible de trouver la meilleure technique en termes de temps d'exécution et de capacité à détecter les objectifs polluants. Si on reprend nos deux premiers graphiques (Figures 6.3 et 6.4), On constate que combiner nos deux analyses de flot de données M_{NA} et M_{Eq} est toujours une bonne idée. En effet, ces deux techniques peuvent être effectuées en parallèle sans coût supplémentaire et ainsi cumuler les objectifs polluants détectés. On est donc toujours aussi rapide, mais cette fois si on détecte entre 29.2% et 94.2% polluants pour une moyenne de 59.9%. On remarque également que l'utilisation des deux techniques donne un résultat inférieur à la somme des deux techniques séparées, on peut donc en conclure que certains objectifs non-applicables étaient également des objectifs équivalents.

Puisque cette méthode de combiner les deux techniques fonctionne très bien, nous les combinerons à chaque fois en plus d'autres techniques dans la suite de nos expérimentations. On commencera par filtrer tous les objectifs non-applicables et équivalents avec $M_{\mathrm{NA\_Eq}}$ pour ensuite appliquer nos techniques basées sur l'analyse statique. Sur les deux graphiques précédents, on remarque que les performances des autres combinaisons sont très proches de $M_{\mathrm{NA\_Eq}}$. Pour faciliter la lecture de résultats, nous introduisons deux nouveaux

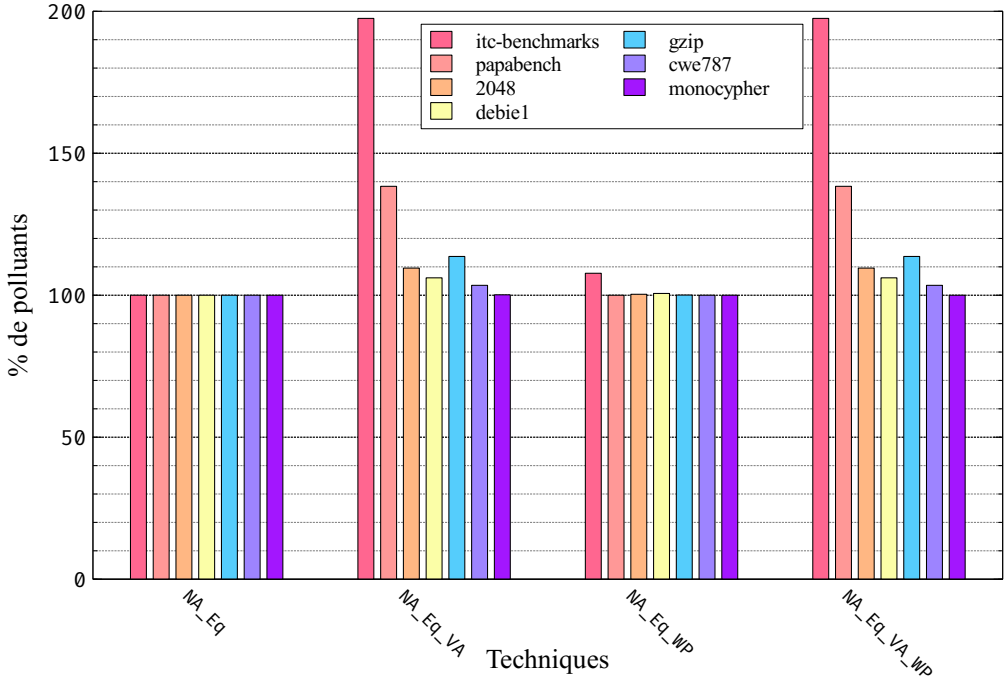


FIGURE 6.5 : Pourcentage d'objectifs polluants détectés par rapport à $M_{\mathrm{NA,Eq}}$

graphiques, où cette fois les résultats sont données par rapport à $M_{\mathrm{NA\_Eq}}$.

En regardant le graphique de la Figure 6.5, on constate qu'ajouter M_{VA} à $M_{\mathrm{NA\_Eq}}$ ($M_{\mathrm{NA\_Eq\_VA}}$) permet de détecter plus d'objectifs polluants, allant de près de deux fois plus d'objectifs pour *itc-benchmarks* (de 29.2% à 57.6%) à quelques pourcents, voire dixièmes de pourcents pour les moins bons exemples. Cependant, si on prend également en compte le temps avec le graphique de la Figure 6.6, on voit que le coût d'utilisation additionnel n'est pas négligeable, l'échelle du graphique étant logarithmique. Il faut alors se poser la question : faut-il privilégier le temps d'exécution à la capacité de détection ? La réponse à cette question variera selon les cas et choix de l'utilisateur. Dans tous les cas, utiliser M_{VA} après $M_{\mathrm{NA\_Eq}}$ permet d'améliorer ses performances puisque beaucoup d'objectifs sont alors filtrés avant de lancer l'analyse.

Concernant M_{WP}, encore une fois les résultats sont moins bons. Après application de la technique $M_{\mathrm{NA\_Eq}}$, cette technique ne permet pas de trouver beaucoup d'objectifs polluants (au maximum 2.2% sur *itc-benchmarks)* mais prend toujours beaucoup de temps comparée aux autres techniques. Ici le timeout sur *monocypher* est d'ailleurs toujours présent.

Enfin, utiliser à la fois M_{VA} et M_{WP} dans la colonne $M_{\mathrm{NA\_Eq\_VA\_WP}}$

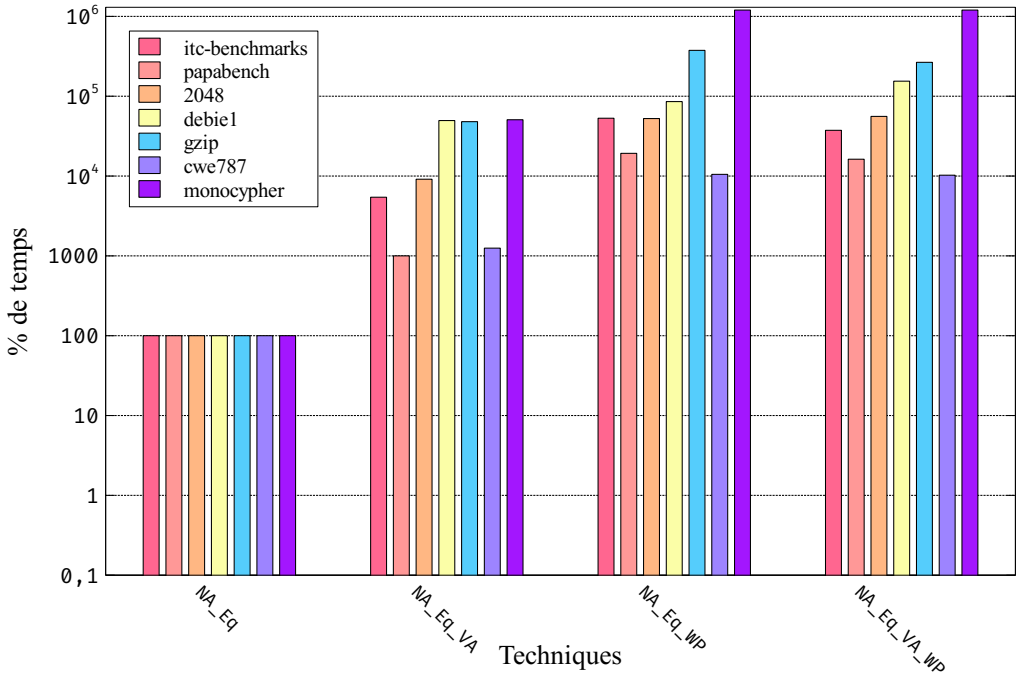


FIGURE 6.6 : Pourcentage de temps par rapport à $M_{\mathrm{NA,Eq}}$

ne semble présenter aucun avantage. La détection est équivalente à celle de $M_{\mathrm{NA\_Eq\_VA}}$, mais les performances, elles, sont une fois encore très ralenties à cause de M_{WP}.

6.2.4 Bilan

On peut maintenant répondre à nos questions de recherche. En ce qui concerne la **RQ1**, l'analyse de flot de données avec M_{NA} et M_{Eq} est très efficace pour détecter tous les objectifs non-applicables et équivalents. Elle ne permet en revanche pas de détecter les objectifs infaisables qui nécessitent des analyses plus poussées, mais passe par contre très bien à l'échelle, avec des temps d'exécution très courts peu importe la taille du programme expérimenté.

L'analyse statique, pour la **RQ2**, avec M_{VA} et M_{WP} permet de détecter certains objectifs non-applicables et infaisables. M_{VA} est plutôt efficace, avec le plus grand nombre d'objectifs détectés comme polluants. Cependant, certains objectifs sont détectés comme polluants en partant du point d'entrée utilisé pour l'analyse et ne le sont donc pas de manière absolue. M_{WP} est la technique la moins efficace, avec une détection plutôt faible, même si les objectifs détectés comme polluants le sont de manière certaine. Enfin, l'analyse statique ne

passe pas très bien à l'échelle. Son utilisation sur un programme volumineux, ou un programme avec beaucoup d'objectifs (les deux sont souvent liés) augmente grandement le temps nécessaire, surtout pour M_{WP} et on a même le cas M_{VA} qui échoue sur monocypher à cause d'une trop grande utilisation de la mémoire.

Enfin, pour répondre à la **RQ3**, la combinaison des techniques semble efficace, car combiner les deux analyses M_{NA} et M_{Eq} est toujours positif, les deux étant faites en parallèle dans la même analyse de flot de données ce qui n'augmente donc pas le temps nécessaire. De plus, une fois cette détection faite, on a moins d'objectifs présents pour l'analyse statique, ce qui a donc aussi un impact positif sur cette dernière.

Pour conclure, contrairement aux résultats obtenus par des travaux précédents sur des critères plus simples [Bar+15 ; Mar+18], ici la technique utilisant Wp n'apporte pas beaucoup de détection comparée aux autres techniques. Nous pensons que c'est dû à la nature complexe des critères de flot de données, où l'infaisabilité à moins de chance d'être détecté par un raisonnement local. Cela pourrait donner lieux à de nouveaux travaux pour essayer d'améliorer les performances de cette technique en ajoutant des annotations supplémentaires. La meilleure technique en temps d'exécution est de loin $M_{\text{NA\_Eq}}$, cependant il est assez intéressant de pouvoir trouver, même un peu, plus d'objectifs polluants et l'utilisation de $M_{\text{NA\_Eq\_VA}}$ peut donc être recommandée au prix des performances.

6.3 Évaluation immédiate et retardée lors du test aux limites des critères de flot de données

On a présenté dans le Chapitre 5 (Section 5.2) un nouveau critère combinant les critères de flot de données classiques avec les tests aux limites afin de contraindre les définitions des séquences à des valeurs spécifiques (minimum ou maximum du type concerné) et tester ainsi des cas particuliers de flot de données qui pourraient amener à des débordements d'entiers par exemple. On a également vu dans la Section 5.1 qu'il était possible d'instrumenter ce type de critères d'au moins deux façons différentes : pour une évaluation *immédiate* et *retardée*. Ce choix pourrait avoir un impact théorique sur l'efficacité de la génération de tests et nous avons donc voulu faire une évaluation pour mesurer les performances des deux approches et les comparer. L'intuition voudrait que le retardement de l'évaluation améliore les performances, puisque cette méthode réduit le nombre de chemins explorés.

6.3.1 Description des benchmarks

Pour tester ce nouveau type de critères et nos deux approches, nous avons choisi des exemples de code C classiques qui font partie de la suite de tests de PathCrawler. Tous ces codes sont relativement courts, mais utilisent notamment des boucles et des tableaux. Nos résultats seront présentés dans le tableau de la Figure 6.7.

- **ADPCM.** Adaptive Differential Pulse Code Modulation, ou Modulation par Impulsions et Codage Différentiel Adaptatif en français, est un algorithme de compression de données avec perte ;
- **Merge.** Ce programme sert à fusionner deux tableaux triés dans un troisième tableau trié ;
- **Bsearch.** Une implémentation d'un algorithme de recherche par dichotomie ;
- **TriType.** Triangle Type est un petit algorithme qui prend en entrée trois longueurs de côtés et qui renvoie le type de triangle : quelconque, isocèle, équilatéral ou un code erreur si ce n'est pas un triangle ;
- **Tcas.** Traffic alert and Collision Avoidance System, ou système d'alerte de trafic et d'évitement de collision en français, est un programme utilisé en aviation pour calculer et éviter les collisions ;
- **Bsort.** Pour Bubble sort ou tri à bulles en français, est une implémentation simple de cet algorithme de tri.

6.3.2 Protocole expérimental

Pour réaliser nos expérimentations, nous utiliserons les différents greffons présentés précédemment :

- Annotation des objectifs avec LAnnotate ;
- Détection d'infaisables avec LUncov ;
- Génération de tests avec PathCrawler.

Pour chaque programme présenté dans la Section 6.3.1, chaque outil sera appliqué deux fois : une première fois pour la version *immédiate* de notre approche, c'est-à-dire que le prédicat de départ de la séquence sera évalué au début de cette dernière (soit au niveau de la définition) et une seconde fois pour la version *retardée*, où le prédicat de départ de la séquence sera évalué à la fin (au niveau de l'utilisation). Le choix entre ces deux approches est réalisé via la première étape d'annotation, car LAnnotate permet de choisir quel type d'évaluation on veut avoir. Le critère utilisé pour ces expérimentations sera **BDUC** pour

Benchmark Nb. lignes	Séquences	Infaisables	Couvertes	Approche	Temps Infaisables	Temps Génération	Chemins explorés	Tests générés
ADPCM	24	12	6	Immédiate	0.05s	2.5s	658	21
26 lignes				Retardée	0.07s	1.5s	333	14
Merge	48	44	0	Immédiate	0.1s	4.1s	671	16
26 lignes				Retardée	0.18s	3.7s	633	16
Bsearch	20	16	4	Immédiate	0.06s	3.0s	823	29
20 lignes				Retardée	0.08s	0.8s	12	3
TriType	78	42	36	Immédiate	0.2s	9.9s	2264	121
22 lignes				Retardée	0.37s	1.7s	116	24
Tcas	16	16	0	Immédiate	0.04s	-	-	-
124 lignes				Retardée	0.05s	-	-	-
Bsort	18	10	3	Immédiate	0.05	32min21	38873	363
18 lignes				Retardée	0.06	3min12	16158	154

FIGURE 6.7 : Résumé des expériences avec LTEST sur des séquences de flot de données aux limites

Boundary Def-Use Coverage que nous avons ajouté dans LAnnotate. Comme indiqué précédemment (Section 5.2), lorsque ce critère est sélectionné, LAnnotate va créer deux séquences par paire def-use si le type de la variable concernée le permet (c'est-à-dire si c'est un type entier), une pour son minimum et une pour son maximum.

Une fois l'annotation faite avec notre critère générant des paires def-use aux limites, on utilisera LUNCOV comme dans la Section 6.2 et en particulier l'approche utilisant EVA pour détecter les objectifs infaisables s'il y en a. Enfin, nous utiliserons PATHCRAWLER pour générer les tests et mesurer ses performances (nombre de chemins parcourus, nombre de cas de tests et temps d'exécution).

Pour notre évaluation, nous cherchons à répondre aux questions de recherche suivantes :

RQ4 : Quelle est la meilleure approche, entre *immédiate* et *retardée*, du point de vue de la génération de tests ?

RQ5 : Quelle est la meilleure approche, entre *immédiate* et *retardée*, du point de vue de la détection des objectifs polluants ?

RQ6 : Quelle est la meilleure approche, entre *immédiate* et *retardée*, sur l'ensemble de la chaîne ?

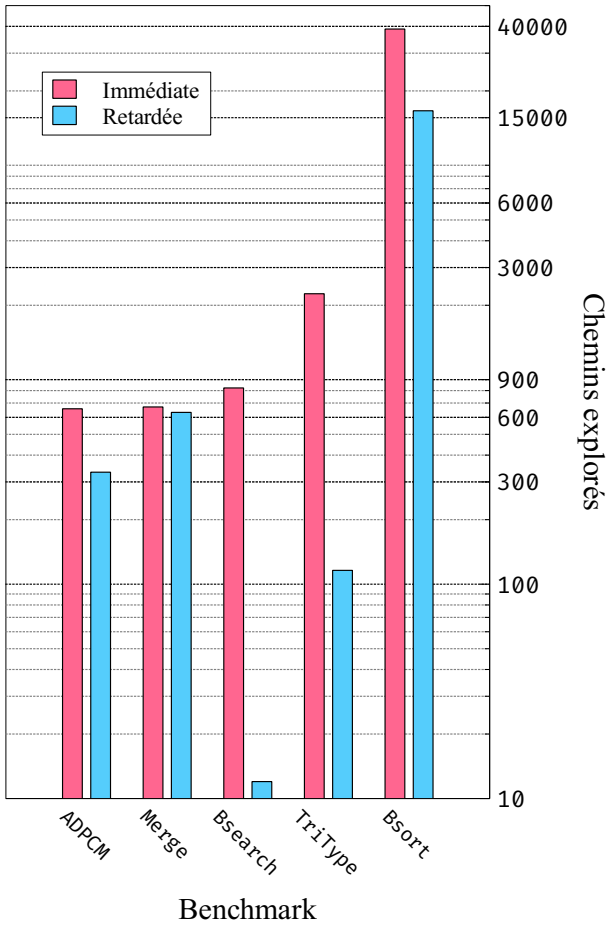


FIGURE 6.8 : Nombre de chemins explorés par la génération de tests

6.3.3 Analyse des résultats

Le tableau de la Figure 6.7 résume nos résultats obtenus en comparant nos deux approches *immédiate* et *retardée* sur des petits programmes C.

Pour chaque programme, on s'intéressera de manière globale à sa taille en nombre de lignes, au nombre d'objectifs (séquences aux limites), au nombre d'objectifs infaisables et couverts. Ensuite, pour chaque approche, on mesurera le temps de la détection d'infaisables, le temps mis par la génération de tests, le nombre de chemins explorés et pour finir le nombre de tests générés par PATHCRAWLER.

La Figure 6.8 illustre graphiquement le nombre de chemins explorés avec une échelle logarithmique. On constate que les mesures obtenues correspondent à ce qu'on attendait, l'approche d'évaluation *retardée* diminue significativement le nombre de chemins explorés lors de la génération de tests. On

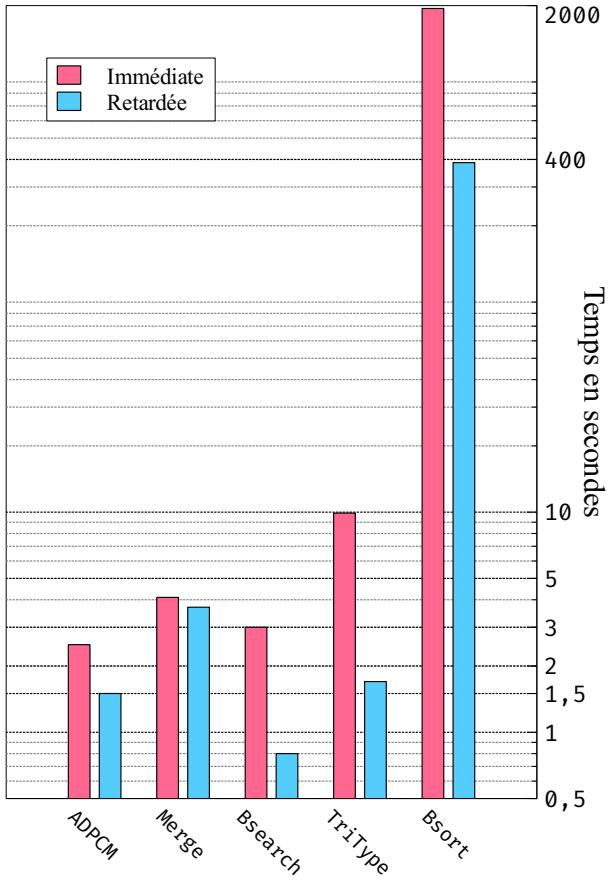


FIGURE 6.9 : Temps d'exécution de la génération de tests

passe ainsi de 658 chemins explorés à 333 sur ADPCM et de 38873 à 16158 sur Bsort, soit près de 50% de réduction. Les exemples de Bsearch et TriType sont les plus marquants, où on passe de 823 et 2264 chemins dans l'approche *immédiate* à 12 (-99%) et 116 (-95%) chemins dans l'approche *retardée*. Enfin, on constate que l'approche n'a que peu d'impact, même s'il existe, sur l'exemple Merge. Cette réduction du nombre de chemins explorés afin de couvrir les labels a donc également un impact sur le nombre de tests générés par notre outil, moins on explore et moins on génère de tests.

Concernant Tcas, tous les objectifs sont infaisables, il n'est donc pas possible de les couvrir en utilisant la génération de tests.

En termes de temps d'exécution, on constate avec la Figure 6.9 que l'approche d'évaluation *immédiate* est plus lente, la différence avec l'approche *retardée* étant très marquée sur certains exemples. On gagne ainsi de quelques

Benchmark Nb. lignes	Séquences	Approche	Infaisables	Temps Infaisables
2048	350	Immédiate	281	10s
376 lignes		Retardée	275	13s
debie1	1380	Immédiate	1129	123s
5165 lignes		Retardée	1104	235s
itc-benchmarks	1516	Immédiate	1495	53s
11825 lignes		Retardée	1497	80s

FIGURE 6.10 : Mesure de la détection d'infaisables avec LTEST sur des programmes d'Open Source Case Studies

dixièmes de secondes sur Merge, à quelques secondes sur ADPCM, Bsearch et TriType. Enfin, sur l'exemple de Bsort, les performances sont améliorées par un facteur 10, passant de plus de 32 minutes à un peu plus de 3 minutes.

Cependant, on constate des résultats inverses sur le temps d'exécution de la détection d'infaisables. En effet, même si la différence est négligeable comparée au reste, on observe une légère réduction du temps de la détection sur l'approche *immédiate* par rapport à l'approche *retardée*. L'approche *retardée* semble donc avoir un impact négatif sur la détection d'objectifs infaisables. Les temps des exemples précédents ne semblent pas très significatifs, car très faibles, mais cette différence existe bel et bien. Le passage de gros exemples dans PATHCRAWLER prenant un temps considérable, nous n'avons pas pu tester la génération de tests sur des exemples plus gros comme pour les objectifs polluants via Open Source Case Studies, mais nous avons voulu quand même mesurer les performances de LUNCOV en fonction des approches sur certains des exemples. Le tableau de la Figure 6.10 montre ces mesures.

Ici, de manière similaire aux exemples précédents, on mesure le nombre d'objectifs et les temps d'analyse entre les deux approches sur la détection d'infaisables avec LUNCOV. On observe beaucoup mieux la différence de temps, l'approche *retardée* est plus longue, de quelques secondes sur 2048 a presque 2 minutes sur debie1, ce qui est en accord avec nos mesures sur les petits exemples. On note également une légère différence sur le nombre d'objectifs infaisables détectés.

6.3.4 Bilan

A partir des résultats obtenus sur nos petits exemples, on peut répondre à nos questions de recherche. Dans un premier temps pour la **RQ4**, l'approche *retardée* semble plus efficace dans tous les cas pour la génération de tests. En effet, elle permet de souvent réduire fortement l'exploration et le nombre de tests générés qui sont deux métriques très importantes pour le test, car pour fonctionner sur des programmes plus volumineux il faut optimiser au maximum les performances pour un meilleur passage à l'échelle. Cette différence s'explique notamment grâce à la technique d'instrumentation choisie pour l'évaluation retardée, similaire à l'instrumentation compacte (Section 2.5). Ici au lieu de créer deux branches pour la génération au niveau du prédicat avec la version *immédiate*, on place le prédicat dans le label à la fin de la séquence (comme nous avons illustré dans la Figure 5.4). Par conséquent, ces deux branches supplémentaires (et les chemins qui vont avec) seront explorées une seule fois pour essayer de couvrir le label, puis ignorés pour le reste de l'exploration.

Cependant, pour la **RQ5**, l'approche *retardée* semble moins efficace pour la détection d'infaisables. Les tests sur des programmes plus volumineux mettent en avant une différence significative du temps nécessaire pour compléter l'analyse. De plus la différence du nombre d'objectifs polluants détectés semble indiquer que selon l'approche, EVA va plus ou moins réussir à prouver l'infaisabilité de certains objectifs. Cette différence est cependant minime et ne permet pas vraiment de conclure quelle approche est à privilégier ici.

Dans le cas où l'on souhaite effectuer à la fois la détection d'infaisables et la génération de tests il faut alors décider de l'approche à privilégier. En réponse à la **RQ6**, la solution idéale semble d'effectuer une première annotation en utilisant l'approche *immédiate*, dans le but de détecter les objectifs infaisables, puis, en gardant ces résultats, revenir à l'approche *retardée* pour la génération de tests. Le temps nécessaire à l'annotation étant négligeable, surtout sur des codes volumineux (cf. résultats de la Section 6.2), faire deux fois cette étape ne sera pas très impactant sur les résultats globaux.

Chapitre 7

Vérification de contre-mesures contre l'injection de fautes

Sommaire

Après avoir contribué à la détection d'objectifs polluants pour les critères de flot de données (Chapitre 4) nous nous sommes demandés si cette détection ne pouvait pas servir dans un cadre assez différent : la détection d'erreurs dans les contre-mesures contre l'injection de fautes. Dans ce chapitre, nous présenterons d'abord ce domaine et les enjeux qui l'entourent (Section 7.1), puis

```
if(password != secret) return 1; //Erreur, mot de passe faux
if(password != secret) return 1; //Erreur, mot de passe faux
// Protégé : Authentifié avec succès
```

FIGURE 7.1 : Vérification d'un mot de passe avec contre-mesure.

nous détaillerons notre approche pour vérifier ces contre-mesures (Section 7.2) et les difficultés rencontrées (Section 7.3) ainsi que son implémentation (Section 7.4) et nous finirons par un cas d'étude sur WOOKEY, un logiciel écrit en C qui contient des contre-mesures et est supposé être résistant à un certain type d'attaques (Section 7.5).

7.1 Injections de fautes et contre-mesures

7.1.1 Contexte

On a vu dans le Chapitre 2 plusieurs techniques permettant d'injecter des fautes dans un système dans le but de modifier son comportement et de trouver des vulnérabilités. Pour contrer ces attaques, les concepteurs de logiciels embarqués utilisent des techniques de contre-mesures. Ainsi, au lieu d'utiliser un système booléen classique avec `0` et `1` en C, on peut vouloir utiliser `0×55aa55aa` et `0×aa55aa55`, des booléens ayant une distance binaire maximale appelés *secbool*, pour protéger l'inversion de bit. On peut également utiliser des techniques à base de code correcteur pour vérifier l'intégrité des données. Un des schémas souvent utilisés se base sur la redondance de code [Bar+10 ; Bar+12]. Dans certains de ces schémas, les vérifications critiques (instructions conditionnelles) dans le code sont dupliquées comme dans la Figure 7.1 présentée dans la section suivante. De cette façon, si un attaquant réussit à contourner une vérification critique en injectant une faute pour inverser le résultat d'un test, la vérification redondante empêchera quand même d'atteindre la zone protégée du code. Ce type de contre-mesures fait l'hypothèse qu'il est improbable d'injecter deux fautes par attaque matérielle pendant la même exécution de manière coordonnée. On peut généraliser ça pour tout nombre $k \geq 1$ de fautes coordonnées, chaque vérification critique doit alors être répétée $k + 1$ fois. Les travaux que nous allons présenter ici portent sur ce type de contre-mesures à base de redondance de code.

7.1.2 Exemples

La Figure 7.1 illustre un fragment de code C simple qui utilise une contre-mesure à base de redondance de code. En supposant que `password` est un mot de passe fourni par l'utilisateur et que `secret` est le mot de passe correct, la duplication de l'instruction conditionnelle nous assure qu'un mauvais mot de passe sera détecté même si une des conditions est inversée par une attaque. La Figure 7.2 montre un exemple plus intéressant avec une redondance de vérification d'intégrité faite par la fonction `check_code_integrity`. Pour protéger l'inversion de bit, cette fonction renvoie une valeur de type `secbool` comme présenté précédemment. La deuxième instruction **`if`** est écrite un peu différemment et est erronée : le développeur aurait dû utiliser une négation bit-à-bit `~chk2` au lieu d'une négation logique `!chk2`. Que `chk2` vaille `sectrue` ou `secfalse`, sa négation logique donnera toujours 0 et le test à la ligne 21 est donc toujours faux. Par conséquent, si un attaquant réussit à inverser le résultat du test à la ligne 19, il pourra ensuite exécuter le code protégé de la ligne 22 même si la vérification d'intégrité échoue. Cet exemple illustre une contre-mesure incorrecte, causée par une mauvaise utilisation du type `secbool`. D'autres exemples de contre-mesures seront donnés dans la suite de ce chapitre.

L'exemple ci-dessus montre qu'écrire des contre-mesures n'est pas trivial. Il est donc important de pouvoir vérifier qu'elles sont bien implémentées. Nous avons choisi de faire cette vérification au niveau du code source, mais beaucoup d'autres approches existent à tous les niveaux (Section 2.2). Vérifier le code source implique qu'on ne vérifie pas que la compilation a conservé les contre-mesures correctement. En effet, les compilateurs peuvent être tentés de simplifier du code qu'ils considèrent inutile, par exemple du code *mort* (c'est-à-dire inatteignable). Or, il se trouve que les contre-mesures par redondance de code donnent souvent lieu à du code mort lors d'une exécution normale (sans attaque). Il faut donc penser à désactiver ce type d'optimisations si on veut les conserver. Par exemple, les règles de développement en C de l'Agence nationale de sécurité des systèmes d'information (ANSSI) [ANS21] spécifient que *le développeur doit s'assurer qu'un haut niveau d'optimisation n'élimine pas du code défensif ou des contre-mesures logicielles manuellement ajoutées.*

7.2 Approche de vérification

Notre approche de vérification fonctionne de la manière suivante. L'utilisateur commence par définir ce que nous appellerons des *zones critiques*, c'est-à-dire des zones dans lesquelles une mutation pourrait être problématique et dont on voudrait s'assurer que toutes les conditions sont résistantes aux attaques par

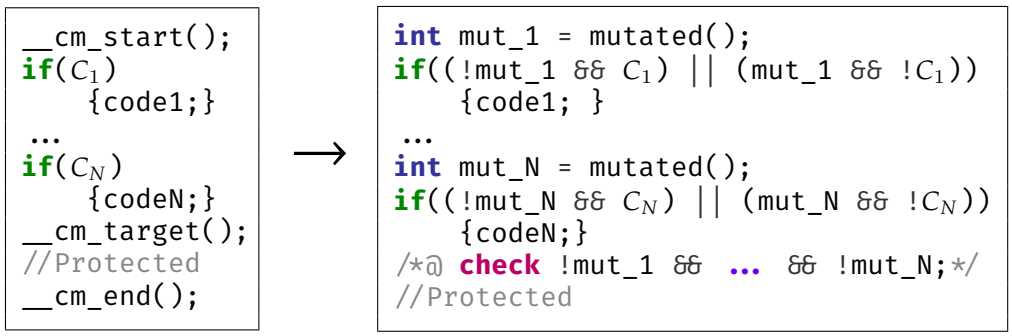
```c
// valeurs vrai/faux sécurisées
typedef enum {secfalse = 0x55aa55aa,
              sectrue = 0xaa55aa55} secbool;

#define SIZE 2
#define SUM 10
int integrity[SIZE];

secbool check_code_integrity(){ // Vérification d'intégrité
    int sum=0;
    for(int i = 0; i < SIZE; i++)
        sum += integrity[i];
    if(sum == SUM) return sectrue;
    return secfalse;
}

int main(){
    secbool chk1=check_code_integrity();
    if(chk1 != sectrue) return 1; // Erreur, code compromis
    secbool chk2=check_code_integrity();
    if(!chk2 == sectrue) return 1; // Contre-mesure erronée
    // Protégé : Succès de la vérification d'intégrité
}
```

FIGURE 7.2 : Vérification d'intégrité avec une contre-mesure.

injection de fautes. Cette étape permet de concentrer notre analyse uniquement sur les étapes critiques (contenant une authentification, des vérifications d'intégrité, un contrôle de version, etc.) puisque les autres parties du code (après l'authentification par exemple) n'intégreront pas de contre-mesures. Selon les cas, la zone *protégée* (qui donne par exemple accès à des ressources ou informations critiques) peut être à l'intérieur de la zone critique, ou juste après. Notre approche ajoute deux nouvelles macros, `__cm_start()` et `__cm_end()` permettant de délimiter une section critique.

Dans cette zone critique, tous les tests (dans le cas du langage C qui nous intéresse, les conditions `if( ... )`) seront instrumentés pour simuler des fautes introduites par un attaquant selon le modèle d'attaque considéré : un attaquant peut inverser jusqu'à k tests dans le programme.

> **Remarque 26.** Le code étant normalisé par FRAMA-C (voir la Section 2.4), les tests de sortie des boucles `for` et `while` sont également concernés puisqu'ils seront explicitement introduits sous forme d'instructions condition-



(a) Exemple de code annoté à la main

(b) Et son instrumentation automatique par mutations pour la simulation de fautes

FIGURE 7.3 : Exemple simple de notre approche

> nelles. Nous en reparlerons dans la Section 7.3 sur les difficultés de notre approche.

Enfin, on ajoute une macro `__cm_target()` qui indiquera la portion de code dont on veut protéger l'accès. La Figure 7.3a donne un exemple avec les macros que l'on vient de présenter.

De la même manière que pour la détection d'objectifs infaisables, nous utiliserons ensuite un outil de vérification déductive sur le code instrumenté pour essayer de prouver formellement la protection du code spécifié. La Figure 7.3b présente le schéma d'instrumentation pour une partie critique du code avec N tests. Nous allons maintenant détailler cette instrumentation.

7.2.1 Simulation de fautes

Pour simuler une possibilité d'injection de faute, on introduit une fonction appelée `mutated`, pour laquelle on fournit uniquement une spécification ACSL, comme montré dans la Figure 7.4. Ici, la spécification nous permet donc de décrire formellement le comportement de la fonction `mutated` sans avoir besoin de fournir son implémentation, car elle n'est pas requise pour notre technique basée sur la vérification déductive.

> **Remarque 27.** Il pourrait être intéressant, pour d'autres types d'analyse, comme l'analyse par interprétation abstraite ou la recherche d'attaques par génération de tests, de fournir une implémentation de cette fonction.

On peut simuler au plus k fautes (pour tout $k \geq 1$) en changeant la valeur de `MAX_MUTATION` à k (ici, on choisit $k = 1$). Les spécifications des lignes 3 à

```
1  #define MAX_MUTATION 1 // Nombre maximum de mutations
2  unsigned int cpt_mut = 0;
3  /*@
4    assigns cpt_mut;
5    behavior cannot_mutate:
6      assumes cpt_mut ≥ MAX_MUTATION;
7      ensures !\result;
8      ensures cpt_mut == \at(cpt_mut, Pre);
9    behavior can_mutate:
10     assumes cpt_mut < MAX_MUTATION;
11     ensures  \result ⟺ cpt_mut == \at(cpt_mut,Pre) + 1;
12     ensures !\result ⟺ cpt_mut == \at(cpt_mut,Pre);
13 */
14 int mutated();
```

FIGURE 7.4 : Fonction non interprétée `mutated` et son contrat

13 garantissent que la fonction `mutated` retournera *vrai* (c'est-à-dire différent de 0) au plus `MAX_MUTATION` fois pendant l'exécution. L'ordre dans lequel elle renvoie *vrai* ou *faux* n'est pas spécifié et peut donc être arbitraire. Pour compter le nombre de fois où la fonction a retourné *vrai* jusqu'à présent, on introduit une nouvelle variable, `cpt_mut`. La ligne 4 spécifie que `mutated` ne modifie que `cpt_mut` et n'interfère donc pas avec le reste du code de l'application. Ensuite, on considère deux cas (lignes 5 et 9) :

- Si le nombre maximal de mutations a été atteint (ligne 6), la fonction retourne *faux* (donc zéro), c'est-à-dire qu'elle ne provoque pas de mutation et le compteur reste inchangé (lignes 7 et 8) ;
- Si le nombre maximal de mutations n'a pas été atteint (ligne 10), alors soit la fonction retourne *vrai* et incrémente le compteur (provoque une mutation), soit elle retourne *faux* et ne touche pas au compteur (lignes 11 et 12).

Comme montré dans la Figure 7.3b (par exemple à la ligne 1), pour chaque test d'une condition C_i, un appel à `mutated` est ajouté et son résultat est stocké dans une variable `mut_i`, qu'on appelle un *déclencheur de mutation*. La valeur de C_i est ensuite combinée avec notre déclencheur de mutation dans le but de déclencher une inversion de test, c'est-à-dire de prendre la branche opposée, si `mutated` retourne *vrai* (à la ligne 2 par exemple). Dans le cas où `mutated` retourne *faux*, on garde la bonne branche.

Ce genre d'instrumentation peut être comparé, bien qu'à un niveau d'abstraction légèrement plus élevé, à celles réalisées par l'outil LAZART [Pot+14] sur du bytecode LLVM ou bien avec FRAMA-C [Lau+19], mais en se basant sur une traduction de code assembleur au niveau C.

7.2.2 Propriétés vérifiées

Au début de la section de code protégée (à l'emplacement de la macro `__cm_target()`, ligne 8 dans la Figure 7.3b), on insère une annotation ACSL **check** qui fait la conjonction de la négation de tous nos déclencheurs de mutation. Ici l'objectif est d'essayer de prouver que cette conjonction est vraie, c'est-à-dire que tous les déclencheurs sont *faux* à ce point du programme. Si en atteignant la partie protégée on est capable de prouver cette annotation, alors on peut conclure que la section protégée du code ne peut jamais être atteinte avec entre 1 et k inversions de test. En d'autres termes, la contre-mesure à base de redondance de code est correctement implémentée et protège effectivement le code qui suit en cas d'inversion de test. En effet, s'il existe un chemin d'attaque avec un nombre de fautes différent de zéro, c'est-à-dire sur lequel un déclencheur de mutation au moins est *vrai*, cette annotation ne pourra pas être prouvée. Pour essayer de prouver cette annotation, on se repose sur le greffon Wp de Frama-C, qui est basé sur la vérification déductive.

La technique de vérification est donc basée sur la propriété suivante.

> **Propriété 1.** Une contre-mesure est considérée comme efficace si et seulement si l'assertion **check** est vérifiée, c'est-à-dire que la preuve formelle démontre l'inatteignabilité de cette assertion par injection de fautes conformément au modèle considéré.

7.3 Difficultés rencontrées et solutions proposées

Notre méthode se veut la plus automatique possible, l'objectif étant de pouvoir prouver l'efficacité des contre-mesures sans être un expert en vérification déductive et en spécification de code. Toutefois, cette approche (voir Section 6.2.3) pose un problème particulier : la vérification déductive et les outils qui la réalisent demandent souvent de spécifier en détail certaines parties du code pour les aider à raisonner correctement sur le programme. De plus, le langage C est complexe et gérer tous les cas possibles n'est pas forcément trivial. Nous allons voir comment contourner certaines de ces difficultés.

7.3.1 Appels de fonctions

Comme on l'a mentionné, les outils de vérification déductive comme Wp ont besoin de ce qu'on appelle des contrats de fonction pour pouvoir raisonner sur un code qui appellerait ensuite ces fonctions. Il faudrait alors, de la même manière que pour notre fonction `mutated` de la Figure 7.4, fournir la

spécification de chaque fonction appelée dans une section critique. Dans notre approche, avoir à les fournir irait à l'encontre de notre objectif de rendre ce processus de vérification le plus automatique possible. Par exemple, pour faire fonctionner notre technique sur le code de la Figure 7.2, l'utilisateur aurait à fournir une spécification complète de la fonction `check_code_integrity`. Essentiellement, cette technique [1] transforme la fonction en une sorte de macro, qui fait inclure tout le code de la fonction au moment de l'appel plutôt que de faire l'appel.

Pour éviter ce genre de cas, on propose de contourner le problème en inlinant les fonctions appelées (mot clé `inline` en C) dans une section critique pour que la vérification déductive puisse raisonner sur le code sans annotations additionnelles.

Le code de cette fonction est maintenant dupliqué à plusieurs endroits : celui de sa déclaration initiale et également à tous les endroits où elle est appelée. L'utilisateur peut choisir soit d'instrumenter le code inliné de la fonction, soit de ne pas le faire en considérant que la fonction est déjà protégée contre l'injection de fautes (et le prouver séparément) ou qu'elle n'a pas d'importance vis-à-vis des contre-mesures présentes dans le code de l'appelant. Dans les exemples présentés ici, pour plus de lisibilité, les blocs de code venant de fonctions inlinées ne seront pas instrumentés.

7.3.2 Boucles

Le problème ici est assez similaire à celui des appels de fonction. Pour que les outils de vérification déductive fonctionnent correctement, il faut les aider en donnant des spécifications pour chaque boucle, comme nous l'avons présenté dans la Section 2.4 avec ACSL, avec un invariant (ce qui doit être vrai pour chaque itération), un variant (une borne supérieure du nombre d'itérations restantes qui sert à prouver la terminaison), etc. Ces spécifications demandent donc de l'expertise et réduisent l'automaticité de notre approche. Pour les contourner, on va dérouler les boucles [2]. Avec l'outil Frama-C, il suffit pour cela d'ajouter, juste avant la boucle de la ligne 11 pour notre exemple de la Figure 7.2, l'annotation `/*@ loop pragma UNROLL SIZE+1, "completely";*/`. On demande ici à Frama-C de dérouler la boucle `SIZE+1` fois et de considérer (et essayer de prouver) ensuite que la boucle a été déroulée entièrement. Le code sera alors transformé pour effectuer les `SIZE+1` premières itérations hors de la boucle.

1. Un exemple complet d'une fonction inlinée sera donnée ci-dessous pour le code de la Figure 7.2 (Figure 7.12 après annotation) par la Figure A.7 en annexe.
2. Un exemple complet d'une boucle déroulée sera donnée ci-dessous pour le code de la Figure 7.2 (Figure 7.12 après annotation) par la Figure A.7 en annexe.

```
for(int i = 0;i < SIZE; i++){
  __cm_start();
  if(password[i] != secret[i]) break; // Mot de passe faux
  if(password[i] != secret[i]) break; // Mot de passe faux
  __cm_target();
  // Protégé : Authentifié avec succès
  __cm_end();
}
```

(a) Zone critique à l'intérieur la boucle

```
__cm_start();
for(int i = 0;i < SIZE; i++){
  if(password[i] != secret[i]) break; // Mot de passe faux
  if(password[i] != secret[i]) break; // Mot de passe faux
  __cm_target();
  // Protégé : Authentifié avec succès
}
__cm_end();
```

(b) Zone critique qui englobe la boucle

FIGURE 7.5 : Contre-mesures dans une boucle

Ce déroulage ainsi que le mot clé **inline** suffisent dans notre exemple pour permettre à WP de raisonner sur le code.

Cependant, ces solutions ont des limitations : elles ne fonctionneront pas, par exemple, si le nombre maximal d'itérations d'une boucle est trop grand, ou ne peut pas être borné. En pratique, dans les sections de code critique, les boucles sans nombre d'itérations borné ne sont pas communes et il est souvent possible de déterminer le nombre maximal d'itérations (un mot de passe est lu trois fois au plus, le code secret à vérifier est de longueur finie, etc.).

Dérouler les boucles ajoute une nouvelle contrainte : que faire si la boucle déroulée contient une contre-mesure ? Nous avons choisi pour ce genre de cas de dupliquer l'ensemble de nos macros pour les faire apparaître dans chaque itération. Ainsi, si on déroule une boucle qui contient 2 conditions à muter 10 fois, alors on aura un total de 22 variables de mutations (deux par itération déroulée, plus 2 pour le corps de la boucle). On aura également 11 **check** à vérifier au lieu d'un.

Enfin, lorsqu'une boucle contient une contre-mesure, on peut se poser la question de l'emplacement des bornes de la section critique. Dans la Figure 7.5a on vérifie la contre-mesure indépendamment de la boucle, la vérification se passe ici sans problème, la contre-mesure est validée. En revanche si on consi-

```
1  __cm_start();
2  for(int i = 0;i < SIZE; i++){
3    if(!(i < SIZE)) break; // Protège d'une mutation
4    if(password[i] != secret[i]) break; // Mot de passe faux
5    if(password[i] != secret[i]) break; // Mot de passe faux
6    __cm_target();
7    // Protégé : Authentifié avec succès
8  }
9  __cm_end();
```

FIGURE 7.6 : Contre-mesures dans une boucle, avec protection sur la condition de sortie

dère que la boucle entière est critique comme dans la Figure 7.5b (on recommande cette version), ici la preuve ne passe plus et la contre-mesure n'est plus valide car la condition de la boucle n'est pas protégée. La raison n'est pas triviale : comme nous l'avons déjà mentionné, le code C est normalisé par FRAMA-C, de sorte que les boucles sont toutes transformées en boucle `while` faisant ainsi apparaître explicitement la condition de sortie sous forme d'une instruction conditionnelle. Dès lors, si la boucle entière est en zone critique, la condition de sortie sera également instrumentée par notre approche avec un déclencheur de mutation. Dans ce cas tout se passe bien jusqu'à ce qu'on essaie de muter le test de sortie lors de la dernière itération de la boucle.

Si on fait cela, alors on va atteindre la ligne 3 avec la variable `i` qui aura pour valeur `SIZE` et qui dépasse donc la taille maximale du tableau : on crée donc un comportement non défini (*undefined behavior* en anglais). Notre vérification ici ne peut donc plus fonctionner, car WP, en voyant ce comportement non défini, ne sera plus capable de prouver ce qui suit. Une solution dans ce cas est d'ajouter une contre-mesure sur la condition de sortie de la boucle comme dans la Figure 7.6. Dans ce cas la condition de sortie est protégée, on évite donc le comportement indéfini et notre contre-mesure est vérifiée.

> **Remarque 28.** Dans cet exemple, l'attaque n'a qu'un intérêt limité, car pour se produire elle nécessite au préalable de connaître le mot de passe pour atteindre la dernière itération.

```
if(password != secret || password != secret) return 1;
// Protégé : Authentifié avec succès
```

FIGURE 7.7 : Contre-mesure avec test dupliqué dans une instruction conditionnelle

7.3.3 Duplication de conditionnelle, ou duplication de condition ?

Dans les exemples que nous avons vus jusqu'à présent, on trouve toujours le même schéma : si une instruction conditionnelle **if** réalise un test (ou condition) critique, alors on duplique cette instruction (et donc la condition qui va avec). C'est ce qu'on retrouve dans les Figures 7.1 à 7.3 et 7.5. Cependant, il est possible de dupliquer uniquement la condition, sans dupliquer la conditionnelle. Pour ce faire, on introduit un opérateur binaire (par exemple ||) et on effectue deux fois le test dans une seule instruction. Si un des deux tests est valide, alors on rentre dans le bloc.

Par exemple la Figure 7.1 pourrait également s'écrire comme la Figure 7.7. Ici les deux codes sont sémantiquement équivalents : une fois compilé la condition sera séparée en deux tests : en muter un seul ne suffira donc pas pour contourner cette vérification.

Cependant, face à des contre-mesures de ce type, notre instrumentation initiale ne fonctionne plus. On ne veut pas muter le test global, mais chaque côté du || qui représente en fait le test initial dupliqué. Dans le fragment code suivant :

```
if (x < 0 || x > 10) return 1;
if (!(x >= 0) && !(x <= 10)) return 1;
// section critique
// x entre 0 et 10
```

On voit que la contre-mesure contient des opérateurs logiques qui ne sont pas utilisés comme séparation entre deux tests dupliqués comme dans la Figure 7.7. Dès lors, se pose la question : comment différencier une condition classique qui contient un opérateur binaire d'une condition dupliquée avec deux tests regroupés par un opérateur ?

Pour permettre de gérer correctement ces comportements, on fait donc l'hypothèse suivante : sans précision supplémentaire, toutes les conditions de la zone critique sont sous la forme initiale qu'on a présentée, à savoir

```
__cm_start();
// La prochaine conditionnelle contient un test dupliqué
__cm_double_if();
if(password != secret || password != secret) return 1;
__cm_target();
// Protégé : Authentifié avec succès
__cm_end();
```

FIGURE 7.8 : Contre-mesure avec test dupliqué dans une instruction conditionnelle

un test par instruction conditionnelle. On introduit une nouvelle macro, `__cm_double_if()`, qu'il faudra alors ajouter dans le cas où une condition contient en fait 2 tests séparés par un opérateur binaire comme dans la Figure 7.8.

> **Remarque 29.** Il est compliqué de gérer tous les cas possibles ici, comment faire si une condition contient trois tests, pour se prémunir de deux injections coordonnées ? Dans notre cas d'étude et dans les exemples qu'on a pu observer jusqu'à présent, cette forme n'a jamais été utilisée. On choisit donc de faire une hypothèse supplémentaire : les contre-mesures sont écrites sous une des deux formes qu'on a présentées précédemment, c'est-à-dire soit en copiant plusieurs fois une conditionnelle, soit avec une conditionnelle dont la condition ne contient que deux tests. Supporter d'autres cas est plus ou moins trivial et pourrait être ajouté à notre implémentation (en créant de nouvelles macros et en les instrumentant correctement ensuite dans notre outil), mais on peut aussi réécrire les contre-mesures pour correspondre à ces deux schémas plus courants.

> **Remarque 30.** Il est important de bien comprendre où est la zone que l'on veut protéger quand on choisit la forme des contre-mesures. On verra dans la Section 7.5 qu'une erreur est vite arrivée...

7.3.4 Contre-mesures a posteriori

La Figure 7.9 présente un cas assez particulier, rencontré lors de nos expérimentations. Ici, on a bien une contre-mesure, un peu différente des autres, mais fonctionnant sur le même principe. On fait une première vérification, si elle est correcte, alors on rentre dans la condition. À l'intérieur, on revérifie une

```
1 __cm_start();
2 if(password == secret && version == last){
3   if(password != secret || version != last) return 1;
4   __cm_target();
5   [ ... ] // Code protégé
6   if(password != secret || version != last) return 1;
7   __cm_target();
8 }
9 __cm_end();
```

FIGURE 7.9 : Exemple de contre-mesure avec vérification a posteriori

deuxième fois pour protéger contre l'injection de fautes. On exécute ensuite notre code protégé et on réalise une dernière vérification, à la fin. Ici on peut donc définir deux cibles différentes avec `__cm_target()`. La première après la ligne 3, qui vérifiera les deux premières conditions et la deuxième dans le bloc après la condition de la ligne 6.

Ce cas particulier de code protégé encadré par des contre-mesures a ajouté d'autres problèmes. Que faire du code au milieu de notre zone critique, est-ce qu'on veut également muter les conditions qu'il contient ? Est-ce que ce code peut lui-même contenir des contre-mesures ?

Pour répondre à la première question, par défaut toutes les instructions conditionnelles à part celles issues de fonctions inlinées (voir la Section 7.3.1 sur les appels de fonctions) sont instrumentées. Toutefois, muter une condition qui n'est pas une contre-mesure rendra souvent impossible notre vérification. Si la mutation a lieu dans cette condition, comme elle n'est pas protégée, alors l'exécution continuera normalement et on atteindra notre cible. Notre propriété étant "On ne peut pas atteindre ce point si une mutation a eu lieu", on sera incapable de le prouver. On ajoute une nouvelle macro, `__cm_ignore_if()` permettant à l'utilisateur de préciser dans l'instrumentation qu'il ne souhaite pas prendre en compte la mutation d'une condition. Dans la Figure 7.10, la macro à la ligne 4 permet de ne pas instrumenter l'instruction conditionnelle de débogage.

> **Remarque 31.** En faisant ce choix, l'utilisateur indique explicitement que cette condition n'est pas critique et/ou n'influence pas les contre-mesures autour. Ainsi, l'absence de contre-mesure est tracée et permet de chercher `__cm_ignore_if` pour voir où ces macros se trouvent et si l'absence de contre-mesure est justifiée (éventuellement par un commentaire).

```
__cm_start();
if(password == secret && version == last){
  if(password != secret || version != last) return 1;
  __cm_ignore_if();
  if(debug) printf("Début de la zone protégée");
  __cm_target();
  [...] // Code protégé
  if(password != secret || version != last) return 1;
  __cm_target();
}
__cm_end();
```

FIGURE 7.10 : Exemple de contre-mesure avec vérification a posteriori

```
if(password == secret && version == last){
  if(password != secret || version != last) return 1;
  // Début Code protégé
  if(check_integrity() == secfalse) return 2;
  if(check_integrity() != sectrue) return 2;
  // Code très protégé
  // Fin Code protégé
  if(password != secret || version != last) return 1;
}
```

FIGURE 7.11 : Zones critiques imbriquées

Par ailleurs, il faut également considérer le cas de sections critiques imbriquées. La Figure 7.11 illustre un cas de ce genre. Ici on aimerait pouvoir vérifier les contre-mesures de manière indépendante, notre implémentation permet donc d'imbriquer les zones critiques les unes dans les autres, autant que nécessaire, pour faire ces vérifications.

7.4 Implémentation

La technique de vérification des contre-mesures décrite précédemment a été implémenté en OCAML dans LTEST, l'ensemble d'outils utilisé par les travaux précédents, comme un nouveau critère, *Redundant Check Countermeasures* (**RCC**) ou "Contre-mesures à base de vérifications redondantes". Ce critère instrumente les contre-mesures en utilisant des annotations fournies par l'utilisateur. Le code instrumenté peut être donné à LUNCOV qui essaiera ensuite, en utilisant WP, de vérifier que le point cible défini ne peut pas être atteint par N inversions de tests.

Nous allons ici développer un exemple complet d'annotation et instrumentation d'une contre-mesure en nous basant sur le code de vérification d'intégrité de la Figure 7.2. Comme nous l'avons détaillé, notre approche nécessite une première étape d'annotation manuelle. Cette étape demande peu d'expertise et donne le code obtenu dans la Figure 7.12. L'utilisateur commence par repérer la contre-mesure, ici de la ligne 18 à la ligne 21 et l'encadre pour créer une zone critique avec les macros `__cm_start()` et `__cm_end()`. Ensuite, il définit avec `__cm_target()` le point du code qui doit être protégé par la contre-mesure. Cependant, ces annotations ne seront pas suffisantes ici. En effet, la contre-mesure fait un appel à la fonction `check_code_integrity` et comme nous l'avons détaillé dans la Section 7.3.1, nous ne serons pas capables de vérifier la Propriété 1 dans ce cas. L'utilisateur doit donc s'assurer que la fonction est inlinée pour intégrer directement le bloc de la fonction dans le corps de la fonction `main` à la place de l'appel de fonction. Enfin, la boucle de la ligne 11 va aussi mettre en difficulté notre analyse (voir la Section 7.3.2). Comme nous l'avons déjà mentionné, il faudra ici ajouter une annotation (ligne 11) pour demander à Frama-C de dérouler la boucle (on veut la dérouler `SIZE+1` fois, soit 3 ici).

Maintenant que le code est annoté, l'utilisateur peut utiliser Frama-C et LAnnotate pour instrumenter le code. La commande suivante normalise et instrumente le code de la Figure 7.12.

```
> frama-c -lannot=RCC test_integrity.c -inline-calls check_code_integrity
```

Le résultat de code obtenu est disponible en annexe dans Figure A.7. L'argument `-lannot=RCC` appelle LAnnotate et lui indique qu'il faut transformer le code pour le critère **RCC** mentionné ci-dessus et `-inline-calls check_code_integrity` indique à Frama-C que l'on souhaite inliner dans le code tous les appels à la fonction `check_code_integrity`.

Comme détaillé dans la Section 2.4, Frama-C va normaliser le code avec plusieurs normalisations (transformations de code). Pour notre exemple avec la commande donnée, cela inclut l'inlining de fonctions et le déroulage de boucles. Dans le code obtenu, on retrouve la fonction `mutated` que nous avons présenté dans la Section 7.2.1 avec le compteur de mutations et le nombre de mutations considérés par le modèle de faute (une ici). On retrouve ensuite la fonction `check_code_integrity_fc_inline` (qui a été renommé lors de la normalisation) avec comme principale différence que la boucle est déroulée trois fois (de la ligne 32 à la ligne 43 de la Figure A.7). Ensuite, la boucle apparaît normalement (sous sa forme normalisée, donc **`while`** au lieu de **`for`**). Le mot clé `"completely"` a quant à lui ajouté un invariant faux à la ligne 45, car on considère que ce code est inatteignable puisqu'on a déroulé complètement la boucle.

```
1  // valeurs vrai/faux sécurisées
2  typedef enum {secfalse = 0×55aa55aa,
3                sectrue = 0×aa55aa55} secbool;
4  #define SIZE 2
5  #define SUM 10
6  int integrity[SIZE];
7
8  // Vérification d'intégrité
9  secbool check_code_integrity(){
10   int sum = 0;
11   /*@ loop pragma UNROLL SIZE+1, "completely";*/
12   for(int i = 0; i < SIZE; i++)
13     sum += integrity[i];
14   if(sum == SUM) return sectrue;
15   return secfalse;
16 }
17
18 int main(){
19   __cm_start();
20   secbool chk1=check_code_integrity();
21   if(chk1 != sectrue) return 1; // Erreur, code compromis
22   secbool chk2=check_code_integrity();
23   if(!chk2 == sectrue) return 1; // Contre-mesure erronée
24   // Protégé : Succès de la vérification d'intégrité
25   __cm_target();
26   __cm_end();
27 }
```

FIGURE 7.12 : Annotation manuelle du code de la Figure 7.2

On retrouve le code de cette fonction inliné dans la fonction `main` (des lignes 69 à 98 et des lignes 109 à 138 de la Figure A.7). Il est précédé de l'initialisation à 0 de nos déclencheurs de mutations. Ces déclencheurs sont utilisés ensuite dans les deux conditions pour simuler une faute éventuelle (lignes 103 et 142 de la Figure A.7) d'après notre approche expliquée dans la Section 7.2.1. Enfin, un **`label`** est ajouté dans le code comme présenté dans les Chapitres 2 et 3. Ce label peut être ensuite utilisé par LUNCOV pour vérifier notre propriété comme vu dans les Chapitres 2 et 6. L'expression sera donc inversée pour lui correspondre et on pourra ainsi essayer de prouver qu'aucune mutation ne peut atteindre ce point. Ici, comme cela a été mentionné auparavant, le code contient une erreur et donc nos outils ne seront pas en mesure de vérifier cette contre-mesure. Un simple changement à la ligne 23 de la Figure 7.12 par `~chk2` (pour effectuer une négation bit-à-bit) règle l'erreur et permet de prouver la contre-mesure.

FIGURE 7.13 : Un périphérique WooKey assemblé

7.5 Cas d'étude : WooKey

WooKey [Ben+19] est un projet open-source et open-hardware développé par l'ANSSI et ayant pour objectif de fournir un prototype de périphérique de stockage USB sécurisé avec chiffrement des données, authentification utilisateur forte et résistance à certaines menaces comme en cas de perte ou vol du périphérique USB (ne pas permettre à n'importe qui de lire le contenu, protéger contre certains types d'injections de fautes, etc.). Un exemple de périphérique assemblé de WooKey est donné en Figure 7.13.

Le boîtier sur la photo est un périphérique WooKey. Il est relié à l'ordinateur par un câble USB. Pour l'utiliser, il faut insérer une carte à puce personnelle et entrer via l'écran tactile deux codes PIN différents. Les données sont alors accessibles comme une clé USB classique. Enfin, à la fin de la session, l'utilisateur re-verrouille les données via l'interface graphique et retire sa carte.

7.5.1 Architecture

WooKey utilise beaucoup de pilotes pour gérer des fonctions clés comme le SDIO [3], la cryptographie ou l'USB. Tous les composants de WooKey sont gérés par EwoK, un micronoyau ciblant les microcontrôleurs et systèmes embarqués. EwoK est implémenté en Ada/SPARK, un langage fortement typé souvent utilisé dans des domaines critiques (aviation, médical, etc.) pour assurer des propriétés de sûreté et de sécurité aux différentes applications. Ici son rôle est entre autre de gérer le contrôle d'accès aux ressources physiques, isoler strictement

3. Secure Digital Input Output, une interface sécurisée pour carte numérique.

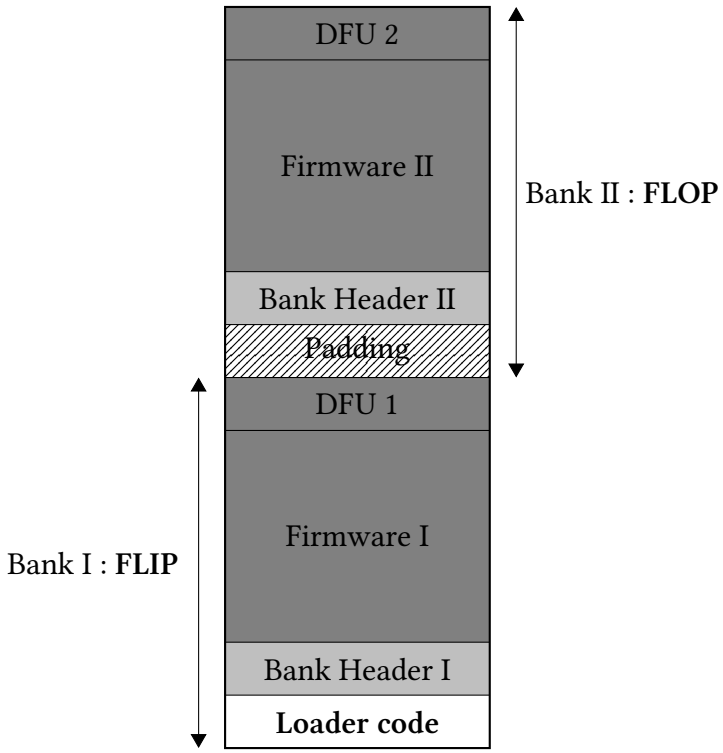


FIGURE 7.14 : Agencement de la mémoire flash avec redondance flip-flop

les tâches et les pilotes de périphériques ou encore de mettre en place le principe de moindre privilège [4].

Cependant, EwoK est contenu dans le firmware de WOOKEY et dépend donc d'un code tiers pour être exécuté : le programme d'amorçage (bootloader en anglais).

En effet, WOOKEY possède une mémoire flash qui contient tous les composants logiciels nécessaires à son fonctionnement. Cette mémoire, résumé dans la Figure 7.14, tirée de la thèse de Virgile Robles [Rob22] , est séparée en deux banques, *flip* et *flop*, qui contiennent une copie potentiellement différente du firmware, donc du micronoyau EwoK, mais aussi de l'ensemble des tâches utilisateur (appelé mode nominal). Chaque banque contient aussi sa copie du DFU (Device Firmware Update) responsable de mettre à jour le firmware (déclenché via un bouton mécanique sur le périphérique pendant la phase d'amorçage de WOOKEY) ainsi qu'un entête regroupant plusieurs informations comme le numéro de version, un drapeau (flag) stockant le statut de la dernière mise à jour et une clé de hash qui sera vérifiée par le programme d'amorçage.

4. Une tâche ne doit avoir accès qu'aux ressources dont elle a besoin pour fonctionner.

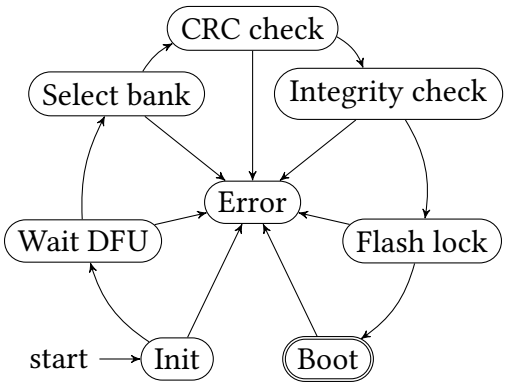


FIGURE 7.15 : Automate de la séquence d'amorçage

Ce mécanisme *flip-flop* a pour principale objectif, avec la redondance logicielle, de repérer d'éventuelles corruptions de fichiers en cas de déconnexion imprévue par exemple et de mettre en place une politique anti-retour en arrière (anti-rollback).

La seule zone non-redondante est celle qui contient le programme d'amorçage. Il ne peut pas être mis à jour et est responsable de la bonne mise en route de WooKey. Le programme d'amorçage sera chargé d'exécuter en séquence toutes les tâches nécessaires à l'amorçage correct de WooKey. La Figure 7.15 résume cette séquence. Après avoir choisi le mode de fonctionnement (DFU ou nominal), il choisira quelle banque utiliser (la plus à jour) et effectuera plusieurs étapes dont des vérifications (CRC, intégrité) avant de démarrer.

7.5.2 Vérification des sections critiques

Le programme d'amorçage de WooKey contient 11 sections critiques avec contre-mesures à base de redondance de code. Ces sections sont responsables de la bonne transition des états (de la Figure 7.15), de vérifier l'intégrité du firmware, du choix de la banque (flip ou flop) et du Contrôle de Redondance Cyclique (CRC). Nous avons considéré dans ces travaux la version du commit 00fd1c6 sur le dépôt du projet WooKey [5].

En utilisant notre approche, nous avons donc instrumenté ces morceaux de code pour définir les sections critiques, les cibles qui doivent être protégées par les contre-mesures et les cas particuliers détaillés dans la Section 7.3. Le tableau de la Figure 7.16 résume ces résultats que nous allons détailler maintenant.

La première colonne donne la position de la zone critique dans le code. Les 3 colonnes suivantes nous indiquent la présence de difficultés que nous avons

5. https ://github.com/wookey-project/bootloader/

Section critique Location (Début-Fin)	Contient		Nb de Cibles (Déroulé)	Prouvé	Analyse
	Boucles	Appel(s) de Fonction			
automaton.c :71-80	non	non	1 (13)	✓	Correct
automaton.c :383-394	non	non	1	✗	Bug
automaton.c :428-434	oui	oui	1	✓	Correct
automaton.c :457-463	oui	oui	1	✓	Correct
hash.c :87-95	non	non	1	✓	Correct
hash.c :119-130	oui	oui	1	✗	Correct ?
main.c :408-467	non	non	2	✓	Correct
main.c :421-435	non	non	2	✓	Correct
main.c :436-450	non	non	2	✓	Correct
main.c :468-495	non	non	2	✓	Correct
main.c :588-600	non	non	1	✓	Correct
Total (avant déroulage/inlining)			11		
Total (après déroulage/inlining)			27		
Prouvés			25/27		
Temps			130s		

FIGURE 7.16 : Résumé des expériences avec LTEST sur les contre-mesures de WOOKEY

détaillé dans la Section 7.3. On y trouve donc la présence de boucles, d'appels de fonctions ainsi que le nombre de cibles. Le nombre entre parenthèses nous indique si la cible se trouve dans une boucle qu'il a fallu dérouler. Dans ce cas, la cible est dupliquée pour chaque tour qu'il faut ensuite prouver séparément. Une seule contre-mesure était dans ce cas, car elle se trouve dans une fonction qui est ensuite appelée pour réaliser les contre-mesures des lignes 3 et 4 du tableau. Les contre-mesures du fichier *main* qui contiennent deux cibles sont des cas de contre-mesures a posteriori (voir la Section 7.3.4) où on veut donc vérifier plusieurs cibles différentes dans la même session critique.

On est capable de prouver l'efficacité de 9 d'entre elles en environ deux minutes. Une des deux sections non prouvées est donnée dans la Figure 7.17 avec l'instrumentation nécessaire à nos outils. Cette contre-mesure est correcte, mais trop dure à vérifier en utilisant WP sans ajouter des contrats de fonction et invariants de boucles : elle réalise trois appels de fonctions utilisant une boucle et des opérations binaires, de sorte que dérouler complètement et inliner génère des conditions de vérification trop complexes. Sa preuve en utilisant des annotations additionnelles a été reportée à des travaux futurs.

```
__cm_start();
if(!are_equal(digest, fw->fw_sig.hash, SHA256_DIGEST_SIZE)){
  goto err;
}
if(!are_equal(fw->fw_sig.hash, digest, SHA256_DIGEST_SIZE)){
  goto err;
}
if(!are_equal(digest, fw->fw_sig.hash, SHA256_DIGEST_SIZE)){
  goto err;
}
__cm_target();
__cm_end();
```

FIGURE 7.17 : Contre-mesure de WooKey trop dure à prouver de manière automatique

```
1 /* condition dupliquée */
2 if (new_state == 0×ff && !(new_state != 0×ff)) {
3   dbg_log("%s: PANIC! this should never arise!", __func__);
4   dbg_flush();
5   loader_set_state(LOADER_ERROR);
6   return;
7 }
8 //Safe code
```

FIGURE 7.18 : Contre-mesure fausse dans WooKey

La seconde cible non prouvée est montrée dans la Figure 7.18. Ce code place l'état du programme d'amorçage à `LOADER_ERROR` et sort de la fonction si la valeur de `new_state` est incorrecte. La contre-mesure est créée en faisant la conjonction de la condition initiale et d'une reformulation équivalente (voir la ligne 2) et correspond donc à une des difficultés mentionnées dans la Section 7.3. En utilisant notre méthode, nous n'avons pas été en mesure de prouver que cette contre-mesure était correcte. Et dans ce cas, elle était effectivement fausse : le double test de la ligne 2 a un effet opposé à celui recherché. Au lieu de protéger le code critique après l'instruction conditionnelle, il "protège" celui à l'intérieur, en permettant avec une seule mutation de contourner la vérification. La Figure 7.19b donne le motif correct pour une conditionnelle avec duplication de test où les occurrences de C_1 peuvent être équivalentes, mais pas nécessairement identiques, notamment pour éviter les attaques par canaux auxiliaires en donnant la position des vérifications redondantes. Ici l'erreur vient du fait que les développeurs ont utilisé l'opérateur logique ET (`&&`) au lieu de l'opérateur OU (`||`) contrairement à la Figure 7.19b. Ce type de contre-mesure utilisant des

```
/* double if protection */
if (C1 && C1) {
  // Safe code
}
// Error
```

(a) Code protégé à l'intérieur

```
/* double if protection */
if (C1 || C1) {
  // Error
}
// Safe code
```

(b) Code protégé à l'extérieur

FIGURE 7.19 : modèles de bonnes contre-mesures

opérateurs logiques au lieu d'instructions **if** successives, est utilisé plusieurs fois dans le programme d'amorçage de WOOKEY, mais c'est le seul cas où le code protégé est en dehors du bloc de la condition. Dans tous les autres cas, l'erreur est attrapée après, comme dans la Figure 7.19a.

Cette erreur a été signalée aux développeurs qui ont confirmé qu'il s'agissait bien d'une erreur. Cet exemple montre qu'il est très facile de faire une erreur dans les contre-mesures et qu'il n'est pas évident de les trouver, comme le code fonctionne correctement et qu'une injection est nécessaire pour espérer trouver une erreur. Cela confirme le besoin d'outils dédiés pour vérifier les contre-mesures. Après avoir corrigé l'erreur détectée, nous avons prouvé une implémentation correcte de cette contre-mesure. *Notre outil a donc été capable de prouver automatiquement ~90% des sections critiques dans le programme d'amorçage sans ajouter de contrats.*

Chapitre 8

Conclusion et perspectives

Sommaire

8.1 Synthèse des contributions

Dans cette thèse, nous avons présenté des solutions et méthodes permettant d'améliorer les différents processus de test logiciel, comme la création des objectifs de test, la détection des objectifs polluants, la génération de tests ou la mesure de couverture d'une suite de tests.

Les critères de couverture de code permettent de définir des objectifs de test, ayant pour principal intérêt d'obtenir une mesure concrète du degré de représentativité d'une suite de tests. Cette mesure permet en particulier de générer et minimiser une suite de tests dans le but de couvrir ces objectifs. Cependant, il existe beaucoup de critères différents, plus ou moins complexes, et les traiter de manière générique n'est pas trivial. Les critères de flot de données sont des exemples de critères très utilisés pour la validation de logiciels, mais ils sont difficiles à implémenter. Contrairement aux critères logiques classiques (tels que les instructions, les décisions ou les conditions), les objectifs de flot de données mettent en relations plusieurs points, parfois très éloignés, du programme. Les paires définition-utilisation sont des briques de base pour les objectifs de flot de données, où l'on souhaite exécuter lors des tests un chemin allant de la dé-

finition d'une variable vers une utilisation de cette dernière, sans redéfinir la variable le long du chemin.

Nous avons introduit des techniques permettant de détecter les objectifs de flot de données polluants, c'est-à-dire les objectifs gênant les différents services de test. Les objectifs non-applicables représentent la classe des objectifs pour lesquelles il n'existe pas de chemin dans le graphe de flot de contrôle permettant de valider l'objectif. Les objectifs infaisables regroupent les objectifs pour lesquelles il existe un chemin au moins, mais aucun des chemins existants n'est exécutable. Enfin, nous avons défini une notion d'équivalence entre les objectifs permettant de réduire le nombre total d'objectifs sans perte de couverture vis-à-vis du critère choisi.

Ces différents types d'objectifs polluants peuvent être détectés en utilisant plusieurs analyses : l'analyse de flot de données pour les objectifs non-applicables et les objectifs polluants, et l'interprétation abstraite et le calcul de plus faible précondition pour les objectifs infaisables. Nous avons proposé des adaptations des techniques classiques pour ce contexte.

Nous avons implémenté nos techniques, via la plateforme de vérification de code C Frama-C, dans plusieurs greffons afin de détecter des objectifs polluants dans du code C. Dans LAnnotate, un outil pour la création d'objectifs de test, nous avons ajouté le support des critères de flot de données, et utilisé une analyse de flot de données pour filtrer les objectifs non-applicables et équivalents. Ensuite, avec LUncov, un outil pour la détection d'objectifs de test polluants, nous avons mis en place une détection automatique des objectifs infaisables, en utilisant Wp et Eva, deux greffons d'analyse statique de Frama-C. De plus, nos critères ont été modélisés sous une forme permettant à des outils comme PathCrawler de générer une suite de tests couvrant les critères de flot de données.

Nous avons ensuite évalué et comparé les différentes approches dans l'objectif de déterminer une méthode avec le meilleur compromis entre nombre d'objectifs polluants détectés et temps de calcul des analyses.

Ce meilleur support des critères de flot de données nous a ensuite permis d'explorer de nouvelles possibilités de critères, en ajoutant par exemple des limites aux critères de flot de données classiques, mais également en expérimentant avec la notion de visibilité, utilisée dans le monde du test en tant qu'une heuristique supplémentaire lors de la sélection de tests pour observer l'impact concret des objectifs sur l'exécution du programme. Nous avons proposé et étudié deux approches de modélisation pour de tels objectifs de flot de données complexes, avec l'évaluation immédiate et l'évaluation retardée. Nous avons constaté que cette dernière était plus efficace pour la génération de tests.

Les applications de ce type de techniques d'analyse dépassent le domaine du test. Nous avons montré qu'il était possible d'utiliser la détection d'objectifs polluants dans le domaine de la sécurité et de l'injection de fautes pour vérifier la correction de contre-mesures. Notre approche comprend la spécification d'une fonction simulant, vis-à-vis des prouveurs automatiques, la possibilité d'injection d'un nombre défini de fautes dans le programme. Cela nous permet alors de vérifier que les contre-mesures fonctionnent comme attendu par les développeurs qui les ont écrites. Un cas d'étude nous a d'ailleurs permis de trouver une erreur dans le programme WooKey, qui avait pourtant déjà été vérifié par de nombreux organismes [ANS+20].

8.2 Travaux futurs

Comme rappelé ci-dessus, nos travaux ont conduit à des résultats très intéressants pour le test et plus généralement la validation de logiciels. Ils ont également ouvert la voie à plusieurs axes de recherche que nous allons maintenant décrire.

8.2.1 Détection d'objectifs polluants et génération de tests

Performances. On a vu dans le Chapitre 6 les performances de nos outils en fonction des techniques utilisées. Une première piste de travaux est d'améliorer ces performances, notamment pour l'interprétation abstraite et le calcul de plus faible précondition. Dans le cas de l'interprétation abstraite, nous avons utilisé les paramètres par défaut fournis par Eva pour effectuer l'analyse. Il est possible d'améliorer les performances au cas par cas sur nos benchmarks en trouvant les paramètres ayant le meilleur compromis capacité de détection / temps de calcul. On pourrait, par exemple, automatiser la paramétrisation de l'analyse par Eva en utilisant du machine learning [Cha+17]. Dans le cas de Wp, certaines options peuvent également être changées, mais d'autres optimisations peuvent être réalisées. Pour le moment, l'analyse est automatique et ne nécessite pas de spécifications ACSL. Cependant, ces spécifications sont souvent obligatoires pour permettre aux prouveurs de trouver des objectifs infaisables. Des travaux existent déjà sur la génération automatique d'invariants [Oli+16] pour les boucles, et pourraient être utilisés avant nos analyses pour améliorer leurs résultats.

Actuellement, les objectifs sont vérifiés séquentiellement un par un. Des travaux avaient été effectués pour essayer de paralléliser l'analyse en utilisant plusieurs instances de Wp [Mar+18]. Avec l'arrivée du support du multi-cœur dans les prochaines versions d'OCaml [Oca21], il sera alors possible de réimplémen-

ter en partie certains greffons pour permettre d'améliorer leurs performances en utilisant plus de ressources matérielles.

Support des hyperlabels. Dans cette thèse, nous nous sommes principalement intéressés aux critères de flot de données, et en particulier aux séquences, une des constructions d'hyperlabels. Cependant, les autres constructions des hyperlabels ne sont pas encore toutes supportées par les différentes techniques. La détection d'objectifs infaisables fonctionne pour les conjonctions et les disjonctions, mais le support des liaisons et des gardes n'est pas encore disponible. En ce qui concerne la génération de tests, seuls les labels et les séquences sont supportés pour le moment. Une piste pour de futurs travaux serait donc d'améliorer les techniques existantes pour supporter les autres constructions d'hyperlabels, et donc ainsi de supporter plus de critères de couverture de code. Dans le cas de la détection d'objectifs polluants, le support des liaisons et des gardes demandera de créer des assertions qui dépendront du critère. L'aspect générique demandera beaucoup d'efforts d'implémentation. Pour la génération de tests, la nature des liaisons (sur plusieurs traces d'exécution) impliquera également un effort d'implémentation conséquent pour, par exemple, faire deux exécutions en parallèle et les guider avec des contraintes similaires pour couvrir les labels liés tout en satisfaisant la garde.

Techniques d'analyse. Afin de détecter nos objectifs polluants, nous avons utilisé l'analyse de flot de données, l'interprétation abstraite et le calcul de plus faible précondition. On a vu dans le Chapitre 2 que d'autres techniques comme l'exécution symbolique ou le model checking pouvaient être utilisées pour cette détection. Ces techniques pourraient donc être ajoutées dans nos outils afin de donner à l'utilisateur plus de choix quant à l'analyse à utiliser. On a vu, dans le Chapitre 5 avec la visibilité, qu'il était possible de déduire une sorte d'infaisabilité d'un objectif si PathCrawler n'arrivait pas à le couvrir en explorant tous les chemins. L'infaisabilité n'est pas facile à prouver par l'absence de couverture lors de la génération, mais on pourrait combiner cette approche pour vérifier avec LUncov uniquement l'infaisabilité des objectifs non-couverts par la génération de tests. De plus, des travaux utilisant le model checking pour la détection de paires def-use infaisables ont été effectués [Su+15] et il pourrait être intéressant d'utiliser cette technique pour la comparer aux autres approches.

Nouveaux critères. Un avantage des techniques présentées dans cette thèse est la capacité de modéliser et de supporter de nouveaux critères de flot de données de façon générique à l'aide de leur représentation par des séquences. On a présenté, dans le Chapitre 5, des possibilités de raffinements de critères basés

sur le flot de données. Il existe un très grand nombre de critères, et LAnnotate en supporte à l'heure actuelle 27 différents. On a également vu dans le Chapitre 2 que des travaux avaient été effectués pour créer des nouveaux critères, comme par exemple Kolchin et al. [Kol+21 ; Kol+22] avec des critères de flot de données plus complexes, créant par exemple des chaines d'utilisations de variables ou avec la notion de paire def-use *pure*, qui interdit qu'une variable soit redéfinie avant d'exécuter la paire $[d; u]$, sauf si la redéfinition d' domine d (on passe obligatoirement par d' avant d'exécuter d). LAnnotate permet déjà d'utiliser une analyse de flot de données comme on a vu pour la détection d'objectifs d'équivalents et non-applicables, calculer la domination est donc possible. De plus, la nouvelle forme d'instrumentation des séquences permet ce genre de constructions, LAnnotate semble donc a priori en mesure de supporter ces critères avec un effort de développement relativement modéré.

8.2.2 Vérification des contre-mesures

Nous avons présenté dans le Chapitre 7 une technique, mêlant spécification, instrumentation et détection d'objectifs infaisables, afin de prouver formellement l'efficacité des contre-mesures visant à protéger des zones critiques contre certains types d'attaques matérielles en dupliquant des instructions conditionnelles. Plusieurs pistes sont envisageables pour étendre ces travaux.

Modèle de fautes. Notre modèle de fautes concerne la possibilité, par un attaquant, de modifier le résultat d'une décision en inversant le résultat d'une ou plusieurs conditions pendant l'exécution du programme. Mais il existe beaucoup d'autres modèles, et notre approche pourrait être adaptée pour certains d'entre eux. Par exemple, on pourrait considérer le modèle de fautes permettant à un attaquant de modifier la valeur de n'importe quelle variable lors d'une définition en mettant à 0 ou 1 une plage de bits donnée, ou encore de modifier la valeur de retour des fonctions de la même manière. Il faudrait, par exemple, introduire d'autres fonctions de simulation de mutations avec des spécifications légèrement différentes de notre modèle actuel. Dans le cas du type `secbool`, on pourrait vouloir que la spécification n'autorise jamais une variable de ce type à muter vers une valeur de type `secbool`. Dans le cas où on aurait plusieurs fonctions simulant des fautes, il faudrait choisir si on veut un compteur de mutations partagé (on autorise n'importe quelle mutation, et combinaison de mutations, en fonction de la taille du compteur) ou si chaque fonction de simulation doit avoir son propre compteur.

Fonction de mutation. Notre approche utilise la fonction `mutated`, et plus spécifiquement sa spécification ACSL, afin de simuler la possibilité de fautes dans le programme. Fournir une implémentation de cette fonction permettrait d'utiliser d'autres approches, comme celle de Lazart [Pot+14], permettant de générer des attaques utilisant de la génération de tests à base d'exécution symbolique dynamique.

Annexe A

Figures complémentaires

Les quatre figures suivantes illustrent des graphes de flot de contrôle Mini-C obtenus à partir de fonction C avant et après une étape d'annotation avec des objectifs de test.

```
int f(int x, int y){
int res = 0;
if(x == y || x < y)
  res = 0;
else
  res = 1;
return res;
}
```

(a) Avant instrumentation avec des labels

```
int f(int x, int y){
int res = 0;
label(1, x == y);
label(2, x != y);
label(3, x < y);
label(4, x >= y);
if(x == y || x < y)
  res = 0;
else
  res = 1;
return res;
}
```

(b) Après instrumentation avec des labels

Figure A.1 : Figure 3.2 du Chapitre 3, annotation avec le critère CC

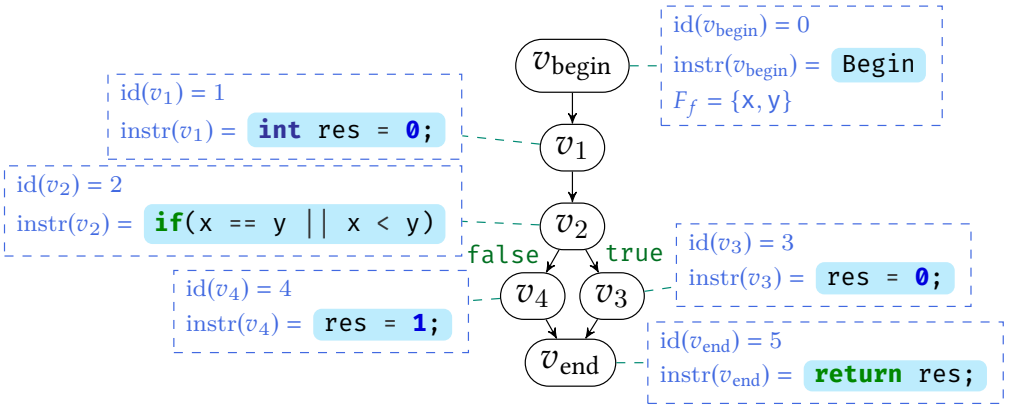


FIGURE A.2 : Graphe MINI-C du code de la Figure 3.2a (et Figure A.1a) avant instrumentation

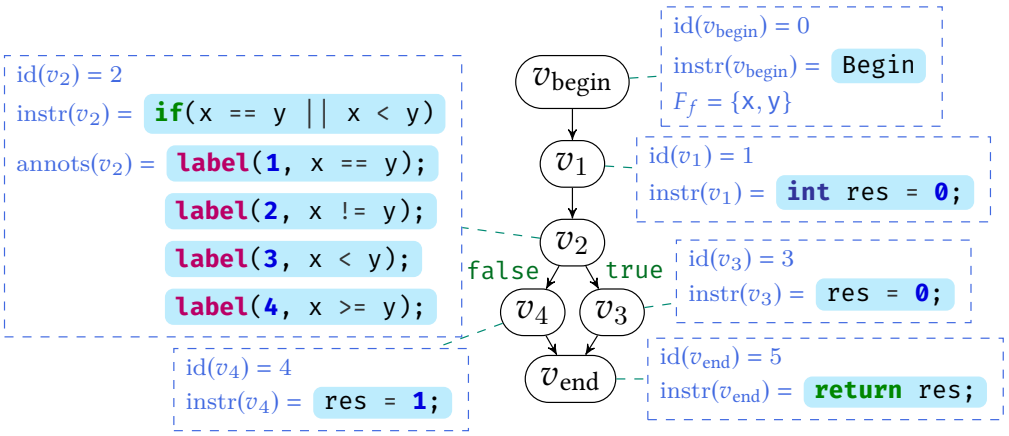


FIGURE A.3 : Graphe MINI-C du code de la Figure 3.2b (et Figure A.1b) après instrumentation

```
int f(int cond){
v1: int x = 42;
v2: if(cond){
  v3: x = x + 1;
}
v4: return x;
}
```

(a) Avant instrumentation avec des séquences

```
int f(int cond){
int status_cond_1 = 1;
int status_x_2 = 0;
int status_x_3 = 0;
int status_x_4 = 0;
int status_x_5 = 0;
v1: int x = 42;
status_x_2 = 1;
status_x_3 = 0;
status_x_4 = 1;
status_x_5 = 0;
label(1,status_cond_1 == 1);
v2: if(cond){
  label(2,status_x_2 == 1);
  label(3,status_x_3 == 1);
  v3: x = x + 1;
  status_x_2 = 0;
  status_x_3 = 1;
  status_x_4 = 0;
  status_x_5 = 1;
}
label(4,status_x_4 == 1);
label(5,status_x_5 == 1);
v4: return x;
}
```

(b) Après instrumentation avec des séquences

FIGURE A.4 : Figure 3.5 du Chapitre 3, annotation d'une fonction avec des séquences de flot de données

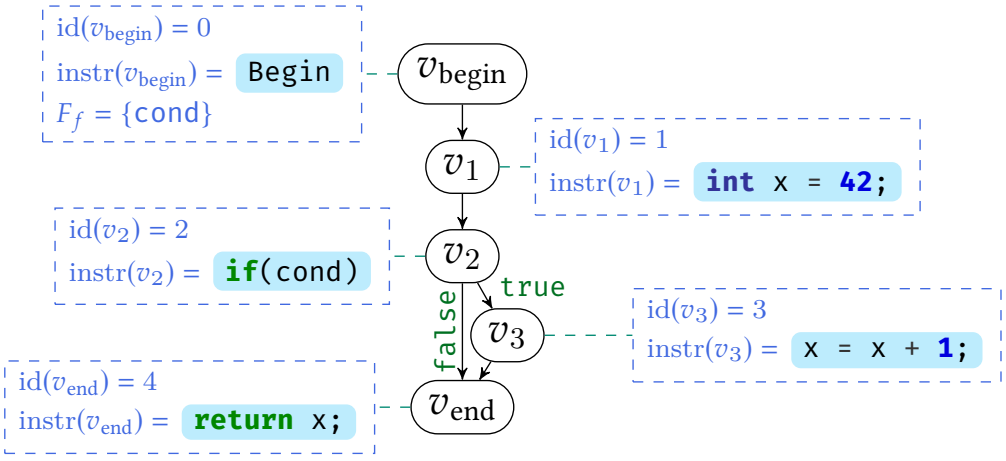


FIGURE A.5 : Graphe MINI-C du code de la Figure 3.5a (et Figure A.4a) avant instrumentation

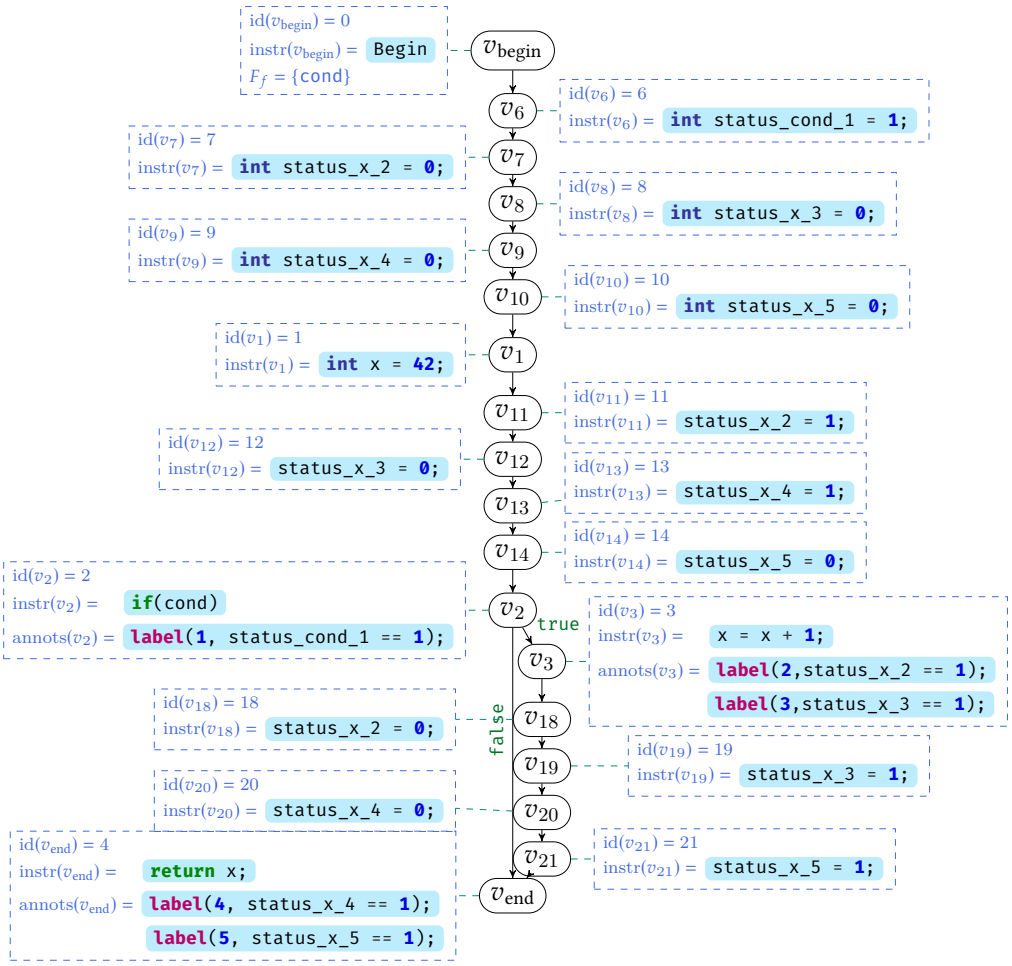


FIGURE A.6 : Graphe MINI-C du code de la Figure 3.5b (et Figure A.4b) après instrumentation

La figure suivante est le résultat de l'instrumentation du code de la Figure 7.12 avec nos outils pour la vérification de contre-mesures avec la commande suivante.

```
> frama-c -lannot=RCC test_integrity.c -inline-calls check_code_integrity
```

```
/* Generated by Frama-C LTest */
#ifndef label
#define label( ... ) do{}while(0)
#endif

#define MAX_MUTATION 1
unsigned int cpt_mutation = 0;

/*@ assigns cpt_mutation, \result;
behavior can_mutate:
  assumes cpt_mutation < MAX_MUTATION;
  ensures \result ⟺
            cpt_mutation == \at(cpt_mutation, Pre) + 1;
  ensures !\result ⟺
            cpt_mutation == \at(cpt_mutation, Pre);
behavior cant_mutate:
  assumes cpt_mutation ⩾ MAX_MUTATION;
  ensures !\result;
  ensures cpt_mutation == \at(cpt_mutation, Pre);
complete behaviors;
disjoint behaviors;
*/
int mutated(void);
enum __anonenum_secbool_1 {secfalse = 0×55aa55aa,
                           sectrue = 0×aa55aa55};
typedef enum __anonenum_secbool_1 secbool;

int integrity[2];
__inline static secbool check_code_integrity_fc_inline(void){
  secbool __retres;
  int sum = 0;
  int i = 0; // Début du déroulage
  if (! (i < 2)) goto unrolling_2_loop; // 1ère itération
  sum += integrity[i];
  i ++;
  unrolling_5_loop: ;
  if (! (i < 2)) goto unrolling_2_loop; // 2ème itération
  sum += integrity[i];
  i ++;
  unrolling_4_loop: ;
  if (! (i < 2)) goto unrolling_2_loop; // 3ème itération
```

```
42    sum += integrity[i];
43    i ++;
44    unrolling_3_loop: ;
45    /*@ loop invariant \false;
46        loop pragma UNROLL 2 + 1, "completely";
47        loop pragma UNROLL "done", 3;
48    */
49    while (i < 2) {
50      sum += integrity[i];
51      i ++;
52    }
53    unrolling_2_loop: ;
54    if (sum == 10) {
55      __retres = sectrue;
56      goto return_label;
57    }
58    __retres = secfalse;
59    return_label: return __retres;
60  }
61
62  int main(void){
63    secbool __inline_tmp_9, __inline_tmp;
64    int __retres;
65    int lannot_mut_1 = 0;
66    int lannot_mut_2 = 0;
67    // Début de l'inlining du premier appel de
68    // la fonction check_code_integrity
69    secbool __retres_8;
70    int sum = 0;
71    int i = 0;
72    if (! (i < 2)) goto unrolling_7_loop;
73    sum += integrity[i];
74    i ++;
75    unrolling_10_loop: ;
76    if (! (i < 2)) goto unrolling_7_loop;
77    sum += integrity[i];
78    i ++;
79    unrolling_9_loop: ;
80    if (! (i < 2)) goto unrolling_7_loop;
81    sum += integrity[i];
82    i ++;
83    unrolling_8_loop: ;
84    /*@ loop invariant \false;
85        loop pragma UNROLL 2 + 1, "completely";
86        loop pragma UNROLL "done", 3;
87    */
88    while (i < 2) {
89      sum += integrity[i];
90      i ++;
```

```
91    }
92    unrolling_7_loop: ;
93    if (sum == 10) {
94      __retres_8 = sectrue;
95      goto return_label_0;
96    }
97    __retres_8 = secfalse;
98    return_label_0: __inline_tmp = __retres_8;
99    // Fin de l'inlining de la fonction check_code_integrity
100   secbool chk1 = __inline_tmp; //Récupération du résultat
101   lannot_mut_1 = mutated();
102   if (lannot_mut_1 && ! (chk1 != 0×aa55aa55) ||
103     ! lannot_mut_1 && chk1 != 0×aa55aa55){
104     __retres = 1;
105     goto return_label;
106   }
107   // Début de l'inlining du 2ème appel de
108   // la fonction check_code_integrity
109   secbool __retres_13;
110   int sum_11 = 0;
111   int i_12 = 0;
112   if (! (i_12 < 2)) goto unrolling_12_loop;
113   sum_11 += integrity[i_12];
114   i_12 ++;
115   unrolling_15_loop: ;
116   if (! (i_12 < 2)) goto unrolling_12_loop;
117   sum_11 += integrity[i_12];
118   i_12 ++;
119   unrolling_14_loop: ;
120   if (! (i_12 < 2)) goto unrolling_12_loop;
121   sum_11 += integrity[i_12];
122   i_12 ++;
123   unrolling_13_loop: ;
124   /*@ loop invariant \false;
125       loop pragma UNROLL 2 + 1, "completely";
126       loop pragma UNROLL "done", 3;
127   */
128   while (i_12 < 2) {
129     sum_11 += integrity[i_12];
130     i_12 ++;
131   }
132   unrolling_12_loop: ;
133   if (sum_11 == 10) {
134     __retres_13 = sectrue;
135     goto return_label_1;
136   }
137   __retres_13 = secfalse;
138   return_label_1: __inline_tmp_9 = __retres_13;
139   // Fin de l'inlining de la fonction check_code_integrity
```

```
140   secbool chk2 = __inline_tmp_9; //Récupération du résultat
141   lannot_mut_2 = mutated();
142   if(lannot_mut_2 && !((unsigned int)(!chk2) == 0×aa55aa55)||
143     !lannot_mut_2 && (unsigned int)(!chk2) == 0×aa55aa55) {
144     __retres = 1;
145     goto return_label;
146   }
147   label(lannot_mut_1 || lannot_mut_2,1,"RCC");
148   __retres = 0;
149   return_label: return __retres;
150 }
```

FIGURE A.7 : Instrumentation du code de la Figure 7.12 avec FRAMA-C et LANNOTATE

Bibliographie

[Gen92] General Accounting Office Washington DC Information Management And Technology Division. *Patriot missile defense : Software Problem Led to System Failure at Dhahran, Arabie Saoudite*. 1992.

[Lio+96] Jacques-Louis Lions, Lennart Luebeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle et Colin O'Halloran. *Ariane 5 flight 501 failure report by the inquiry board*. 1996.

[Dur+14] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer et Vern Paxson. « The Matter of Heartbleed ». In : *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. Sous la dir. de Carey Williamson, Aditya Akella et Nina Taft. ACM, 2014, p. 475-488. doi : 10.1145/2663716.2663755.

[Lee+15] In Lee et Kyoochun Lee. « The Internet of Things (IoT) : Applications, investments, and challenges for enterprises ». In : *Business horizons* 58.4 (2015), p. 431-440. doi : https://doi.org/10.1016/j.bushor.2015.03.008.

[Wik22] Wikipedia, The Free Encyclopedia. *List of software bugs*. https://en.wikipedia.org/wiki/List_of_software_bugs. Consulté le 15 juin 2022.

[Tur37] A. M. Turing. « On Computable Numbers, with an Application to the Entscheidungsproblem ». In : *Proceedings of the London Mathematical Society* (1937).

[Ric53] Henry Gordon Rice. « Classes of recursively enumerable sets and their decision problems ». In : *Transactions of the American Mathematical society* 74.2 (1953), p. 358-366. doi : 10.2307/1990888.

[Hoa69] C. A. R. Hoare. « An Axiomatic Basis for Computer Programming ». In : *Commun. ACM* 12.10 (1969), p. 576-580. doi : 10.1145/363235.363259.

[Cou+77] Patrick Cousot et Radhia Cousot. « Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints ». In : *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Sous la dir. de Robert M. Graham, Michael A. Harrison et Ravi Sethi. ACM, 1977, p. 238-252. doi : 10.1145/512950.512973.

[Kin76] James C. King. « Symbolic Execution and Program Testing ». In : *Commun. ACM* 19.7 (1976), p. 385-394. doi : 10.1145/360248.360252.

[Amm+16] Paul Ammann et Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[Bar+14a] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye et Nikolai Kosmatov. « An All-in-One Toolkit for Automated White-Box Testing ». In : *Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014. Proceedings*. Sous la dir. de Martina Seidl et Nikolai Tillmann. T. 8570. Lecture Notes in Computer Science. Springer, 2014, p. 53-60. doi : 10.1007/978-3-319-09099-3_4.

[Mar+17a] Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov et Virgile Prevosto. « Generic and Effective Specification of Structural Test Objectives ». In : *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, p. 436-441. doi : 10.1109/ICST.2017.48.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles et Boris Yakobowski. « Frama-C : A software analysis perspective ». In : *Formal Aspects Comput.* 27.3 (2015), p. 573-609. doi : 10.1007/s00165-014-0326-7.

[Bau+21a] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles et Nicky Williams. « The dogged pursuit of bug-free C programs : the Frama-C software analysis platform ». In : *Commun. ACM* 64.8 (2021), p. 56-68. doi : 10.1145/3470569.

[Wil+05] Nicky Williams, Bruno Marre, Patricia Mouy et Muriel Roger. « PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis ». In : *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*. Sous la dir. de Mario Dal Cin, Mohamed Kaâniche et András Pataricza. T. 3463. Lecture Notes in Computer Science. Springer, 2005, p. 281-292. doi : 10.1007/11408901_21.

[Mar+17b] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov et Virgile Prevosto. « Taming Coverage Criteria Heterogeneity with LTest ». In : *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 2017, p. 500-507. doi : 10.1109/ICST.2017.57.

[Mar+20] Thibault Martin, Nikolai Kosmatov, Virgile Prevosto et Matthieu Lemerre. « Detection of Polluting Test Objectives for Dataflow Criteria ». In : *Integrated Formal Methods - 16th International Conference, IFM 2020, Lugano, Switzerland, November 16-20, 2020, Proceedings*. Sous la dir. de Brijesh Dongol et Elena Troubitsyna. T. 12546. Lecture Notes in Computer Science. Springer, 2020, p. 337-345. doi : 10.1007/978-3-030-63461-2_18.

[Mar+21] Thibault Martin, Nikolai Kosmatov, Virgile Prevosto et Matthieu Lemerre. « Détection d'objectifs de test polluants pour les critères de flot de données ». In : *20èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels* (2021), p. 15.

[Mar+22] Thibault Martin, Nikolai Kosmatov et Virgile Prevosto. « Verifying redundant-check based countermeasures : a case study ». In : *SAC '22 : The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*. Sous la dir. de Jiman Hong, Miroslav Bures, Juw Won Park et Tomás Cerný. ACM, 2022, p. 1849-1852. doi : 10.1145/3477314.3507341.

[Bux+70] John N Buxton et Brian Randell. *Software Engineering Techniques : Report of a conference sponsored by the NATO Science Committee*. 1970.

[Rap+82] Sandra Rapps et Elaine J. Weyuker. « Data Flow Analysis Techniques for Test Data Selection ». In : *Proceedings, 6th International Conference on Software Engineering, Tokyo, Japan, September 13-16, 1982*. Sous la dir. d'Yutaka Ohno, Victor R. Basili, Hajime Enomoto, Koji Kobayashi et Raymond T. Yeh. IEEE Computer Society, 1982, p. 272-278.

[Amm+08] Paul Ammann et Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. doi : 10 . 1017 / CBO9780511809163.

[Mar+18] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto et Loïc Correnson. « Time to clean your test objectives ». In : *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Sous la dir. de Michel Chaudron, Ivica Crnkovic, Marsha Chechik et Mark Harman. ACM, 2018, p. 456-467. doi : 10.1145/3180155.3180191.

[Amm+03] Paul Ammann, A. Jefferson Offutt et Hong Huang. « Coverage Criteria for Logical Expressions ». In : *14th International Symposium on Software Reliability Engineering (ISSRE 2003), 17-20 November 2003, Denver, CO, USA*. IEEE Computer Society, 2003, p. 99-107. doi : 10.1109/ISSRE.2003.1251034.

[Hem15] Hadi Hemmati. « How Effective Are Code Coverage Criteria ? » In : *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*. IEEE, 2015, p. 151-156. doi : 10.1109/QRS.2015.30.

[Fra+93] Phyllis G. Frankl et Stewart N. Weiss. « An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing ». In : *IEEE Trans. Software Eng.* 19.8 (1993), p. 774-787. doi : 10.1109/32.238581.

[Fra+97] Phyllis G. Frankl, Stewart N. Weiss et Cang Hu. « All-uses vs mutation testing : An experimental comparison of effectiveness ». In : *J. Syst. Softw.* 38.3 (1997), p. 235-253. doi : 10 . 1016 / S0164 - 1212(96)00154-9.

[And+06] James H. Andrews, Lionel C. Briand, Yvan Labiche et Akbar Siami Namin. « Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria ». In : *IEEE Trans. Software Eng.* 32.8 (2006), p. 608-624. doi : 10.1109/TSE.2006.83.

[Yan+09] Qian Yang, J. Jenny Li et David M. Weiss. « A Survey of Coverage-Based Testing Tools ». In : *Comput. J.* 52.5 (2009), p. 589-597. doi : 10.1093/comjnl/bxm021.

[Chi+94] John Chilenski et Philip Newcomb. « Formal Specification Tools for Test Coverage Analysis ». In : *Proceedings KBSE'94, the Ninth Knowledge-Based Software Engineering Conference, Monterey, California, USA, September 20-23, 1994*. IEEE Computer Society, 1994, p. 59-68. doi : 10.1109/KBSE.1994.342677.

[Bul09] Bullseye. *Bullseye Testing Technology : BullseyeCoverage*. http://bullseye.com/. 2009.

[Gco] Gcov. *GCC's Gcov*. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[Ldr] Ldracover. *LDRA – LDRACover*. https://ldra.com/products/ldracover/.

[Chi+01] John Joseph Chilenski et al. *Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Rapp. tech. United States. Federal Aviation Administration. Office of Aviation Research, 2001.

[Tes] Testwell. *Testwell CTC++ : Test Coverage Analyzer for C/C++*. http://www.testwell.fi/ctcdesc.html.

[Rap+85] Sandra Rapps et Elaine J. Weyuker. « Selecting Software Test Data Using Data Flow Information ». In : *IEEE Trans. Software Eng.* 11.4 (1985), p. 367-375. doi : 10.1109/TSE.1985.232226.

[Fra+88] Phyllis G. Frankl et Elaine J. Weyuker. « An Applicable Family of Data Flow Testing Criteria ». In : *IEEE Trans. Software Eng.* 14.10 (1988), p. 1483-1498. doi : 10.1109/32.6194.

[Din+12] Sun Ding et Hee Beng Kuan Tan. « Detection of Infeasible Paths : Approaches and Challenges ». In : *Evaluation of Novel Approaches to Software Engineering - 7th International Conference, ENASE 2012, Warsaw, Poland, June 29-30, 2012, Revised Selected Papers*. Sous la dir. de Leszek A. Maciaszek et Joaquim Filipe. T. 410. Communications in Computer and Information Science. Springer, 2012, p. 64-78. doi : 10.1007/978-3-642-45422-6_5.

[Alt96] Peter Altenbernd. « On the False Path Problem in Hard Real-Time Programs ». In : *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, RTS 1996, L'Aquila, Italy, June 12-14, 1996*. IEEE Computer Society, 1996, p. 102-107. doi : 10.1109/EMWRTS.1996.557827.

[Che+05] Ting Chen, Tulika Mitra, Abhik Roychoudhury et Vivy Suhendra. « Exploiting Branch Constraints without Exhaustive Path Enumeration ». In : *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 5, 2005, Palma de Mallorca, Spain*. Sous la dir. de Reinhard Wilhelm. T. 1. OASIcs. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[Su+15] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He et Zhendong Su. « Combining Symbolic Execution and Model Checking for Data Flow Testing ». In : *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Sous la dir. d'Antonia Bertolino, Gerardo Canfora et Sebastian G. Elbaum. IEEE Computer Society, 2015, p. 654-665. doi : 10.1109/ICSE.2015.81.

[Flo67] Robert W Floyd. « Assigning meanings to programs ». In : *Program Verification*. Springer, 1967, p. 65-81. doi : 10.1007/978-94-011-1793-7_4.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Bor00] Richard Bornat. « Proving Pointer Programs in Hoare Logic ». In : *Mathematics of Program Construction, 5th International Conference, MPC 2000, Ponte de Lima, Portugal, July 3-5, 2000, Proceedings*. Sous la dir. de Roland Carl Backhouse et José Nuno Oliveira. T. 1837. Lecture Notes in Computer Science. Springer, 2000, p. 102-126. doi : 10.1007/10722010_8.

[Fla+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe et Raymie Stata. « Extended Static Checking for Java ». In : *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. Sous la dir. de Jens Knoop et Laurie J. Hendren. ACM, 2002, p. 234-245. doi : 10.1145/512529.512558.

[Bec+07] Bernhard Beckert, Reiner Hähnle et Peter H. Schmitt, éd. *Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino*. T. 4334. Lecture Notes in Computer Science. Springer, 2007. doi : 10.1007/978-3-540-69061-0.

[Bar03] John Gilbert Presslie Barnes. *High integrity software : the spark approach to safety and security : sample chapters*. Pearson Education, 2003.

[Com+96] Joseph J. Comuzzi et Johnson M. Hart. « Program Slicing Using Weakest Preconditions ». In : *FME '96 : Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18-22, 1996, Proceedings*. Sous la dir. de Marie-Claude Gaudel et Jim Woodcock. T. 1051. Lecture Notes in Computer Science. Springer, 1996, p. 557-575. doi : 10.1007/3-540-60973-3_107.

[Ran+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau et Dominique Schoen. « Applying Formal Proof Techniques to Avionics Software : A Pragmatic Approach ». In : *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume II*. Sous la dir. de Jeannette M. Wing, Jim Woodcock et Jim Davies. T. 1709. Lecture Notes in Computer Science. Springer, 1999, p. 1798-1815. doi : 10.1007/3-540-48118-4_45.

[Bau+22] Patrick Baudin, François Bobot, Loïc Correnson et Zaynah Dargaye. *WP plugin manual*. http://frama-c.com/download/frama-c-wp-manual.pdf. 2022.

[Her76] P. M. Herman. « A Data Flow Analysis Approach to Program Testing ». In : *Aust. Comput. J.* 8.3 (1976), p. 92-96.

[All+76] Frances E. Allen et John Cocke. « A Program Data Flow Analysis Procedure ». In : *Commun. ACM* 19.3 (1976), p. 137-147. doi : 10.1145/360018.360025.

[Fos+76] Lloyd D. Fosdick et Leon J. Osterweil. « Data Flow Analysis in Software Reliability ». In : *ACM Comput. Surv.* 8.3 (1976), p. 305-330. doi : 10.1145/356674.356676.

[All70] Frances E. Allen. « Control flow analysis ». In : *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*. Sous la dir. de Robert S. Northcote. ACM, 1970, p. 1-19. doi : 10.1145/800028.808479.

[Su+17] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen et Zhendong Su. « A Survey on Data-Flow Testing ». In : *ACM Comput. Surv.* 50.1 (2017), 5 :1-5 :35. doi : 10.1145/3020266.

[Har+89] Mary Jean Harrold et Mary Lou Soffa. « Interprocedual Data Flow Testing ». In : *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification, TAV 1989, Key West, Florida, USA, December 13-15, 1989*. Sous la dir. de Richard A. Kemmerer. ACM, 1989, p. 158-167. doi : 10.1145/75308.75327.

[Cla+89] Lori A. Clarke, Andy Podgurski, Debra J. Richardson et Steven J. Zeil. « A Formal Evaluation of Data Flow Path Selection Criteria ». In : *IEEE Trans. Software Eng.* 15.11 (1989), p. 1318-1332. doi : 10.1109/32.41326.

[Wey90] Elaine J. Weyuker. « The Cost of Data Flow Testing : An Empirical Study ». In : *IEEE Trans. Software Eng.* 16.2 (1990), p. 121-128. doi : 10.1109/32.44376.

[Kol+21] Alexander Kolchin, Stepan Potiyenko et Thomas Weigert. « Extending data flow coverage with redefinition analysis ». In : *International Conference on Information and Digital Technologies, IDT 2021, Zilina, Slovakia, June 22-24, 2021*. IEEE, 2021, p. 293-296. doi : 10.1109/IDT52577.2021.9497535.

[Kol+22] Alexander Kolchin et Stepan Potiyenko. « Extending Data Flow Coverage to Test Constraint Refinements ». In : *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings*. Sous la dir. de Maurice H. ter Beek et Rosemary Monahan. T. 13274. Lecture Notes in Computer Science. Springer, 2022, p. 313-321. doi : 10.1007/978-3-031-07727-2_17.

[Pol99] Polyspace. *Polyspace Verifier*. https://www.mathworks.com/products/polyspace.html. 1999.

[Cou+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival. « The ASTREÉ Analyzer ». In : *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of*

Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Sous la dir. de Shmuel Sagiv. T. 3444. Lecture Notes in Computer Science. Springer, 2005, p. 21-30. DOI : 10.1007/978-3-540-31987-0_3.

[Bla+17] Sandrine Blazy, David Bühler et Boris Yakobowski. « Structuring Abstract Interpreters Through State and Value Abstractions ». In : *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. Sous la dir. d'Ahmed Bouajjani et David Monniaux. T. 10145. Lecture Notes in Computer Science. Springer, 2017, p. 112-130. DOI : 10.1007/978-3-319-52234-0_7.

[Our15] Alain Ourghanlian. « Evaluation of static analysis tools used to assess software important to nuclear power plant safety ». In : *Nuclear Engineering and Technology* 47.2 (2015). Special Issue on ISOFIC/ISSNP2014, p. 212-218. DOI : https://doi.org/10.1016/j.net.2014.12.009.

[Eba+19] Arnaud Ebalard, Patricia Mouy et Ryad Benadjila. « Journey to a RTE-free X.509 parser ». In : *Symposium sur la sécurité des technologies de l'information et des communications*. Rennes, 2019.

[inf] infer. *Infer Static Analyzer*. https://fbinfer.com/.

[Dis+06] Dino Distefano, Peter W. O'Hearn et Hongseok Yang. « A Local Shape Analysis Based on Separation Logic ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*. Sous la dir. d'Holger Hermanns et Jens Palsberg. T. 3920. Lecture Notes in Computer Science. Springer, 2006, p. 287-302. DOI : 10.1007/11691372_19.

[Cla76] Lori A. Clarke. « A program testing system ». In : *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*. Sous la dir. de John A. Gosden et Olin G. Johnson. ACM, 1976, p. 488-491. DOI : 10.1145/800191.805647.

[Boy+75] Robert S. Boyer, Bernard Elspas et Karl N. Levitt. « SELECT - a formal system for testing and debugging programs by symbolic execution ». In : *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*. Sous la dir. de Martin L. Shooman et Raymond T. Yeh. ACM, 1975, p. 234-245. DOI : 10.1145/800027.808445.

[How77] William E. Howden. « Symbolic Testing and the DISSECT Symbolic Evaluation System ». In : *IEEE Trans. Software Eng.* 3.4 (1977), p. 266-278. doi : 10.1109/TSE.1977.231144.

[Sen07] Koushik Sen. « Concolic testing ». In : *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. Sous la dir. de R. E. Kurt Stirewalt, Alexander Egyed et Bernd Fischer. ACM, 2007, p. 571-572. doi : 10.1145/1321631.1321746.

[Maj+07] Rupak Majumdar et Koushik Sen. « Hybrid Concolic Testing ». In : *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, p. 416-426. doi : 10.1109/ICSE.2007.41.

[Sen+05] Koushik Sen, Darko Marinov et Gul Agha. « CUTE : a concolic unit testing engine for C ». In : *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. Sous la dir. de Michel Wermelinger et Harald C. Gall. ACM, 2005, p. 263-272. doi : 10.1145/1081706.1081750.

[Sen+06] Koushik Sen et Gul Agha. « CUTE and jCUTE : Concolic Unit Testing and Explicit Path Model-Checking Tools ». In : *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Sous la dir. de Thomas Ball et Robert B. Jones. T. 4144. Lecture Notes in Computer Science. Springer, 2006, p. 419-423. doi : 10.1007/11817963_38.

[God+05] Patrice Godefroid, Nils Klarlund et Koushik Sen. « DART : directed automated random testing ». In : *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. Sous la dir. de Vivek Sarkar et Mary W. Hall. ACM, 2005, p. 213-223. doi : 10.1145/1065010.1065036.

[Cad+08] Cristian Cadar, Daniel Dunbar et Dawson R. Engler. « KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs ». In : *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Sous la dir. de Richard Draves et Robbert van Renesse. USENIX Association, 2008, p. 209-224.

[God+12] Patrice Godefroid, Michael Y. Levin et David A. Molnar. « SAGE : Whitebox Fuzzing for Security Testing ». In : t. 10. 1. 2012, p. 20. doi : 10.1145/2090147.2094081.

[Hsu+97] Mei-Chen Hsueh, Timothy K. Tsai et Ravishankar K. Iyer. « Fault Injection Techniques and Tools ». In : *Computer* 30.4 (1997), p. 75-82. doi : 10.1109/2.585157.

[Zia+04] Haissam Ziade, Rafic A. Ayoubi et Raoul Velazco. « A Survey on Fault Injection Techniques ». In : *Int. Arab J. Inf. Technol.* 1.2 (2004), p. 171-186.

[Arl+90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-Claude Laprie, Eliane Martins et David Powell. « Fault Injection for Dependability Validation : A Methodology and Some Applications ». In : *IEEE Trans. Software Eng.* 16.2 (1990), p. 166-182. doi : 10.1109/32.44380.

[Kar+94] Johan Karlsson, Peter Lidén, Peter Dahlgren, Rolf Johansson et Ulf Gunneflo. « Using heavy-ion radiation to validate fault-handling mechanisms ». In : *IEEE Micro* 14.1 (1994), p. 8-23. doi : 10.1109/40.259894.

[Bar+12] Alessandro Barenghi, Luca Breveglieri, Israel Koren et David Naccache. « Fault Injection Attacks on Cryptographic Devices : Theory, Practice, and Countermeasures ». In : *Proc. IEEE* 100.11 (2012), p. 3056-3076. doi : 10.1109/JPROC.2012.2188769.

[Bha+14] Shivam Bhasin, Paolo Maistri et Francesco Regazzoni. « Malicious wave : A survey on actively tampering using electromagnetic glitch ». In : *2014 International Symposium on Electromagnetic Compatibility, Tokyo*. IEEE, 2014, p. 318-321.

[Che+21] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald et Flavio D. Garcia. « VoltPillager : Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface ». In : *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Sous la dir. de Michael Bailey et Rachel Greenstadt. USENIX Association, 2021, p. 699-716.

[Mur+20] Kit Murdock, David F. Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss et Frank Piessens. « Plundervolt : Software-based Fault Injection Attacks against Intel SGX ». In : *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May*

18-21, 2020. IEEE, 2020, p. 1466-1482. DOI : 10.1109/SP40000.2020.00057.

[Car+98] João CARREIRA, Henrique MADEIRA et João Gabriel SILVA. « Xception : A Technique for the Experimental Evaluation of Dependability in Modern Computers ». In : *IEEE Trans. Software Eng.* 24.2 (1998), p. 125-136. DOI : 10.1109/32.666826.

[Ben+98] Alfredo BENSO, Paolo PRINETTO, Maurizio REBAUDENGO et Matteo Sonza REORDA. « EXFI : a low-cost fault injection system for embedded microprocessor-based boards ». In : *ACM Trans. Design Autom. Electr. Syst.* 3.4 (1998), p. 626-634. DOI : 10.1145/296333.296351.

[Kan+95] Ghani A. KANAWATI, Nasser A. KANAWATI et Jacob A. ABRAHAM. « FERRARI : A Flexible Software-Based Fault and Error Injection System ». In : *IEEE Trans. Computers* 44.2 (1995), p. 248-260. DOI : 10.1109/12.364536.

[Jen+94] Eric JENN, Jean ARLAT, Marcus RIMÉN, Joakim OHLSSON et Johan KARLSSON. « Fault Injection into VHDL Models : The MEFISTO Tool ». In : *Digest of Papers : FTCS/24, The Twenty-Fourth Annual International Symposium on Fault-Tolerant Computing, Austin, Texas, USA, June 15-17, 1994*. IEEE Computer Society, 1994, p. 66-75. DOI : 10.1109/FTCS.1994.315656.

[Fol+98] Peter FOLKESSON, Sven SVENSSON et Johan KARLSSON. « A Comparison of Simulation Based and Scan Chain Implemented Fault Injection ». In : *Digest of Papers : FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23-25, 1998*. IEEE Computer Society, 1998, p. 284-293. DOI : 10.1109/FTCS.1998.689479.

[Rim+93] M. RIMÉN, J. OHLSSON, J. KARLSSON, E. JENN et J. ARLAT. « Design guidelines of a VHDL-based simulation tool for the validation of fault tolerance ». In : *Proc. 1st ESPRIT Basic Research Project PDCS-2 Open Workshop*. Citeseer. 1993, p. 461-483.

[Pot+14] Marie-Laure POTET, Laurent MOUNIER, Maxime PUYS et Louis DUREUIL. « Lazart : A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections ». In : *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, p. 213-222. DOI : 10.1109/ICST.2014.34.

[Chr+13] Maria Christofi, Boutheina Chetali, Louis Goubin et David Vigilant. « Formal verification of a CRT-RSA implementation against fault attacks ». In : *J. Cryptogr. Eng.* 3.3 (2013), p. 157-167. doi : 10.1007/s13389-013-0049-3.

[Hey+19] Karine Heydemann, Jean-François Lalande et Pascal Berthomé. « Formally verified software countermeasures for control-flow integrity of smart card C code ». In : *Comput. Secur.* 85 (2019), p. 202-224. doi : 10.1016/j.cose.2019.05.004.

[Lac+21] Guilhem Lacombe, David Féliot, Étienne Boespflug et Marie-Laure Potet. « Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities ». In : *International Workshop on Security Proofs for Embedded Systems (PROOFS)*. 2021.

[ANS+20] ANSSI, Amossys, EDSI, LETI, Lexfo, Oppida, Quarkslab, SERMA, Synacktiv, Thales et Trusted Labs. « Inter-CESTI : Methodological and Technical Feedbacks on Hardware Devices Evaluations ». In : *SSTIC*. 2020.

[Bar+14b] Sébastien Bardin, Nikolai Kosmatov et François Cheynier. « Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria ». In : *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 2014, p. 173-182. doi : 10.1109/ICST.2014.30.

[Nec+02] George C. Necula, Scott McPeak, Shree Prakash Rahul et Westley Weimer. « CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs ». In : *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Sous la dir. de R. Nigel Horspool. T. 2304. Lecture Notes in Computer Science. Springer, 2002, p. 213-228. doi : 10.1007/3-540-45937-5_16.

[Bau+21b] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filiâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*. http://frama-c.com/download/acsl.pdf. 2021.

[Cor14] Loïc Correnson. « Qed. Computing What Remains to Be Proved ». In : *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*.

Sous la dir. de Julia M. Badger et Kristin Yvonne Rozier. T. 8430. Lecture Notes in Computer Science. Springer, 2014, p. 215-229. doi : 10.1007/978-3-319-06200-6_17.

[Con+18] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala et Alain Mebsout. « Alt-Ergo 2.2 ». In : *SMT Workshop : International Workshop on Satisfiability Modulo Theories*. 2018.

[Mou+08] Leonardo Mendonça de Moura et Nikolaj S. Bjørner. « Z3 : An Efficient SMT Solver ». In : *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Sous la dir. de C. R. Ramakrishnan et Jakob Rehof. T. 4963. Lecture Notes in Computer Science. Springer, 2008, p. 337-340. doi : 10.1007/978-3-540-78800-3_24.

[Bar+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds et Cesare Tinelli. « CVC4 ». In : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Sous la dir. de Ganesh Gopalakrishnan et Shaz Qadeer. T. 6806. Lecture Notes in Computer Science. Springer, 2011, p. 171-177. doi : 10.1007/978-3-642-22110-1_14.

[Wil+04] Nicky Williams, Bruno Marre et Patricia Mouy. « On-the-Fly Generation of K-Path Tests for C Functions ». In : *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*. IEEE Computer Society, 2004, p. 290-293. doi : 10.1109/ASE.2004.10020.

[Kos08] Nikolai Kosmatov. « All-Paths Test Generation for Programs with Internal Aliases ». In : *19th International Symposium on Software Reliability Engineering (ISSRE 2008), 11-14 November 2008, Seattle/Redmond, WA, USA*. IEEE Computer Society, 2008, p. 147-156. doi : 10.1109/ISSRE.2008.25.

[Bot+09] Bernard Botella, Mickaël Delahaye, Stéphane Hong Tuan Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger et Nicky Williams. « Automating Structural Testing of C Programs : Experience with PathCrawler ». In : *Proceedings of the 4th International Workshop on Automation of Software Test, AST 2009, Vancouver, BC, Canada, May 18-19, 2009*. Sous la dir. de Dimitris Dranidis, Ste-

phen P. Masticola et Paul A. Strooper. IEEE Computer Society, 2009, p. 70-78. doi : 10.1109/IWAST.2009.5069043.

[Bar+21] Sébastien Bardin, Nikolai Kosmatov, Michaël Marcozzi et Mickaël Delahaye. « Specify and measure, cover and reveal : A unified framework for automated test generation ». In : *Sci. Comput. Program.* 207 (2021), p. 102641. doi : 10.1016/j.scico.2021.102641.

[Bob+17] François Bobot, Zakaria Chihani et Bruno Marre. « Real Behavior of Floating Point Numbers ». In : *Proceedings of the 15th International Workshop on Satisfiability Modulo Theories affiliated with the International Conference on Computer-Aided Verification (CAV 2017), Heidelberg, Germany, July 22 - 23, 2017*. Sous la dir. de Martin Brain et Liana Hadarean. T. 1889. CEUR Workshop Proceedings. CEUR-WS.org, 2017, p. 50-62.

[Bar+18] Sébastien Bardin, Nikolai Kosmatov, Bruno Marre, David Mentré et Nicky Williams. « Test Case Generation with PathCrawler/LTest : How to Automate an Industrial Testing Process ». In : *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*. Sous la dir. de Tiziana Margaria et Bernhard Steffen. T. 11247. Lecture Notes in Computer Science. Springer, 2018, p. 104-120. doi : 10.1007/978-3-030-03427-6_12.

[Kil73] Gary A. Kildall. « A Unified Approach to Global Program Optimization ». In : *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*. Sous la dir. de Patrick C. Fischer et Jeffrey D. Ullman. ACM Press, 1973, p. 194-206. doi : 10.1145/512927.512945.

[Pro59] Reese T. Prosser. « Applications of Boolean matrices to the analysis of flow diagrams ». In : *Papers presented at the 1959 eastern joint IRE-AIEE-ACM computer conference, IRE-AIEE-ACM 1959 (Eastern), Boston, Massachusetts, USA, December 1-3, 1959*. Sous la dir. de Frank E. Heart. ACM, 1959, p. 133-138. doi : 10.1145/1460299.1460314.

[Nem+06] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun et Marianne De Michiel. « PapaBench : a Free Real-Time Benchmark ». In : *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*. Sous la dir. de Frank Mueller. T. 4. OASIcs. Internationales Begegnungs- und

Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[Kui+01] J. Kuitunen, G. Drolshagen, JAM McDonnell, H. Svedhem, M. Leese, H. Mannermaa, M. Kaipiainen et V. Sipinen. « DEBIE-first standard in-situ debris monitoring instrument ». In : *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP* 473 (2001), p. 185-190.

[Bar+15] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon et Jean-Yves Marion. « Sound and Quasi-Complete Detection of Infeasible Test Requirements ». In : *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 2015, p. 1-10. doi : 10.1109/ICST.2015.7102607.

[Bar+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi et Francesco Regazzoni. « Countermeasures against fault attacks on software implemented AES : effectiveness and cost ». In : *Proceedings of the 5th Workshop on Embedded Systems Security, WESS 2010, Scottsdale, AZ, USA, October 24, 2010*. ACM, 2010, p. 7. doi : 10.1145/1873548.1873555.

[ANS21] ANSSI. « Règles de programmation pour le développement sécurisé de logiciels en langage C ». In : 2021.

[Lau+19] Johan Laurent, Christophe Deleuze, Vincent Beroulle et Florian Pebay-Peyroula. « Analyzing Software Security Against Complex Fault Models with Frama-C Value Analysis ». In : *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2019, Atlanta, GA, USA, August 24, 2019*. IEEE, 2019, p. 33-40. doi : 10.1109/FDTC.2019.00013.

[Ben+19] Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry et Philippe Trebuchet. « WooKey : designing a trusted and efficient USB device ». In : *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*. Sous la dir. de David Balenson. ACM, 2019, p. 673-686. doi : 10.1145/3359789.3359802.

[Rob22] Virgile Robles. « Specifying and verifying high-level requirements on large programs : application to security of C programs. (Spécifier et vérifier des exigences de haut niveau sur des programmes importants : application à la sécurité des programmes C) ». Thèse de doct. University of Paris-Saclay, France, 2022.

[Cha+17] Kwonsoo Chae, Hakjoo Oh, Kihong Heo et Hongseok Yang. « Automatically generating features for learning program analysis heuristics for C-like languages ». In : *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 101 :1-101 :25. DOI : 10.1145/3133925.

[Oli+16] Steven de Oliveira, Saddek Bensalem et Virgile Prevosto. « Polynomial Invariants by Linear Algebra ». In : *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*. Sous la dir. de Cyrille Artho, Axel Legay et Doron Peled. T. 9938. Lecture Notes in Computer Science. 2016, p. 479-494. DOI : 10.1007/978-3-319-46520-3_30.

[Oca21] Discuss Ocaml. *The road to OCaml 5.0*. https://discuss.ocaml.org/t/the-road-to-ocaml-5-0/8584. 2021.

Index