Réutilisations de caches et d'invariants pour l'analyse statique incrémentale

Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle, Julien Signoles

Réutilisations de caches et d'invariants pour l'analyse statique incrémentale

Mamy Razafintsialonina[1,2], David Bühler[1], Antoine Miné[2], Valentin Perrelle[1] et Julien Signoles[1]

[1]Université Paris-Saclay, CEA, List, Palaiseau, France
[2]Sorbonne Université, CNRS, LIP6, Paris, France

L'analyse statique de programmes permet aujourd'hui d'analyser des programmes de grande taille, avec une très bonne précision, tout en étant raisonnablement rapide. Néanmoins, les temps d'analyse continuent de se compter en minutes, voire dizaines de minutes, ce qui rend compliqué leur intégration dans les processus de développement : les modifications d'un programme y sont très fréquentes et requièrent donc d'obtenir rapidement les résultats de l'analyseur. Néanmoins, ces modifications sont souvent mineures, de l'ordre de quelques lignes de code tout au plus. L'analyse statique incrémentale exploite cette caractéristique pour permettre à un analyseur statique de se contenter d'actualiser les résultats d'une analyse antérieure plutôt que de tout recalculer, ce qui permet des gains de temps significatifs. Cet article présente deux nouvelles approches pour l'analyse statique incrémentale, l'une réutilisant des caches de fonction et l'autre des invariants de boucle. Nous les avons implémentées dans Eva, l'analyseur de valeurs par interprétation abstraite de Frama-C en utilisant une nouvelle fonctionalité de cette plateforme permettant de comparer deux programmes. Nos travaux ont été évalués sur un ensemble de commits de programmes réels.

1 Introduction

Les logiciels modernes sont caractérisés par leur complexité, leur taille, mais aussi par leur évolution constante, que ce soit pour ajouter de nouvelles fonctionnalités, corriger des erreurs ou répondre à de nouvelles exigences. Cette évolutivité constante rend l'analyse statique de programme à la fois cruciale et difficile. Cruciale, car les bugs dans les logiciels peuvent avoir des conséquences désastreuses, particulièrement sur les systèmes critiques : il est donc souvent nécessaire de lancer des analyses sur ceux-ci, afin d'assurer la qualité du logiciel et de détecter d'éventuels problèmes le plus tôt possible. Mais cela est aussi difficile, car ces analyses prennent beaucoup de temps en raison de la taille du code, alors même que les changements apportés à chaque modification sont souvent petits et très localisés. L'analyse statique incrémentale est une solution prometteuse pour pallier ces problèmes. En effet, elle tire parti de la similarité entre différentes versions d'un même programme, afin de permettre une analyse plus rapide en réutilisant les résultats des analyses précédentes. Dans l'état de l'art [MD15, MOJ18], l'analyse complète d'un programme est souvent privilégiée pour garantir une approche sûre, sans omission d'alarmes. Toutefois, en pratique, l'analyse incrémentale, bien que non complète, peut apporter plus de rapidité, en assouplissant possiblement la sûreté [CDD+15, MGR13]. Le dilemme réside donc dans la détermination précise des portions du programme à ré-analyser et celles dont les résultats peuvent être réutilisés

des analyses précédentes. En effet, un changement syntaxique minime peut engendrer des effets sémantiques importants sur des parties syntaxiquement non modifiées du programme, rendant la ré-analyse potentiellement plus coûteuse que la modification elle-même.

Dans cet article, nous proposons une méthode d'analyse incrémentale pour l'interprétation abstraite [CC77, Cou21], fondée sur une correspondance de programmes et des heuristiques pour identifier de manière sûre des résultats d'analyse de fonctions et de boucles qui peuvent être réutilisés pour accélérer l'analyse du nouveau programme. L'approche choisie garantie la sûreté, tout en préservant une grande précision. Elle est générique et peut être implémentée sur n'importe quel interpréteur abstrait. Nous avons choisi ici le greffon Eva [BBY17] de la plateforme Frama-C [BBB+21] dédiée à l'analyse de code C. Cette implémentation a été évaluée sur un ensemble de commits issus du développement de deux programmes réels, PolarSSL et Monocypher. Cette évaluation montre une réduction significative du temps d'analyse contre une légère hausse de la consommation mémoire, tout en limitant la perte de précision par rapport à une analyse traditionnelle. En offrant une analyse incrémentale plus rapide, tout en conservant la sûreté et une grande précision, nous espérons favoriser l'intégration des analyseurs statiques dans les processus de développement [JSMHB13, CB16], notamment ceux d'intégration continue. En résumé, nos contributions sont les suivantes :

— une technique de correspondance de programmes pour identifier les fonctions non modifiées ;
— une approche basée sur une heuristique pour la correspondance entre les conditions de boucle de deux fonctions modifiées ;
— une approche sûre pour réutiliser le cache de l'analyse d'un programme précédent pour accélérer l'analyse d'un programme modifié ou non.

Travaux connexes Les techniques de mise en cache et de réutilisation des résultats de calculs, en particulier celles fondées sur la mémoïsation des fonctions pures [ALL96, FT90], ont été largement étudiées et appliquées, y compris dans l'analyse de programmes, remontant au moins au développement d'analyses flot de données pour supporter la compilation continue et réactive [Ryd83]. Des travaux plus récents ont contribué à l'incrémentalisation de plusieurs classes d'analyse de programmes, notamment les analyses flot de données [AB14, DAL+17], et les analyses fondées sur des extensions de Datalog [SEV16]. Ces approches, bien qu'efficaces, sont généralements limitées à des classes d'analyses spécifiques, imposant des restrictions sur les domaines abstraits en excluant certains domaines comme ceux de hauteur infinie.

Les analyses modulaires [CC02], fondées sur les résumés [VJL07, MGR13, O'H18, CDOY11, VdPSVEDR20, CDD+15] s'étendent naturellement à l'analyse incrémentale. En particulier, [VdPSVEDR20] propose un cadre modulaire d'analyse flot de donnée fondée sur un algorithme de liste de travail. Lors d'une analyse incrémentale, les composants affectés par les modifications sont initialement ajoutés dans la liste, et ceux indirectement affectés le sont au fur et à mesure. Infer [CDD+15] et Coverity [MGR13] supportent l'analyse incrémentale et fonctionnent sur des programmes de grande taille. Cependant, leurs approches n'offrent pas de garantie de sûreté. En particulier, l'analyse incrémentale d'Infer retourne seulement les nouvelles alarmes des fonctions analysées. Un problème partagé des résumés est la perte de précision induite par le formalisme utilisé. Une analyse modulaire sûre implique un résumé capable de représenter tous les contextes d'appels possibles, ce qui induit un compromis entre précision et coût de l'analyse. Notre approche est inspirée des résumés et garantie la sûreté, tout en préservant une grande précision de l'analyse.

Les travaux dans [SEV20, EST+22] sont plus récents et proposent des approches pour améliorer la précision. L'analyse incrémentale est implémentée sur un solveur générique de point-fixe [SV21]. Ils montrent que les structures de données utilisées par le solveur peuvent être exploitées pour limiter les parties à ré-analyser après une modification. [SCS21] propose une nouvelle sémantique opérationnelle qui réifie l'interprétation abstraite sur un graphe acyclique orienté, maintient les dépendances sur le graphe et implémente l'incrémentalité sur le graphe en invalidant les nœuds affectés par les modifications. Cette approche nécessite des constructions dédiées pour les boucles, les appels de fonctions dynamiques ou les récursions.

Cependant, ces travaux n'ont pas été évalués sur des programmes réels et les cas d'étude sont limités à des programmes simples. [NEH19] propose une approche réutilisant les points-fixes précédemment calculés, afin d'accélérer l'analyse des programmes dynamiques comme JavaScript. Leur approche se fonde sur une méthode de comparaison de code source pour faire la correspondance entre deux programmes. La sureté est garantie par une analyse complète du programme. Cette technique est proche de la nôtre. Cependant, étant donné que la précision de l'analyse dépend directement de la méthode de comparaison et correspondance, nous nous concentrons seulement sur la réutilisation des points-fixes de boucle des fonctions modifiées. Mopsa [MOJ18, MOM20] et Astrée [MD15] utilisent des caches pour les fonctions et les itérations de point-fixe pour accélérer l'analyse interprocédurale et multi-processus. Cependant, ces caches ne sont pas conçus pour l'analyse incrémentale et ne sont pas réutilisés pour les versions suivantes du programme.

D'autres travaux se concentrent plutôt sur la comparaison de programmes. Ainsi, les travaux dans [FMB+14, DP16] proposent des algorithmes efficaces pour transformer un AST en un autre, avec des séquences de modifications appelées `edit scripts`. Ces approches permettent de trouver une séquence minimale de modifications pour transformer un AST en un autre, mais ne garantissent pas l'équivalence sémantique entre deux programmes, qui est nécessaire pour identifier les résultats réutilisables de manière sûre. L'application de ces approches à l'analyse incrémentale nécessite une étude plus approfondie. Cela pourrait être une piste intéressante pour améliorer notre approche.

Les travaux dans [DM19] se concentrent sur la comparaison de programmes au niveau de la sémantique, en utilisant une nouvelle sémantique dénotationnelle pour vérifier si deux programmes se comportent de la même manière lorsqu'ils sont exécutés avec les mêmes entrées. Cette approche est intéressante, notamment pour inférer que deux fonctions sont équivalentes. L'application de cette approche à l'analyse incrémentale n'est pas étudiée.

Plan de l'article L'article est organisé comme suit. La section 2 fournit un contexte théorique introductif sur l'interprétation abstraite. La section 3 présente la technique sur quelques exemples motivateurs. La section 4 présente les outils existants dans Frama-C sur lesquels notre approche repose. La section 5 présente notre approche pour l'analyse incrémentale. La section 6 discute de l'implémentation et de l'évaluation de notre approche. Finalement, la section 7 conclut l'article et présente quelques perspectives de recherche.

2 Préliminaires

L'interprétation abstraite [CC77, Cou21] est une théorie générale permettant de définir des domaines abstraits pour sur-approximer de manière sûre les comportements des programmes, tout en fournissant des garanties formelles sur les propriétés analysées.

Analyse d'un programme Soit P un programme, $\mathcal{L}$ l'ensemble de tous les emplacements mémoires possibles dans P et $\mathbb{V}$ l'ensemble de toutes les valeurs qu'on peut stocker dans un emplacement mémoire. Une mémoire $M : \mathcal{L} \to \mathbb{V}$ est une fonction qui associe à chaque emplacement mémoire $l \in \mathcal{L}$ une valeur $v \in \mathbb{V}$. Soit $\mathcal{D} \stackrel{def}{=} \mathcal{P}(M)$ le domaine concret représentant l'ensemble de tous les états mémoires possibles du programme P. Un domaine abstrait $\mathcal{D}^\sharp$ est un ensemble qui sur-approxime l'ensemble des états concrets dans $\mathcal{P}(M)$. Formellement, $\mathcal{D}^\sharp$ est un treillis muni d'une relation d'ordre $\sqsubseteq$, d'une opération d'union (resp. d'intersection) abstraite $\sqcup^\sharp$ (resp. $\sqcap^\sharp$) et d'un plus petit (resp. plus grand) élément $\bot$ (resp. $\top$). Nous introduisons $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(M)$ la fonction de concrétisation qui associe à chaque état abstrait dans $\mathcal{D}^\sharp$ un état concret dans $\mathcal{P}(M)$. Cette fonction permet de convertir un état abstrait vers un état concret.

Pour toute opération concrète $F : \mathcal{P}(M) \to \mathcal{P}(M)$ et sa version abstraite $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, la propriété de sûreté $\forall X^\sharp \in \mathcal{D}^\sharp,\ \gamma(F^\sharp(X^\sharp)) \supseteq F(\gamma(X^\sharp))$ garantit que l'opération abstraite $F^\sharp$ est une sur-approximation de l'opération concrète F.

Analyse d'une boucle L'analyse d'une boucle calcule un point-fixe en itérant sur son corps et en accumulant les états accessibles à chaque itération, jusqu'à l'obtention d'un état stable, appelé invariant. Soit $(\mathcal{D}^\sharp, \sqsubseteq)$, $X^\sharp \in \mathcal{D}^\sharp$ et un opérateur abstrait $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, le test de stabilité est vérifiée par $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$. Dans le concret, on a un (plus petit) point-fixe $X = F(X)$, qui correspond au meilleur invariant inductif. Dans l'abstrait, on a un post-point-fixe $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$, qui n'est pas forcément un point-fixe. De plus, l'existence d'un point-fixe n'est pas toujours garantie, mais celle du post-point-fixe l'est : il suffit de choisir $\top$. Le cadre de l'interprétation abstraite garantit l'existence du post-point-fixe abstrait, qui est une abstraction sûre du point-fixe concret. Le nombre d'itérations pour l'atteindre peut néanmoins être important. Pour accélérer la convergence au prix d'une précision moindre, une analyse par interprétation abstraite utilise un opérateur spécial, appelé élargissement et noté ∇. Plus précisément, $\nabla : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ est un opérateur binaire d'un domaine abstrait $(\mathcal{D}^\sharp, \sqsubseteq)$ s'il calcule une borne supérieure pour toute paire $X^\sharp, Y^\sharp \in \mathcal{D}^\sharp$, c'est-à-dire si $X^\sharp \sqsubseteq X^\sharp \nabla Y^\sharp$ et $Y^\sharp \sqsubseteq Y^\sharp \nabla X^\sharp$, et si, pour toute séquence $Y_i^\sharp \in \mathcal{D}^\sharp$, la séquence $X_i^\sharp$ définie par $X_0^\sharp = Y_0^\sharp$ et $X_{n+1}^\sharp = X_n^\sharp \nabla Y_{n+1}^\sharp$ converge en temps fini. Cet opérateur est intégré dans l'algorithme de calcul du point-fixe. Considérons une boucle dont l'état abstrait initial est $X_0^\sharp$. À chaque itération n, le nouvel état abstrait $X_{n+1}^\sharp$ est calculé en appliquant l'opérateur abstrait $F^\sharp$ sur l'état courant $X_n^\sharp$, c'est-à-dire $X_{n+1}^\sharp = F^\sharp(X_n^\sharp)$. Plutôt que d'utiliser directement $X_{n+1}^\sharp$, on utilise l'opérateur d'élargissement ∇ pour calculer $X_{n+1}^\sharp = X_n^\sharp \nabla F^\sharp(X_n^\sharp)$. En pratique, ∇ est généralement appliqué après un certain nombre d'itérations sans élargissement pour rester précis, avant d'accélérer la convergence.

3 Cas pratiques et illustrations des concepts

Dans cette section, nous présentons des exemples de programmes qui illustrent l'intérêt de l'analyse incrémentale. Nous présentons deux approches différentes, mais complémentaires, pour accélérer l'analyse incrémentale. Considérons le programme suivant :

```
1      // Compute the area of a square (Changed)
2      int calculateArea(int side) {
3  -     return side + side;  // Incorrect formula
4  +     return side * side;  // Correct formula
5      }
6      // Compute the perimeter of a square (Unchanged)
7      int calculatePerimeter(int side) {
8        return 4 * side;
9      }
10     // Main function (Unchanged)
11     void main() {
12       int side = 5;
13       // side = [5; 5]
14       int area = calculateArea(side);
15       // side = [5; 5], area = [10; 10]
16       int perimeter = calculatePerimeter(side);
17       // side = [5; 5], area = [10; 10], perimeter = [20; 20]
18       printf("Area: %d, Perimeter: %d\n", area, perimeter);
19     }
```

Ici, la fonction `calculateArea` calcule l'aire d'un carré. La formule initiale à la ligne 3 est incorrecte. Une version ultérieure du code la corrige par la ligne 4. Les deux autres fonctions, `calculatePerimeter` et `main`, demeurent inchangées. En effectuant une analyse d'intervalle sur le programme incorrect, nous obtenons les résultats d'analyse indiqués en commentaires dans la fonction `main`. Ils associent, à la fin de chaque instruction de cette fonction, un intervalle (ici, singleton) à chaque variable du programme. Par exemple, à la ligne 15, l'intervalle pour la variable `area` après le calcul incorrect est le singleton $[10; 10]$.

Dans le cas d'une analyse statique non incrémentale, toute modification dans une fonction comme `calculateArea` nécessiterait une nouvelle analyse complète de la fonction `main` et de toutes les fonctions qu'elle appelle, y compris `calculatePerimeter`. Ce processus peut être très couteux en ressource, surtout si les fonctions en question sont complexes. Cependant ici, la modification est strictement localisée à la fonction `calculateArea`. Dès lors, il est possible de réutiliser les résultats d'analyse précédents pour la fonction `calculatePerimeter`. Pour être plus précis, lors de la première analyse, nous sauvegardons la paire $(I^\sharp, O^\sharp)$ de toutes les fonctions analysées. Ces états abstraits $I^\sharp$ et $O^\sharp$ représentent l'ensemble des états mémoires accessibles en entrée et en sortie de la fonction. Une abstraction, comme les intervalles par exemple, ne représente que la borne inférieure et la borne supérieure pour chaque variable. Nous verrons dans la section 4 quelles sont les limitations de cette approche. Pour la fonction `calculatePerimeter`, nous sauvegardons la paire d'entrée/sortie ($[5;5]$, $[20;20]$). Lors d'analyses ultérieures, en présence d'un appel à `calculatePerimeter`, deux conditions doivent être vérifiées : d'une part, la fonction elle-même ne doit pas avoir subi de modification syntaxique ni appeler de fonction en ayant subi une et, d'autre part, l'intervalle $I^\sharp_{new}$ associé au paramètre d'entrée `side` doit être identique à celui $I^\sharp$ de l'analyse précédente (par exemple, $I^\sharp_{new} = I^\sharp = [5;5]$). Si ces deux conditions sont réunies, nous pouvons directement réutiliser l'intervalle $O^\sharp$ préalablement sauvegardé pour la valeur de sortie, éliminant ainsi le besoin de ré-analyser le corps de la fonction. En revanche, si l'une de ces conditions n'est pas satisfaite, une analyse complète du corps de la fonction est effectuée comme d'habitude. Nous verrons plus en détail dans la section 4 les propriétés nécessaires pour l'efficacité et la correction de cette approche. Considérons à présent un autre programme :

```
1  // Main function (Changed)
2    int main() {
3      int factorial = 1;
4      int i = 1;
5      int n = 5;
6      while (i <= n) {
7  -       factorial = factorial * i++;
8  +       factorial *= i;
9  +       i++;
10     }
11   }
```

Dans ce code, la fonction `main` calcule la factorielle de $n = 5$. La modification introduite à l'intérieur du corps de la boucle est une simple réécriture stylistique, qui n'a aucun impact sur la sémantique du programme. Cependant, une analyse statique non incrémentale nécessiterait une nouvelle analyse complète de la fonction `main`, car nous ne pouvons pas inférer syntaxiquement que ces programmes sont équivalents. Ce processus peut être très coûteux en ressource, surtout si le corps de la boucle est complexe ou fait appel à des fonctions elles-mêmes compliquées. Par ailleurs, la technique proposée à l'exemple précédent ne permet pas d'accélérer l'analyse puisque, dans ce cas, le code de la fonction analysée a été directement modifié. Le tableau suivant montre le résultat d'une analyse sur la fonction factorielle originale de notre exemple : l'analyse converge en trois itérations, au prix d'une perte de précision sur le résultat de la fonction pour lequel l'analyse conclut seulement qu'il est strictement positif. Une analyse de la version modifiée du programme, sans analyse incrémentale, aboutit exactement aux mêmes résultats avec le même nombre d'itérations.

itérations	i	fact	n
1	$[1, 2]$	$[1, 1]$	$[5, 5]$
2	$[1, +\infty[$	$[1, +\infty[$	$[5, 5]$
3	$[1, +\infty[$	$[1, +\infty[$	$[5, 5]$

Pour accélérer l'analyse de la boucle modifiée, on peut néanmoins réutiliser l'invariant calculé lors de l'analyse précédente. Il est à noter que le calcul du point-fixe de la boucle

commence généralement par un état initial $\bot$. Cependant, nous pouvons également initialiser l'analyse à partir d'un autre état : tant que le test de stabilité $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$ permettant de savoir si le point-fixe est atteint ou non, est réalisé correctement, nous obtiendrons un invariant correct à la fin, quelle que soit la suite des itérés parcourus. Ainsi, nous pouvons initialiser le calcul de point-fixe du programme modifié à partir de l'invariant $(i = [1; +\infty[; fact = [1; +\infty[; n = [5; 5])$ calculé lors de l'analyse précédente. Au mieux, si le nouvel invariant de boucle du programme modifié n'a pas changé, une itération avec le nouveau corps de la boucle à partir de l'ancien invariant suffit à montrer qu'il est toujours un invariant. Au pire, ce n'est pas un invariant, et on continue alors l'itération jusqu'à obtenir une nouvelle valeur stable. Nous obtenons alors les résultats suivants :

itérations	i	fact	n
1	$[1; 2] \sqcup^\sharp [1; +\infty[$	$[1; 2] \sqcup^\sharp [1, +\infty[$	$[5; 5] \sqcup^\sharp [5; 5]$
2	$[1; +\infty[$	$[1; +\infty[$	$[5; 5]$

Ils montrent une convergence en deux itérations, au lieu de trois pour l'analyse non incrémentale, pour un résultat identique. Le test de stabilité est vérifié par $F^\sharp(X_1^\sharp) \sqsubseteq X_1^\sharp$ avec $X_1^\sharp = \{i = [1; 2] \sqcup^\sharp [1; +\infty[; fact = [1; 2] \sqcup^\sharp [1; +\infty[; n = [5; 5] \sqcup^\sharp [5; 5]\}$ et $F^\sharp(X_1^\sharp) = \{i = [1; +\infty[; fact = [1; +\infty[; n = [5; 5]\}$. Même si une telle réduction du nombre d'itérations nécessaires au calcul du point-fixe peut sembler minime sur cet exemple illustratif. Nous verrons que cette approche permet de réduire considérablement le temps nécessaire à une analyse complète sur des exemples plus significatifs, mais peut entraîner une perte de précision. En particulier quand on utilise des stratégies d'itérations plus fines (élargissement amélioré) qui nécessitent un nombre plus grand d'itérations pour maintenir la précision. Dans ce cas, le gain en nombre d'itérations peut être bien plus significatif.

4 Éléments de Frama-C utiles à l'analyse incrémentale

Frama-C est une plateforme open-source, composée d'un ensemble d'analyseurs dédiés à l'analyse de programme C. Elle permet de vérifier des propriétés de sûreté et de sécurité sur des programmes C à l'aide de méthodes formelles. Parmi ses analyseurs, on trouve le greffon Eva, fondé sur l'interprétation abstraite. Il s'agit d'une analyse de valeurs permettant notamment d'inférer des invariants portant sur les variables (et, plus généralement, les zones mémoires) du programme. Eva permet également à l'utilisateur de sélectionner, configurer et combiner différents domaines abstraits pour adapter l'analyse à des besoins spécifiques. Cette section présente les outils existants dans Frama-C et Eva qui ont été sujets à des extensions pour permettre l'analyse incrémentale présentée dans la section 5.

4.1 Memexec : Cache de Résultats d'Analyse de Fonctions

Eva analyse un programme de manière monolithique, en partant du point d'entrée du programme jusqu'aux points de sortie. L'analyse des fonctions est entièrement dépendante des contextes d'appel : à chaque fois qu'une fonction est appelée dans le programme, Eva prend en compte l'état du programme au point d'appel pour analyser la fonction. En conséquence, chaque fonction est analysée une fois par pile d'appel. Bien que cette approche dépendante du contexte garantisse une grande précision, elle peut s'avérer très coûteuse en termes de ressources. Pour pallier ce problème, Eva utilise un mécanisme appelé Memexec [Yak15]. Ce mécanisme agit comme un cache pour sauvegarder les résultats d'analyse d'une fonction. Lorsqu'une même fonction est appelée deux fois dans un contexte identique, Memexec permet à l'analyseur de réutiliser les résultats d'analyse précédemment calculés plutôt que d'analyser à nouveau le corps de la fonction. Ce processus de mise en cache et de réutilisation accélère considérablement l'analyse du programme, tout en conservant un degré élevé de précision.

Pour mieux comprendre ce mécanisme, nous allons modéliser son fonctionnement. Soit P un programme, $M : \mathcal{L} \to \mathbb{V}$ une mémoire, où $\mathcal{L}$ est l'ensemble des emplacements mémoires utilisés par P, $\mathcal{D}^\sharp$ un domaine abstrait et $F : \mathcal{P}(M) \to \mathcal{P}(M)$ une fonction concrète de P, et

$F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ une fonction abstraite qui sur-approxime F. Considérons également deux états abstraits $I^\sharp \in \mathcal{D}^\sharp$ et $O^\sharp \in \mathcal{D}^\sharp$ représentant respectivement l'entrée et la sortie de l'analyse de F, ce qui peut être formalisé par $F^\sharp(I^\sharp) = O^\sharp$. Dans ce cadre, $I^\sharp$ est une sur-approximation de l'ensemble des états concrets possibles avant l'exécution de F, tandis que $O^\sharp$ est une sur-approximation de l'ensemble des états concrets possibles après l'exécution de F.

Hypothèse de base En première approximation, l'analyse d'une fonction dépend uniquement de son état abstrait d'entrée $I^\sharp$ pour produire un état abstrait de sortie $O^\sharp$.

Analyse de F L'analyse de F est enrichie pour qu'elle calcule également l'ensemble des emplacements mémoires lus $R \subseteq \mathcal{L}$ et écrits $W \subseteq \mathcal{L}$ pendant l'analyse du corps de F pour produire son état abstrait de sortie. Le résultat de l'analyse de F avec $I^\sharp$ comme argument est noté $F^\sharp_{rw}(I^\sharp) = (O^\sharp, R, W)$. Les valeurs R et W sont les emplacements mémoires respectivement lus et écrits calculés lors de l'analyse de F. La projection d'un état abstrait $S^\sharp$ sur les emplacements mémoires L est notée $S^\sharp \downarrow L$, conservant l'information calculée pour L et oublie les informations sur les autres en y associant la valeur $\top$. À la fin de l'analyse de F, le quadruplet $(R, W, I^\sharp \downarrow R, O^\sharp \downarrow W)$ est sauvegardé dans un cache appelé Memexec, et est appelé un résumé de F.

Lors d'une analyse ultérieure de F avec un nouvel état abstrait d'entrée $I^\sharp_{new}$, si cet état abstrait est égal à l'état abstrait d'entrée du cache pour les emplacements mémoires lus R, c'est-à-dire si $I^\sharp_{new} \downarrow R = I^\sharp \downarrow R$, alors on peut réutiliser le résultat d'analyse précédent mis en cache, sans ré-analyser le corps de F. Sinon, on analyse F avec $I^\sharp_{new}$ et on met à jour le cache de Memexec. Lorsqu'on utilise le cache, le résultat est obtenu en combinant l'état abstrait de sortie du cache et le nouvel état d'entrée, projetés respectivement sur les emplacements mémoires écrits et ceux non modifiés, c'est-à-dire, en notant $\overline{W}$ le complémentaire de W :

$$O^\sharp \downarrow W \ \sqcap^\sharp \ I^\sharp_{new} \downarrow \overline{W} \tag{1}$$

Exemple Considérons le programme suivant :

```
1  void f(int *p) { *p = 1; }
2  void main() {
3    int a = 0, b = 0;
4    f(&a);
5    f(&a);
6    f(&b);
7  }
```

Le résumé inféré après analyse du premier appel de la fonction `f` à la ligne 4 est le suivant :

$$(R = \{p\}, W = \{a\}, I^\sharp \downarrow R = \{p \to \{\&a\}, a \to \top, b \to \top\},$$
$$O^\sharp \downarrow W = \{p \to \top, a \to [1;1], b \to \top\})$$

On suppose que ce résumé est sauvegardé dans le cache de Memexec. Lors de l'analyse du second appel de la fonction `f` à la ligne 5, l'état abstrait d'entrée est $I^\sharp_{new} = \{p \to \{\&a\}, a \to [0;0], b \to [0;0]\}$ et la projection de $I^\sharp_{new}$ sur les emplacements mémoires lus R est $I^\sharp_{new} \downarrow R = \{p \to \{\&a\}, a \to \top, b \to \top\}$. On vérifie alors si le nouvel état abstrait d'entrée $I^\sharp_{new}$ et l'état abstrait dans le cache $I^\sharp$ coïncident sur les emplacements mémoires lus $I^\sharp_{new} \downarrow R = I^\sharp \downarrow R$, ce qui est le cas. On peut donc réutiliser le résumé présent dans le cache de Memexec pour calculer le résultat d'analyse de la fonction `f` au point 5 sans ré-analyser son corps. Le résultat d'analyse est le suivant :

$$\{p \to \top, a \to [1;1], b \to \top\} \sqcap^\sharp \{p \to \{\&a\}, a \to \top, b \to [0;0]\} =$$
$$\{p \to \{\&a\}, a \to [1;1], b \to [0;0]\}$$

Enfin, lors de l'analyse du dernier appel à f, à la ligne 6, la condition de réutilisation n'est pas vérifiée $\{p \to \{\&b\}, a \to \top, b \to \top\} \neq \{p \to \{\&a\}, a \to \top, b \to \top\}$. Le corps de la fonction `f` est donc ré-analysé, puis le nouveau résumé est ajouté dans le cache.

Correction de l'analyse avec utilisation du cache Soit $I^\sharp$ un état abstrait d'entrée d'analyse de F et $O^\sharp$ un état abstrait de sortie de F. Lors de l'analyse de F, en plus de $O^\sharp$, l'analyseur fournit un ensemble d'emplacements mémoires lus et écrits R et W respectivement, qui vérifient que pour tout état abstrait $I^\sharp_{new}$, si les emplacements mémoires nécessaires pour analyser F avec $I^\sharp_{new}$ et $I^\sharp$ sont les mêmes (2), alors les états abstraits modifiés par F du résumé $O^\sharp$ doit sur-approximer les états abstraits modifiés par F dans $O^\sharp_{new}$ (3) et que les états abstraits non modifiés par F dans $I^\sharp_{new}$ sur-approximent les états abstraits non modifiés par F dans $O^\sharp_{new}$ (4).

$$I^\sharp_{new} \downarrow R = I^\sharp \downarrow R \Rightarrow \tag{2}$$

$$O^\sharp_{new} \downarrow W \sqsubseteq O^\sharp \downarrow W \tag{3}$$

$$\wedge\ O^\sharp_{new} \downarrow \overline{W} \sqsubseteq I^\sharp_{new} \downarrow \overline{W} \tag{4}$$

La vérification de ces conditions lors de la première analyse de F avec $I^\sharp$ permet d'ajouter le résumé dans le cache de Memexec. Lors d'une nouvelle analyse avec un nouvel état abstrait $S^\sharp$, il suffit de vérifier $S^\sharp \downarrow R = I^\sharp \downarrow R$ pour réutiliser le résumé.

Précision de l'analyse Pour un état abstrait $S^\sharp$, une mémoire M, un ensemble d'emplacements mémoires L et une fonction de concrétisation γ, le résultat de la projection de $S^\sharp$ sur L est une sur-approximation de l'ensemble de tous les états concrets possibles dans $\gamma(S^\sharp)$ qui ont la même valeur que M sur L, c'est-à-dire $\gamma(S^\sharp \downarrow L) \supseteq \{M \mid \exists M' \in \gamma(S^\sharp), M' \downarrow L = M\}$. On en déduit que la réutilisation du cache avec (1) est une sur-approximation du résultat d'analyse sans cache $F^\sharp(I^\sharp_{new})$: $\gamma(O^\sharp \downarrow W \ \sqcap^\sharp\ I^\sharp_{new} \downarrow \overline{W}) \supseteq \gamma(F^\sharp(I^\sharp_{new}))$. Cela est dû à la perte d'information lors de la projection des états abstraits sur les emplacements mémoire modifiés. En effet, cette opération perd d'éventuelles propriétés relationnelles entre les emplacements modifiés W et les autres. Considérons la fonction `void f() { x = y - 1; }`. Son analyse avec $I^\sharp = \{x < y\}$ produit $O^\sharp = \{x = y - 1\}$ ainsi que le résumé ($R = \{y\}, W = \{x\}, I^\sharp \downarrow R = \{\top\}, O^\sharp \downarrow W = \{\top\}$). Nous constatons que ce résumé ne contient aucune information sur les propriétés relationnelles de x et y, ce qui induit une perte de précision lors de la réutilisation du cache $\gamma(\top \sqcap^\sharp \top) \supseteq \gamma(O^\sharp)$. En pratique, Eva augmente R et W pour prévenir cette imprécision en prenant $R = W = \{x, y\}$. Nous n'avons cependant pas prouvé que le choix d'Eva garantissait ou ne garantissait pas l'absence de perte de précision.

4.2 AST Diff : Correspondance de programmes

Notre approche sur l'analyse statique incrémentale se fonde sur la réutilisation d'une manière sûre des résultats d'analyse précédents. Pour cela, il est nécessaire d'établir une correspondance entre les deux versions du programme. Cette opération est effectuée à l'aide d'AST Diff, un outil d'analyse syntaxique offert par Frama-C et comparant deux arbres de syntaxe abstraite (AST) raisonnablement proches, correspondant typiquement à deux versions successives d'un même programme. Cette comparaison permet de déterminer si une fonction F donnée a été modifiée ou non. Dans le cas où F n'a pas été modifiée, AST Diff fournit une correspondance des variables et des instructions de F. En revanche, si la fonction a été modifiée, nous perdons toute correspondance.

Pour mieux comprendre AST Diff, nous allons modéliser son fonctionnement. Soit AST_{old} et AST_{new} les arbres syntaxiques abstraits de deux versions successives d'un même programme. Pour chaque fonction F_{new} dans AST_{new}, on cherche une fonction F_{old} de même nom dans AST_{old}. L'équivalence entre F_{new} et F_{old} est déterminée en comparant l'équivalence structurelle des deux fonctions modulo renommage des variables locales(6) et en comparant l'égalité (de nom et de valeur) des variables globales qu'elles utilisent(7) et en tenant compte

des fonctions appelées qui doivent être elles-mêmes deux à deux égales(8).

$$F_{new} \equiv F_{old} \Leftrightarrow \tag{5}$$
$$\mathrm{AST}(F_{new}) \equiv \mathrm{AST}(F_{old}) \tag{6}$$
$$\wedge\, \mathrm{Globals}(F_{new}) = \mathrm{Globals}(F_{old}) \tag{7}$$
$$\wedge\, \forall f_{new} \in \mathrm{Calls}(F_{new}), \exists f_{old} \in \mathrm{Calls}(F_{old}), f_{new} \equiv f_{old} \tag{8}$$

Si l'équivalence (5) est vérifiée, on considère que F_{new} est une fonction qui n'a pas été modifiée. Dans ce cas, on remplit une table de correspondance entre les noeuds de F_{new} correspondant aux instructions et aux utilisations de variables et ceux similaires dans F_{old}. Sinon, on considère F_{new} comme une fonction qui a été modifiée. On note que deux variables globales ne sont équivalentes que si elles ont strictement le même nom.

À l'état de l'art [DP16, FMB+14], les algorithmes de comparaison d'AST calculent un ensemble minimal de modifications nécessaires pour transformer un AST en un autre. Certes, un ensemble vide permet d'inférer que les deux ASTs sont identiques. En revanche, l'application des modifications pour transformer l'AST AST_{old} en AST_{new} ne permet pas de garantir que les résultats d'analyse précédemment liés à AST_{old} soient toujours valides pour AST_{new}. En effet, nous ne cherchons pas à transformer AST_{old} en AST_{new}, nous cherchons à établir une correspondance entre les fonctions dans les deux ASTs. Cela constitue déjà une base solide de garantie de sûreté pour la réutilisation des résultats d'analyse précédents pour les fonctions non modifiées, tout en nous permettant d'identifier déjà les potentielles extensions de cette technique qui pourraient nous être utiles.

5 Approche incrémentale

Cette section présente les approches que nous avons étudiées. Nous nous concentrons sur une réutilisation sûre des résultats d'analyse précédemment sauvegardés. La section 5.1 décrit l'approche consistant à recharger et réutiliser les résumés sauvegardés d'une fonction qui *n'a pas été modifiée*. La section 5.2 propose une extension d'AST Diff pour établir une correspondance entre boucles. La section 5.3 décrit la réutilisation des invariants de boucle.

5.1 Réutilisation des résumés de fonctions

Memexec non incrémental efface ses résultats en fin d'analyse. Nous étendons Memexec pour les sauvegarder. Pour cela, nous procédons en deux étapes : nous rechargeons les résumés de fonctions sauvegardés d'une analyse précédente seulement pour les fonctions qui n'ont pas été modifiées, puis nous réutilisons ces résumés de fonctions pour accélérer l'analyse du programme modifié. Distinguer ces deux étapes est important, car *recharger* le cache d'une fonction est une opération coûteuse qui n'implique pas forcément la *réutilisation* des résumés de fonction dans le cache : il faut donc éviter de la réaliser trop souvent.

Nous détaillons ces étapes dans les paragraphes suivants. Soit un programme P et son cache C, on suppose que le cache est sauvegardé après l'analyse du programme P. Notons $\mathcal{F}_P$ l'ensemble des fonctions du programme P. On appelle un résumé d'une fonction le quadruplet $(R, W, I^\sharp \downarrow R, O^\sharp \downarrow W)$. On suppose également que C contient les résumés de toutes les fonctions analysées du programme P. Ainsi, C est formellement défini comme l'ensemble des paires composées chacune d'une fonction $F \in \mathcal{F}_P$ et de l'ensemble des résumés de F que l'on note S_F, c'est-à-dire $C = \{(F, S_F) \mid \forall F \in \mathcal{F}_P\}$.

Conditions de rechargement du cache Soit un programme P_m, correspondant à une version modifiée de P. Pour analyser P_m, nous rechargeons le cache de résumés de fonction C de P. Cependant, si $P_m \neq P$, alors tous les résumés de fonctions dans C ne sont pas forcément valides pour P_m. En effet, supposons que F soit une fonction de P et F_m la version modifiée de F dans P_m. Nous pouvons alors distinguer deux cas. Si $F \equiv F_m$, alors les résumés de F dans C sont valides pour F_m. On peut donc recharger ces derniers. Sinon, les

résumés de F dans C sont potentiellement invalides pour F_m. On évite de les recharger pour garantir la sûreté. On initialise ensuite le cache C_{new} du programme P_m avec les résumés de fonctions rechargeables de C. Enfin, on analyse le programme P_m avec le cache C_{new}.

Conditions de réutilisation du cache et accélération de l'analyse Soit le programme P_m avec le cache C_{new} initialisé à partir des caches rechargeables de C. La condition de réutilisation des résumés de fonction dans C_{new} est la même que la condition d'utilisation du cache dans Memexec non incrémental : il faut que le nouvel état abstrait d'entrée $I^\sharp_{new}$ coïncide avec un état abstrait $I^\sharp$ présent dans le cache, mais uniquement sur les emplacements mémoires lus R, c'est-à-dire $I^\sharp_{new} \downarrow R = I^\sharp \downarrow R$. Ainsi l'accélération de l'analyse de P_m dépend du nombre de résumés de fonctions réutilisables dans C_{new}.

Exemple Considérons le programme suivant :

```
1      int foo(int *p) { if(*p > 0) *p = 1; else *p = -1; }
2      void main() {
3  -     int a = 0;
4  +     int a = 1;
5        int b = 0;
6        foo(&a);
7        foo(&b);
8      }
```

En analysant le programme non modifié, nous obtenons les résumés suivants pour les deux appels à la fonction `foo` :

$$(\text{foo},\ (R = \{p, a\}, W = \{a\}, I^\sharp \downarrow R = \{p \to \{\&a\}, a \to [0;0], b \to \top\},\ O^\sharp \downarrow W = \{p \to \top, a \to [-1;-1], b \to \top\})) \quad (9)$$

$$(\text{foo},\ (R = \{p, b\}, W = \{b\}, I^\sharp \downarrow R = \{p \to \{\&b\}, a \to \top, b \to [0;0]\},\ O^\sharp \downarrow W = \{p \to \top, a \to \top, b \to [-1;-1]\})) \quad (10)$$

Nous omettons le résumé de la fonction `main`, car la modification de la variable `a` invalide le cache du résumé de cette fonction. En analysant le programme modifié, nous rechargeons seulement le cache de la fonction `foo`. Nous distinguons deux cas. Le résumé (9) de la fonction `foo` n'est pas réutilisé à la ligne 6, car l'état abstrait d'entrée $I^\sharp_{new}$ ne coïncide pas avec l'état abstrait $I^\sharp$ sur les emplacements mémoires R présent dans le cache, c'est-à-dire $\{p \to \{\&a\}, a \to [1;1], b \to \top\} \neq \{p \to \{\&a\}, a \to [0;0], b \to \top\}$. Le résumé (10) de la fonction `foo` est réutilisé à la ligne 7, car l'état abstrait d'entrée $I^\sharp_{new}$ coïncide avec l'état abstrait $I^\sharp$ sur les emplacements mémoires R présent dans le cache, c'est-à-dire $\{p \to \{\&b\}, a \to \top, b \to [0;0]\} = \{p \to \{\&b\}, a \to \top, b \to [0;0]\}$.

Impact des modifications sur l'efficacité en temps de l'analyse Le bénéfice attendu de cette approche varie selon la nature des modifications apportées au programme. Nous distinguons deux cas de modifications :

1. **Modifications mineures :** ces modifications ne sont pas visibles dans l'AST du programme ou n'invalident pas plusieurs caches de résumés de fonctions.
   — Modifications qui n'impactent pas la sémantique du programme (formatage du code, ajout de commentaires, réorganisation des fonctions, etc.).
   — Petites modifications localisées dans une fonction qui n'est pas profondément imbriquée dans la pile d'appel.
2. **Modifications majeures :** Ces modifications peuvent invalider plusieurs caches de résumés de fonctions.
   — Modifications qui impactent la sémantique du programme (ajout, suppression, modification de variables globales utilisées par plusieurs fonctions).

— Petites modifications localisées dans une fonction qui est profondément imbriquée dans la pile d'appel.

Pour les modifications mineures, nous pouvons recharger un grand nombre de résultats d'analyse dans le cache, ce qui augmente la probabilité de réutiliser les résumés de fonctions et donc d'accélérer l'analyse du programme modifié. Pour les modifications majeures, il est très probable que la plupart des caches seront invalidés directement ou indirectement à cause des modifications dans les fonctions profondément imbriquées dans la pile d'appel qui invalident transitivement le cache de toutes les fonctions appelantes, ce qui augmente considérablement le temps d'analyse du programme modifié.

Cette approche hérite des mêmes limitations que Memexec non incrémental. Memexec ne peut ni sauvegarder ni réutiliser les résultats d'analyse d'une fonction qui alloue de la mémoire dynamiquement. En effet, Eva adopte différentes stratégies d'analyse des allocations dynamique qui peuvent dépendre de plusieurs paramètres, tels que le contexte d'appel et le nombre de bases mémoire déjà allouées pour analyser de telles fonctions. Cela signifie que l'hypothèse de base – l'analyse de la fonction ne dépend que de l'état abstrait d'entrée – n'est plus valide. Cela est également valable pour tout analyseur statique qui analyse une fonction avec d'autres paramètres que l'état d'entrée. Nous ne pouvons donc pas recharger les résultats d'analyse de ces fonctions, car ils n'ont pas été sauvegardés dans l'analyse précédente. Par conséquent, le temps d'analyse incrémentale du programme modifié augmente en fonction du nombre de résultats d'analyse de fonctions qui ne peuvent pas être rechargés. Nous verrons dans la section 6 l'impact de cette limitation sur l'efficacité de l'analyse en pratique.

5.2 Extension d'AST Diff

Dans cette section, nous proposons une extension d'AST Diff pour établir une correspondance entre les conditions de boucle de deux fonctions modifiées. Cette condition contrôlant l'accès au corps de la boucle est généralement associé à l'invariant de boucle. Ainsi, établir une correspondance nous permet de recharger l'invariant inféré dans la version précédente du programme d'une fonction modifiée. Cependant, la version actuelle d'AST Diff ne permet pas d'établir une correspondance entre les variables locales et les instructions de deux fonctions modifiées. Nous proposons donc une extension d'AST Diff pour établir une correspondance partielle entre les variables locales et les boucles des deux fonctions modifiées. Pour les variables locales, celles de mêmes noms sont ajoutées à une table. Pour les boucles, il est nécessaire de définir un critère d'équivalence entre les instructions de boucle, car ces dernières peuvent avoir subi une modification. Nous utilisons à cet effet une heuristique fondée sur l'ordre d'apparition des boucles. La correspondance entre les boucles nous permet d'identifier rapidement un invariant de boucle potentiellement réutilisable dans la version actuelle du programme et la correspondance entre les variables locales nous permet d'assurer qu'un invariant réutilisé porte sur les mêmes variables que celui à calculer.

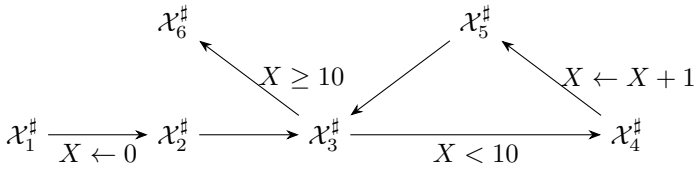
Toutefois, il est important de souligner que cette méthode peut générer de fausses correspondances, notamment si une boucle a été ajoutée ou supprimée. Nous verrons par la suite que ces erreurs n'affectent pas la sûreté de l'analyse. Cependant, elles peuvent tout de même conduire à des faux positifs et donc à une perte de précision. Illustrons cette extension sur l'exemple de la figure 1 qui montre deux versions d'une fonction `f` contenant deux boucles $L1$ et $L2$ et leurs invariants. Soit $L1_{new}$ et $L2_{new}$ les boucles de la fonction modifiée. Initialement, la correspondance entre les deux versions de `f` est perdue, car `f` a été modifiée. Nous cherchons donc à établir une correspondance partielle entre les variables locales et les boucles des deux versions. Ici, nous obtenons $L1 \equiv L1_{new}$ et $L2 \equiv L2_{new}$ grâce à cette heuristique même si, syntaxiquement, $L2 \neq L2_{new}$. Ensuite, la correspondance entre les variables locales garantie que les invariants liés à $L1$ et $L2$ portent sur les mêmes variables que ceux de $L1_{new}$ et $L2_{new}$, même si les valeurs de x diffèrent.

```
1       int f() {
2   -       int x = 0;
3   +       int x = 1;
4           while (x < 10) { // L1, x = [0; 10]
5             x++;
6           }
7           ... // Code
8           int y = 0;
9           while (y < 10) { // L2, y = [0; 10]
10  -           y = y + 1;
11  +           y++;
12          }
13      }
```

Figure 1. Exemple de correspondance de boucle.

$X \leftarrow 0$
while $X < 10$ **do**
  $X \leftarrow X + 1$
end while

(a) Programme



(b) CFG

Figure 2. Exemple de programme incrémentant X et son CFG.

5.3 Réutilisation des invariants de boucle

Les modifications apportées à une fonction profondément imbriquée dans la pile d'appel ou sur plusieurs fonctions peuvent entraîner l'invalidation de nombreux caches de résumés de fonctions, rendant ainsi l'approche précédente inefficace. Nous souhaitons réutiliser d'autres résultats d'analyse pour accélérer l'analyse incrémentale. Pour cela, nous réutilisons les invariants de boucle précédemment sauvegardés, ce qui est possible grâce à notre extension d'AST Diff. Cette approche permet d'accélérer l'analyse des boucles dans une fonction qui a été modifiée. Plus précisément, au lieu de commencer l'itération de la boucle à partir d'un état initial $\mathcal{X}^\sharp$, nous la commençons à partir de l'invariant de boucle sauvegardé combiné avec l'état abstrait d'entrée de la boucle. Cela permet d'éviter des itérations abstraites si l'invariant de boucle sauvegardé est proche de l'invariant de boucle du programme actuel.

La figure 2 introduit un programme incrémentant une variable X et le graphe de flot de contrôle (CFG) correspondant dont chaque nœud est étiqueté par l'état abstrait obtenu après analyse. Soit l'invariant $\mathcal{X}_j^{\sharp i}$, avec i le nombre d'itérations (omis lorsque l'état stable est atteint) et j un point du programme. On note $\mathcal{X}_2^\sharp$ et $\mathcal{X}_5^\sharp$ les états abstraits d'entrée et respectivement de sortie de la boucle. L'opérateur d'élargissement standard est défini par :

$$\begin{aligned} \mathcal{X}_3^{\sharp 0} &= \mathcal{X}_2^\sharp \\ \mathcal{X}_3^{\sharp n+1} &= \mathcal{X}_3^{\sharp n} \ \triangledown^\sharp \ (\mathcal{X}_2^\sharp \sqcup^\sharp \mathcal{X}_5^{\sharp n}) \end{aligned} \tag{11}$$

On suppose que l'invariant de boucle $\mathcal{X}_3^{\sharp old}$ est sauvegardé dans un cache. Nous modifions l'opérateur d'élargissement standard pour réutiliser l'invariant de boucle sauvegardé $\mathcal{X}_3^{\sharp old}$ combiné avec l'état abstrait d'entrée de la boucle $\mathcal{X}_2^\sharp$:

$$\begin{aligned} \mathcal{X}_3^{\sharp 0} &= \mathcal{X}_3^{\sharp old} \sqcup^\sharp \mathcal{X}_2^\sharp \\ \mathcal{X}_3^{\sharp n+1} &= \mathcal{X}_3^{\sharp n} \ \triangledown^\sharp \ (\mathcal{X}_2^\sharp \sqcup^\sharp \mathcal{X}_5^{\sharp n}) \end{aligned} \tag{12}$$

On rappelle que nous pouvons initialiser l'analyse à partir d'un autre état que $\mathcal{X}_2^\sharp$. Nous

$X \leftarrow 0$
while • $X < 10$ **do**
  $X \leftarrow X + 1$
end while

(a) Programme initial

n	$\mathcal{X}_3^{\sharp n}$
0	$[0; 0]$
1	$[0; +\infty[$
2	$[0; +\infty[$

(b) Itérations de la boucle

$X \leftarrow -1$
while • $X < 10$ **do**
  $X \leftarrow X + 1$
end while

(c) Programme modifié

n	$\mathcal{X}_3^{\sharp n}$
0	$[0; +\infty[\sqcup^\sharp [-1; -1]$
1	$[-1; +\infty[$

(d) Itérations de la boucle

$X \leftarrow 1$
while • $X < 10$ **do**
  $X \leftarrow X + 1$
end while

(e) Programme modifié

n	$\mathcal{X}_3^{\sharp n}$
0	$[0; +\infty[\sqcup^\sharp [1; 1]$
1	$[0; +\infty[$

(f) Itérations de la boucle

Figure 3. Analyse d'une boucle sans (en haut) et avec (milieu) réutilisation d'invariant, et avec réutilisation mais perte de précision (bas).

obtiendrons un invariant correct à la fin, tant que le test de stabilité $\mathcal{X}_3^{\sharp n+1} = \mathcal{X}_3^{\sharp n}$ permettant de savoir si le point-fixe est atteint ou non, est réalisé correctement.

Considérons l'exemple en haut de la figure 3 pour lequel nous souhaitons analyser la boucle avec l'opérateur d'élargissement standard (11). Après $n = 2$ itérations, l'invariant de boucle $\mathcal{X}_3^\sharp$ converge vers $[0; +\infty[$. Cet invariant de boucle est sauvegardé dans un cache pour une réutilisation ultérieure. Considérons à présent l'exemple du milieu, où le programme a été modifié en changeant $X \leftarrow 0$ par $X \leftarrow -1$. Soit $\mathcal{X}_3^{\sharp old}$ l'invariant de boucle du programme initial. En analysant la boucle avec l'opérateur d'élargissement incrémentale (12), nous commençons l'itération avec $\mathcal{X}_3^{\sharp old} \sqcup^\sharp \mathcal{X}_2^\sharp = [0, +\infty[\sqcup^\sharp [-1, -1] = [-1; +\infty[$. Ainsi, nous constatons que les itérations convergent en une seule itération, ce qui évite une itération lors du calcul de point-fixe. Cela peut paraître peu, mais en pratique, le temps de calcul induit peut être long, surtout si le corps de la boucle est complexe.

Si l'invariant sauvegardé $\mathcal{X}_3^{\sharp old}$ est suffisamment proche de l'invariant de la boucle du nouveau programme, alors nous évitons des itérations abstraites et nous convergeons plus rapidement vers l'invariant de la nouvelle boucle. Cependant, dans le cas où cet invariant sauvegardé est différent du nouveau, nous pouvons introduire des faux positifs, et donc une perte de précision. Considérons maintenant l'exemple du bas, où l'on a modifié le programme en changeant $X \leftarrow 0$ par $X \leftarrow 1$. Ici aussi, les itérations convergent en une seule itération. Cependant, l'invariant calculé avec notre opérateur d'élargissement est moins précis que s'il avait été calculé avec l'élargissement standard, avec lequel il serait $[1; +\infty[$. Nous avons donc introduit une imprécision, car le nouvel invariant est plus précis que celui sauvegardé.

Amélioration de la précision L'invariant sauvegardé d'une boucle dans une fonction donnée peut ne pas être unique : il peut varier en fonction du contexte d'appel de la fonction ou même de la trace d'exécution, surtout si des techniques avancées comme le partitionnement de trace [MR05] sont employées. Par conséquent, il est possible et souvent avantageux de sauvegarder plusieurs invariants de boucle pour une seule et même fonction. Ainsi, le choix de l'invariant de boucle à réutiliser a un effet direct sur la précision de l'analyse. Pour identifier l'invariant le plus approprié, il est nécessaire de retrouver le contexte dans lequel la fonction est appelée. Plus précisément, des facteurs tels que les valeurs des arguments d'entrée, la pile d'appel et même les propriétés du chemin d'exécution peuvent influencer le choix de l'invariant de boucle à réutiliser. Nous verrons dans la section 6 comment on retrouve le bon contexte d'appel avec Eva pour limiter la perte de précision. On note que, malgré ces améliorations, cette approche d'analyse incrémentale avec réutilisation des invariants

de boucle ne peut pas toujours garantir le même niveau de précision qu'une analyse non incrémentale. Nous verrons néanmoins dans la section 6 que les pertes de précision sont limitées en pratique si les évolutions successives du code le sont aussi.

6 Implémentation et évaluation expérimentale

Implémentation L'implémentation de la recharge des résumés de fonction est relativement simple. En effet, Frama-C permet déjà de sauvegarder les résultats d'analyse dans un fichier dédié. Ainsi, la première étape a été de l'étendre pour y inclure également le cache de Memexec. Finalement, ce dernier est rechargé après la comparaison des arbres syntaxiques abstraits des deux versions du programme, permettant ainsi l'invalidation du cache pour les fonctions qui ont subi une modification.

Pour les invariants de boucle, l'implémentation nécessite une considération particulière du contexte d'appel des fonctions pour le stockage des invariants de boucle. Cela est crucial pour augmenter la précision de l'analyse incrémentale comme expliqué en fin de la section 5.3. Dans ce contexte, nous proposons de sauvegarder les invariants de boucle en se fondant sur la relation fonction $\rightarrow$ pile d'appel $\rightarrow$ arguments $\rightarrow$ partition $\rightarrow \bigsqcup^\sharp$ invariants de boucle. Ici, le contexte d'appel d'une fonction est représenté par sa pile d'appel, ses arguments et le partitionnement. L'opérateur $\bigsqcup^\sharp$ fusionne de multiples invariants en présence du même contexte d'appel. Cet opérateur résout le dilemme du choix de l'invariant lorsque plusieurs invariants de boucle coexistent dans un contexte d'appel identique.

Expérimentations Nos expérimentations visent à répondre aux questions de recherche suivantes :

RQ1 Est-ce que l'analyse incrémentale est plus rapide que l'analyse normale ?

RQ2 Est-ce que l'analyse incrémentale consomme plus de mémoire que l'analyse normale ?

RQ3 Est-ce que l'analyse incrémentale est aussi précise que l'analyse normale ?

RQ4 Est-ce que la réutilisation des invariants de boucle permet d'accélérer l'analyse des boucles dans les fonctions qui ont subi des modifications ?

RQ5 Quel est l'impact du temps de chargement du cache de Memexec et du calcul d'AST Diff sur l'analyse incrémentale ?

Pour cela, nous avons évalué la performance et la précision de notre approche en nous concentrant sur des critères tels que le temps d'analyse, la consommation mémoire et le nombre d'alarmes générées. Nos expérimentations reposent sur l'analyse de deux programmes : PolarSSL [1] et Monocypher [2] issus de la suite Open Source Case Studies (OSCS) [3]. Ces programmes sont chacun analysés successivement sur 10 commits. Seuls les commits incluant des modifications affectant les fichiers sources sont considérés. Il est à noter que ces programmes, écrits en langage C, avaient été préalablement paramétrés et modifiées pour pouvoir être analysés avec Eva. Les modifications consistent en l'ajout d'annotations de code permettant de guider l'analyse afin qu'elle termine en un temps raisonnable.

Pour chaque programme, nous avons d'abord effectué une analyse initiale avec Eva à partir d'un commit initial. Ensuite, nous avons lancé notre analyse incrémentale avec Eva sur les commits suivants. Les tableaux 1a et 1b résument respectivement les versions des commits analysés et les modifications (nombre de fichiers modifiés, nombre de lignes ajoutées et supprimées par diff) entre les commits pour les programmes PolarSSL et Monocypher. Nos approches ont été implémentées sur le greffon Eva dans une branche publique de la version 26.1 de la plateforme Frama-C. Les codes sources de Frama-C et les scripts utilisés pour les expérimentations sont disponibles ici [4]. Enfin, toutes ces expérimentations ont été menées sur un ordinateur équipé d'un processeur Intel Core i7-12800H cadencé à 2.4 GHz, ~~avec 64 Go de RAM et fonctionnant~~ sous le système d'exploitation Linux Manjaro 6.1.51-1.

1. `https://git.frama-c.com/pub/open-source-case-studies/-/tree/master/polarssl`
2. `https://git.frama-c.com/pub/open-source-case-studies/-/tree/master/monocypher`
3. `https://git.frama-c.com/pub/open-source-case-studies`
4. `https://gitlab.com/nyandrianinamamy/jfla2024`

Commit	LoC	Diff
a465d75	28784	1 file/+1
90f242b	28784	2 files/+3/-2
68514b0	28784	6 files/+10/-10
3f5b753	28784	2 files/+2/-1
8648f04	28784	2 files/+6/-1
16e5f81	28794	2 files/+14/-2
df177ba	28799	2 files/+10
3081ba1	28801	2 files/+6/-1
3513868	28807	2 files/+7
62dfcf0	28807	6 files/+10/-10

(a) PolarSSL

Commit	LoC	Diff
ea79193	27700	36 files/+1937
33ab0fe	27700	1 file/+1/-1
c3c74e2	27710	1 file/+57/-73
2c6b521	27710	1 file/+7/-7
310aab8	27708	3 files/+30/-33
57bacd2	27711	1 file/+5
94d0f70	27713	3 files/+33/-30
f5d90ef	27713	1 file /+6/-6
0bfbd92	27735	4 files/+146
a6fe62b	27759	4 files/+2802

(b) Monocypher

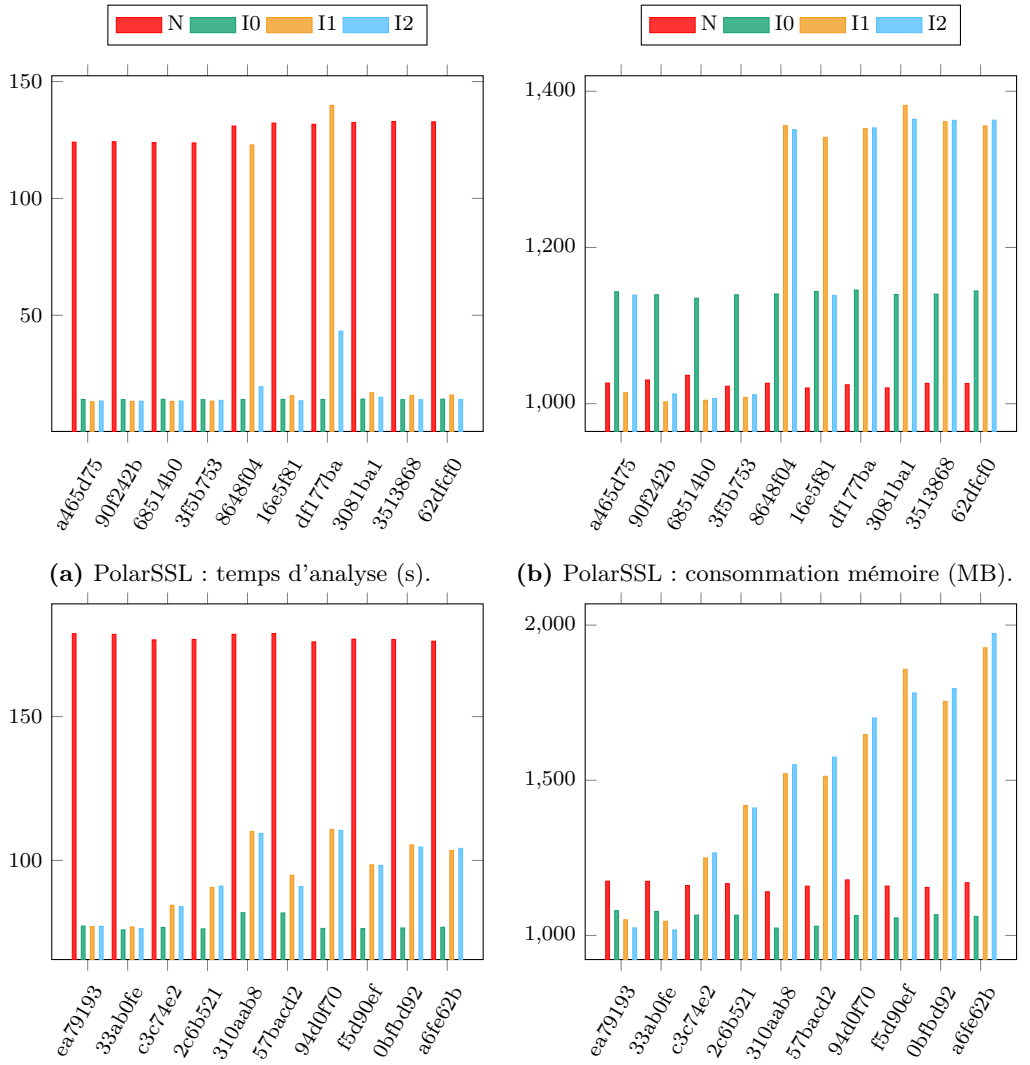
Table 1. Résumé des modifications. **LoC** indique le nombre de lignes de code du programme analysé et **Diff** indique le nombre de fichiers modifiés, ainsi que le nombre de lignes de code ajoutées et, le cas échéant, supprimées.

Évaluation Soit un ensemble de commits $C = \{c_1, \ldots, c_n\}$ et un ensemble de résultats d'analyse $R = \{r_1, \ldots, r_n\}$, où c_i est un commit d'un programme P, n est le nombre de commits et r_i est le résultat de l'analyse de c_i. Quatre résultats sont distingués pour les différentes analyses :

- **Normale N** : analyse normale de c_i sans réutilisation de résultats précédents.
- **Incrémentale I0** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions et des invariants de boucle dans r_i.
- **Incrémentale I1** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions dans r_{i-1}.
- **Incrémentale I2** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions et des invariants de boucle dans r_{i-1}.

Nous évaluons l'efficacité de nos approches en comparant les résultats d'analyse incrémentale avec une analyse normale. Pour les programmes PolarSSL et Monocypher, la figure 4 montre le temps d'analyse et la consommation en mémoire pour chacune des quatre configurations. Ensuite, les tableaux 2a et 2b montrent le nombre d'alarmes générées par chacun. Après, les tableaux 3a et 3b montrent le nombre d'itérations total par programme et le nombre d'invariants de boucle réutilisés. Enfin, les tableaux 3c et 3d montrent le nombre de résumés de fonctions rechargés par une analyse **I1** et le taux d'équivalence de fonctions non modifiées inféré par AST Diff entre chaque commit. L'analyse incrémentale **I0** représente le cas où l'on analyse un programme sur lui-même, c'est-à-dire que l'on réutilise les résultats d'une analyse normale pour l'analyse incrémentale du même programme sans modification. Ce cas permet d'obtenir une estimation du temps d'analyse minimal attendu pour une analyse incrémentale, en réutilisant le maximum de résultats sauvegardés. **I0** illustre également l'impact des limitations du cache de Memexec. L'interprétation des résultats est donnée en réponse aux questions de recherche ci-dessous.

RQ1 Pour le programme PolarSSL, nous obtenons en moyenne une analyse incrémentale **I0 : 9x**, **I1 : 7x** et **I2 : 8x** plus rapide que l'analyse normale **N**. Avec **I0**, le temps d'analyse est constitué en grande partie par le temps de chargement du cache de Memexec et du calcul AST Diff. Pour le programme Monocypher, nous obtenons en moyenne une analyse incrémentale **I0 : 2x**, **I1 : 1.8x** et **I2 : 1.9x** plus rapide que l'analyse normale **N**. Nous constatons avec **I0** que les résultats sont moins bons pour Monocypher, même si le programme n'a pas été modifié. En effet ce programme contient beaucoup de fonctions qui allouent de la mémoire dynamiquement. On rappelle que ces fonctions et les fonctions qui les appellent ne bénéficient pas du cache de Memexec incrémental et non incrémental. Inévitablement, chaque analyse incrémentale doit ré-analyser ces fonctions en entier.



(a) PolarSSL : temps d'analyse (s).

(b) PolarSSL : consommation mémoire (MB).

(c) Monocypher : temps d'analyse (s).

(d) Monocypher : consommation mémoire (MB).

Figure 4. Temps d'analyse et consommation mémoire sur PolarSSL et Monocypher.

RQ2 Pour le programme PolarSSL, nous obtenons une analyse incrémentale qui consomme au maximum pour **I0 : 1.1x**, **I1 : 1.33x** et **I2 : 1.31x** plus de mémoire que l'analyse normale **N**. Pour le programme Monocypher, nous obtenons une analyse incrémentale qui consomme au maximum pour **I0 : 0.91x**, **I1 : 1.63x** et **I2 : 1.67x** plus de mémoire que l'analyse normale **N**. Nous pouvons donc conclure que l'analyse incrémentale consomme plus de mémoire que l'analyse normale. Nous supposons que cette augmentation est due au rapport entre le nombre de résultats rechargés et le taux de réutilisation des résultats. Un taux de réutilisation plus faible implique un nombre plus élevé de création de nouveaux résultats, ce qui augmente la consommation mémoire. Le tableau 3 montre le nombre de résumés de fonctions rechargés.

RQ3 L'analyse incrémentale **I2** des commits `df177ba` et `2c6b521` pour les programmes PolarSSL et Monocypher introduisent chacun une alarme en plus (respectivement 93 contre 92, et 179 contre 178) par rapport à une analyse normale. Nous pouvons donc conclure que l'analyse incrémentale n'est pas aussi précise que l'analyse normale. Cependant, la différence est minime, avec de l'ordre de un pourcent d'alarmes supplémentaires au maximum.

Commit	N	I0	I1	I2
df177ba	92	92	92	93

(a) PolarSSL

Commit	N	I0	I1	I2
2c6b521	178	178	178	179

(b) Monocypher

Table 2. Alarmes générées par les analyses. Seuls les commits qui génèrent plus d'alarmes que l'analyse non incrémentale sont affichés.

Commit	I1	I2	Reuse
a465d75	9	9	0
90f242b	9	9	0
68514b0	9	9	0
3f5b753	58	21	1
8648f04	181133	9170	35
16e5f81	9	9	0
df177ba	200416	46271	477
3081ba1	9	9	0
3513868	9	9	0
62dfcf9	9	9	0

(a) PolarSSL

Commit	I1	I2	Reuse
ea79193	29713	29170	218
33ab0fe	29713	29170	218
c3c74e2	33042	32167	281
2c6b521	48406	44838	774
310aab8	46899	45832	488
57bacd2	31179	30488	243
94d0f70	44927	43796	461
f5d90ef	29713	29170	218
0bfbd92	29713	29170	218
a6fe62b	31950	31328	57

(b) Monocypher

Commit	Reloaded	Equiv(%)
a465d75	12898	100.0
90f242b	12909	100.0
68514b0	12920	100.0
3f5b753	12908	95.0
8648f04	12774	90.0
16e5f81	21202	100.0
df177ba	21190	95.0
3081ba1	32937	100.0
3513868	32948	100.0
62dfcf0	32959	100.0

(c) PolarSSL

Commit	Reloaded	Equiv(%)
ea79193	43537	100
33ab0fe	44027	100
c3c74e2	43567	94
2c6b521	41802	77
310aab8	42007	89
57bacd2	44361	96
94d0f70	43436	89
f5d90ef	46019	100
0bfbd92	46509	100
a6fe62b	46992	97

(d) Monocypher

Table 3. En haut, nombre d'itérations pour **I1** et **I2** et nombre d'invariants réutilisés par **I2**. En bas, nombre de résumés rechargés par **I1** et taux d'équivalence de fonctions non modifiées inféré par AST Diff entre chaque commit.

RQ4 L'analyse incrémentale **I2** est en moyenne **1.8x** plus rapide que l'analyse incrémentale **I1** de PolarSSL et pas de différence en temps pour Monocypher. Nous constatons en particulier pour les commits `8648f04` et `df177ba` de PolarSSL une différence significative car les modifications apportées à ces commits affectent une fonction profondément imbriquée dans la pile d'appel, ce qui invalide plusieurs caches de résumés de fonctions et rend l'analyse incrémentale **I1** moins rapide. L'analyse incrémentale **I2**, quant à elle, est plus rapide, car elle réutilise les invariants de boucle pour accélérer l'analyse de ces fonctions modifiées.

RQ5 Le temps de chargement maximal du cache de Memexec et le calcul d'AST Diff sont respectivement de **9s** et **6s** pour PolarSSL, et de **32s** et **3s** pour Monocypher. Le chargement des emplacements mémoires lus et écrits par chaque fonction est particulièrement long pour Monocypher en raison des allocations dynamiques de mémoire, augmentant le nombre d'emplacements mémoires et le temps de chargement du cache.

Facteurs de risque pour la validité La validité des résultats de cette étude peut être affectée de plusieurs manières. D'abord, l'analyse n'a été effectuée que sur des modifications qui affectent des fichiers sources entre chaque commit. Par conséquent, l'efficacité de l'analyse pour de grandes différences de versions n'a pas été vérifiée, ce qui limite la généralisabilité des résultats, surtout pour des scénarios où les changements sont importants. Ensuite, les effets des modifications des paramètres d'analyse n'ont pas été explorés. Les résultats obtenus

pourraient donc varier significativement en les modifiant. Enfin, l'étude n'a été réalisée que sur deux programmes, ce qui soulève des questions sur sa généralisabilité. Les caractéristiques spécifiques de ces programmes pourraient avoir influencé les résultats, rendant ainsi leur applicabilité à d'autres programmes incertaine.

7 Conclusion et perspectives

Cet article démontre l'intérêt de l'analyse incrémentale pour optimiser le temps d'analyse et la consommation mémoire. Nous avons proposé deux approches différentes, mais complémentaires pour réutiliser de manière sûre et efficace les résultats d'analyse précédents. La première réutilise les résumés de fonction pour les fonctions qui n'ont pas subi de modifications et n'appellent pas des fonctions qui ont été modifiées. La seconde, permet de réutiliser des invariants de boucle pour accélérer l'analyse des boucles d'une fonction modifiée. Nous avons également présenté une technique de comparaison d'AST pour identifier les fonctions non modifiées, ainsi que la correspondance entre les boucles des fonctions modifiées. Ces approches ont été implémentées sur le greffon Eva de la plateforme Frama-C. Les résultats expérimentaux montrent que l'analyse incrémentale permet d'améliorer l'efficacité en temps de l'analyse statique tout en garantissant la sûreté et en conservant une grande précision.

Suite à ces travaux, plusieurs axes de recherche sont possibles. D'abord, nous souhaitons formaliser les approches présentées dans cet article afin d'établir une base théorique solide. Cette formalisation permettra de démontrer la correction des méthodes proposées et leur applicabilité dans un contexte plus large.

Ensuite, nous prévoyons d'appliquer la même approche que celle utilisée pour les résumés de fonctions aux résumés de boucles, ou plus généralement à n'importe quel bloc de code. Cette extension devrait permettre d'éviter d'analyser le corps d'une boucle (ou d'un bloc de code) lorsqu'un état d'entrée identique est rencontré. Le choix de la boucle ou du bloc de code à résumer pourra être fait à l'aide d'heuristique choisit automatiquement, ou manuellement par des experts en analyse statique.

De plus, les résultats d'analyse sont actuellement rechargés au début de chaque nouvelle analyse, ce qui peut être coûteux en mémoire et en temps. Pour y remédier, une recharge partielle ou différée des résultats d'analyse du cache est envisagée, où seul le cache de la fonction en cours d'analyse serait rechargé.

Par ailleurs, les analyseurs statiques sont également utilisés après la phase initiale du développement, généralement par une équipe distincte dont l'objectif est de certifier l'absence d'erreur à l'exécution. Pour cela, les paramètres de l'analyseur sont finement ajustés de manière incrémentale afin d'améliorer la précision. Le coût de ce processus étant d'un ordre de grandeur supérieure à celui de la précédente analyse, nous souhaitons adapter notre approche à cette méthodologie reposant sur un paramétrage incrémental de l'analyseur. Toutefois, il est crucial non seulement d'éviter une perte de précision résultant de l'incrémentalité, comme dans le cas actuel, mais également de chercher à améliorer la précision.

Par la suite, nous envisageons d'étendre l'utilisation du cache de résumé de fonction pour les fonctions qui allouent dynamiquement de la mémoire. Cette extension permettra de couvrir un plus grand nombre de fonctions afin d'augmenter la réutilisation du cache.

Enfin, notre analyse incrémentale est actuellement dépendante de l'analyse complète du programme afin d'assurer la sûreté de l'analyse. Nous cherchons à intégrer une approche plus fondamentale fondée sur l'analyse modulaire, inspirée de travaux de P. et R. Cousot [CC02], pour réduire cette dépendance. Cette approche vise à définir une analyse incrémentale tirant parti de la modularité de l'analyse, afin de réduire les parties du programme à ré-analyser.

Remerciements. Ce travail a bénéficié d'une aide de l'État gérée par l'Agence Nationale de la Recherche au titre de France 2030 portant la référence 'ANR-22-PECY-0005'. Nous remercions également les relecteurs anonymes pour leurs remarques bienveillantes ayant permises d'améliorer notre article.

Références

[AB14] Steven Arzt et Eric Bodden : Reviser : efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. *In International Conference on Software Engineering (ICSE)*, Hyderabad India, mai 2014.

[ALL96] Martin Abadi, Butler Lampson et Jean-Jacques Lévy : Analysis and caching of dependencies. *SIGPLAN Not.*, jun 1996.

[BBB+21] 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. *Communications of the ACM*, 2021.

[BBY17] Sandrine Blazy, David Bühler et Boris Yakobowski : *In International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2017.

[CB16] Maria Christakis et Christian Bird : What developers want and need from program analysis : an empirical study. *In International Conference on Automated Software Engineering (ASE)*, août 2016.

[CC77] Patrick Cousot et Radhia Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Symposium on Principles of programming languages (POPL)*, 1977.

[CC02] Patrick Cousot et Radhia Cousot : Modular Static Program Analysis. *In International Conference on Compiler Construction (CC)*. 2002.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Martino Luca, Peter O'Hearn, Irene Papakonstantinou et Dulma Rodriguez : Moving Fast with Software Verification. 2015.

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter O'Hearn et Hongseok Yang : Compositional Shape Analysis by means of Bi-Abduction. *Journal of the ACM*, décembre 2011.

[Cou21] Patrick Cousot : Principles of Abstract Interpretation. sep 2021.

[DAL+17] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith et Emerson Murphy-Hill : Just-in-time static analysis. *In International Symposium on Software Testing and Analysis (ISSTA)*, juillet 2017.

[DM19] David Delmas et Antoine Miné : Analysis of Software Patches Using Numerical Abstract Interpretation. *In Static Analysis*, volume 11822. Springer International Publishing, Cham, 2019.

[DP16] Georg Dotzler et Michael Philippsen : Move-optimized source code tree differencing. *In International Conference on Automated Software Engineering (ASE)*, 2016.

[EST+22] Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani et Helmut Seidl : Interactive abstract interpretation : Reanalyzing whole programs for cheap, 2022.

[FMB+14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez et Martin Monperrus : Fine-grained and accurate source code differencing. *In International Conference on Automated Software Engineering (ASE)*, 2014.

[FT90] John Field et Tim Teitelbaum : Incremental reduction in the lambda calculus. *In Conference on LISP and Functional Programming (LFP)*, mai 1990.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill et Robert Bowdidge : Why don't software developers use static analysis tools to find bugs ? *In International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, mai 2013.

[MD15] Antoine Miné et David Delmas : Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. *In International Conference on Embedded Software (EMSOFT)*, octobre 2015.

[MGR13] Scott McPeak, Charles-Henri Gros et Murali Krishna Ramanathan : Scalable and incremental software bug detection. *In Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.

[MOJ18] Antoine Miné, Abdelraouf Ouadjaout et Matthieu Journault : Design of a Modular Platform for Static Analysis. *In Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2018.

[MOM20] Raphaël Monat, Abdelraouf Ouadjaout et Antoine Miné : Static Type Analysis by Abstract Interpretation of Python Programs. *In European Conference on Object-Oriented Programming (ECOOP)*, 2020.

[MR05] Laurent Mauborgne et Xavier Rival : Trace partitioning in abstract interpretation based static analyzers. *In* Mooly Sagiv, éditeur : *Symposium on Programming Languages and Systems (APLAS)*, 2005.

[NEH19] Lawton Nichols, Mehmet Emre et Ben Hardekopf : Fixpoint reuse for incremental JavaScript analysis. *In International Workshop on State Of the Art in Program Analysis (SOAP)*, juin 2019.

[O'H18] Peter W. O'Hearn : Continuous Reasoning : Scaling the impact of formal methods. *In Symposium on Logic in Computer Science (LICS)*, juillet 2018.

[Ryd83] Barbara G. Ryder : Incremental data flow analysis. *In Symposium on Principles of programming languages (POPL)*, 1983.

[SCS21] Benno Stein, Bor-Yuh Evan Chang et Manu Sridharan : Demanded abstract interpretation. *In International Conference on Programming Language Design and Implementation (PLDI)*, juin 2021.

[SEV16] Tamás Szabó, Sebastian Erdweg et Markus Voelter : IncA : a DSL for the definition of incremental program analyses. *In International Conference on Automated Software Engineering (ASE)*, août 2016.

[SEV20] Helmut Seidl, Julian Erhard et Ralf Vogler : Incremental Abstract Interpretation. *In From Lambda Calculus to Cybersecurity Through Program Analysis*, volume 12065. 2020.

[SV21] Helmut Seidl et Ralf Vogler : Three improvements to the top-down solver. *Mathematical Structures in Computer Science*, octobre 2021.

[VdPSVEDR20] Jens Van der Plas, Quentin Stievenart, Noah Van Es et Coen De Roover : Incremental Flow Analysis through Computational Dependency Reification. *In International Working Conference on Source Code Analysis and Manipulation (SCAM)*, septembre 2020.

[VJL07] Jan Wen Voung, Ranjit Jhala et Sorin Lerner : RELAY : static race detection on millions of lines of code. *In Joint Meeting of European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC-FSE)*, septembre 2007.

[Yak15] Boris YAKOBOWSKI : Fast whole-program verification using on-the-fly summarization. *In International Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2015.