REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE ABOU-BEKR BELKAID – TLEMCEN

Laboratoire de Recherche en Informatique de Tlemcen

THÈSE LMD

Présentée à :

FACULTE DES SCIENCES – DEPARTEMENT D'INFORMATIQUE

Pour l'obtention du diplôme de :

DOCTORAT

Spécialité: Ingénierie des systèmes d'information et de connaissance et aide à la décision

Par :

Kissi Salim Yahia

Sur le thème

LA VÉRIFICATION FORMELLE DES SYSTÈMES EMBARQUÉS EN PRENANT EN COMPTE LES PROPRIÉTÉS DE SÛRETÉ ET DE SÉCURITÉ.

Thèse dirigée par Seladji Yassamine & Ameur-Boulifa Rabéa

Soutenue publiquement le 17 Juin 2026 à Tlemcen devant le jury composé de :

Président du jury
Dr. Hocine Matallah – LRIT – Département d'informatique, Université de Tlemcen, Algérie

Directeur de thèse
Pr. Yassamine Seladji – LRIT – Département d'informatique, Université de Tlemcen, Algérie

Co-Directeur de thèse
Pr. Rabéa Ameur-Boulifa – LTCI – Télécom Paris, Institut Polytechnique de Paris, France

Examinateurs
Pr. Nabil BELALA – MISC – Université Constantine 2 Abdelhamid Mehri, Algérie
Pr. Yahia Lebbah – LITIO – Département d'informatique, Université d'Oran 1, Algérie
Dr. Houari Mahfoud – LRIT – Département d'informatique, Université de Tlemcen, Algérie

Je dédie ce travail à mes chers parents,
à mon père **Abdellah Mounir** et à ma mère
Amina Rachedi, pour leur amour inconditionnel,
leur soutien constant et les sacrifices qu'ils ont
consentis tout au long de mon parcours.

Je le dédie également à ma famille,
pour sa présence, sa compréhension
et ses encouragements permanents.

Enfin, je dédie ce travail à toutes les personnes
qui ont cru en moi et m'ont soutenu,
de près ou de loin, dans la réalisation de cette thèse.

If I have seen further, it is by standing on the shoulders of Giants.
Sir Isaac Newton

Remerciements

Je rends grâce à Dieu Tout-Puissant pour m'avoir accordé la force, la patience et la persévérance nécessaires à l'aboutissement de ce travail de recherche. Sans Sa volonté et Son soutien, la réalisation de cette thèse n'aurait pas été possible.

J'adresse mes sincères remerciements aux membres du jury, **Dr. Hocine Matallah**, **Pr. Nabil Belala**, **Pr. Yahia Lebbah** et **Dr. Houari Mahfoud**, pour avoir accepté d'évaluer ce travail et pour le temps qu'ils ont consacré à la lecture de ce manuscrit.

Je tiens à exprimer ma profonde gratitude à ma directrice de thèse, **Madame Yassamine Seladji**, pour la confiance qu'elle m'a accordée, pour son encadrement rigoureux, ses conseils précieux et sa disponibilité tout au long de ce travail. Je remercie également ma co-directrice de thèse, **Madame Ameur-Boulifa Rabéa**, pour son accompagnement attentif, ses remarques pertinentes, son soutien scientifique constant et sa disponibilité tout au long de ce travail qui ont largement contribué à l'amélioration de la qualité de cette recherche.

J'adresse mes sincères remerciements à **Monsieur Azeddine Chikh**, directeur du laboratoire, pour son accueil, pour l'environnement de travail favorable qu'il a su instaurer afin de mener cette recherche dans de bonnes conditions.

Je remercie l'ensemble de mes enseignants pour la qualité de leur formation, leurs enseignements enrichissants et les connaissances qu'ils m'ont transmises tout au long de mon parcours universitaire. Mes remerciements vont également à mes collègues pour les échanges scientifiques, l'entraide et l'esprit de collaboration qui ont accompagné ce travail.

Je souhaite exprimer toute ma reconnaissance à ma famille, en particulier à mon père **Abdellah Mounir** et à ma mère **Amina Rachedi**, pour leur amour, leur soutien indéfectible, leurs sacrifices et leurs encouragements constants tout au long de mes études. Leur présence et leur confiance ont constitué une source essentielle de motivation.

Enfin, je remercie chaleureusement mes collègues ainsi que mes amis pour leur soutien moral, leur compréhension et les moments de partage qui ont contribué, chacun à leur manière, à la réussite de ce travail.

Résumé

De nos jours, les systèmes embarqués occupent une place centrale dans de nombreux secteurs tels que le transport, la santé, l'industrie, l'agriculture ou la défense. Leur intégration croissante dans des applications critiques — à l'image des taxis autonomes, des drones militaires ou des dispositifs médicaux intelligents — accentue leur impact sur la société et renforce la nécessité de garantir leur fiabilité. Cette évolution impose à l'être humain de s'adapter à ces technologies tout en conservant un contrôle rigoureux sur leur comportement. Dans ce contexte, les enjeux liés à la sûreté et à la sécurité des systèmes embarqués deviennent majeurs, en particulier face à la complexité croissante des environnements matériels et logiciels.

Cette thèse s'intéresse à la vérification formelle de propriétés de sûreté et de sécurité dans les programmes écrits en langage C, largement utilisés pour le développement de systèmes embarqués. Le problème abordé concerne la détection statique de vulnérabilités et d'erreurs critiques, notamment celles liées aux calculs numériques, à la gestion des entiers et aux dépendances vis-à-vis de l'environnement d'exécution. Les approches classiques d'analyse statique peinent à capturer avec précision l'influence des caractéristiques matérielles, du compilateur et du système d'exploitation sur le comportement réel des programmes.

Pour répondre à ces limitations, cette thèse propose une approche formelle fondée sur la transformation du programme source en un modèle logique exploitable par des solveurs *Satisfiability Modulo Theories* (SMT). L'approche repose sur une analyse statique basée sur la transformation en forme *Static Single Assignment* (SSA), le calcul des conditions de chemin, ainsi que l'intégration explicite de contraintes liées à l'architecture d'exécution. Une base de connaissances décrivant les paramètres matériels et logiciels est exploitée afin d'instancier automatiquement des assertions de sûreté et de sécurité. Les contributions de ce travail incluent un cadre d'analyse sensible au contexte d'exécution, une modélisation formelle des comportements numériques dépendants de l'architecture, ainsi qu'un prototype permettant de détecter efficacement des violations de propriétés critiques dès la phase de développement.

Mots clefs: Méthodes formelles, Analyse statique, Sécurité et sûreté logicielles, Solveurs SMT, Modélisation du contexte d'exécution, Débordements d'entiers.

Abstract

Nowadays, embedded systems play a central role in many domains such as transportation, healthcare, industry, agriculture, and defense. Their increasing integration into critical applications—such as autonomous taxis, military drones, and intelligent medical devices—amplifies their societal impact and reinforces the need to ensure their reliability. This evolution requires humans to adapt to these technologies while maintaining strict control over their behavior. In this context, safety and security issues in embedded systems become paramount, particularly in light of the growing complexity of hardware and software execution environments.

This thesis focuses on the formal verification of safety and security properties in programs written in the C language, which is widely used in embedded system development. The addressed problem concerns the static detection of vulnerabilities and critical errors, especially those related to numerical computations, integer handling, and dependencies on the execution environment. Classical static analysis approaches often struggle to accurately capture the impact of hardware characteristics, compilers, and operating systems on the actual behavior of programs.

To overcome these limitations, this thesis proposes a formal approach based on transforming the source program into a logical model suitable for *Satisfiability Modulo Theories* (SMT) solvers. The approach relies on a static analysis framework using *Static Single Assignment* (SSA) form, path condition computation, and the explicit integration of execution architecture constraints. A knowledge base describing hardware and software parameters is leveraged to automatically instantiate safety and security assertions. The contributions of this work include an execution-context-aware analysis framework, a formal modeling of architecture-dependent numerical behaviors, and a prototype capable of efficiently detecting violations of critical properties at early stages of development.

Keywords: Formal methods, Static analysis, Software security and safety, SMT solvers, Execution context modeling, Integer overflows.

الملخّص

أصبحت الأنظمة المدمجة في الوقت الراهن عنصرًا محوريًا في عدد متزايد من القطاعات، مثل النقل، والصحة، والصناعة، والزراعة، والدفاع. وقد أدى تزايد دمجها في تطبيقات حساسة — كسيارات الأجرة ذاتية القيادة، والطائرات المسيّرة العسكرية، والأجهزة الطبية الذكية — إلى تعاظم أثرها المجتمعي، الأمر الذي يجعل ضمان موثوقيتها مطلبًا أساسيًا لا غنى عنه. وتفرض هذه التحولات على الإنسان التكيّف مع هذه التقنيات الحديثة، مع الحفاظ في الوقت نفسه على تحكّم صارم في سلوكها. وفي هذا السياق، تكتسب قضايا السلامة والأمن في الأنظمة المدمجة أهمية بالغة، ولا سيما في ظل التعقيد المتزايد للبيئات العتادية والبرمجية التي تعمل ضمنها.

تتناول هذه الأطروحة مسألة التحقق الشكلي من خصائص السلامة والأمن في البرامج المكتوبة بلغة C، التي تُعد من أكثر اللغات استخدامًا في تطوير الأنظمة المدمجة. وينصبّ الاهتمام على الكشف الساكن عن الثغرات والأخطاء الحرجة، وبخاصة تلك المرتبطة بالحسابات العددية، وإدارة القيم الصحيحة، والاعتماديات المتعلقة ببيئة التنفيذ. وتُظهر المقاربات التقليدية في التحليل الساكن قصورًا في تمثيل التأثير الفعلي لخصائص العتاد، والمُصرِّف (Compiler)، ونظام التشغيل على السلوك الحقيقي للبرامج.

ولمعالجة هذه المحدوديات، تقترح هذه الأطروحة مقاربة شكلية تقوم على تحويل البرنامج المصدري إلى نموذج منطقي قابل للمعالجة بواسطة محلّلات الإشباع القابلة للتوسيع (SMT). وترتكز هذه المقاربة على تحليل ساكن يعتمد على التحويل إلى الصيغة ذات الإسناد الأحادي (SSA)، وحساب شروط المسارات التنفيذية، فضلًا عن الدمج الصريح للقيود المرتبطة بهندسة التنفيذ. كما يتم توظيف قاعدة معارف تصف المعلمات العتادية والبرمجية بهدف توليد تأكيدات السلامة والأمن بصورة آلية. وتشمل إسهامات هذا العمل إطارًا تحليليًا واعيًا بسياق التنفيذ، ونمذجة شكلية للسلوكيات العددية المعتمدة على البنية المعمارية، إلى جانب نموذج أولي يتيح الكشف الفعّال عن انتهاكات الخصائص الحرجة في المراحل المبكرة من عملية التطوير.

الكلمات المفتاحية: الطرق الشكلية، التحليل الثابت، أمن وسلامة البرمجيات، نمذجة سياق التنفيذ، فيض الأعداد الصحيحة، محللات SMT.

Nos Contributions

Article de journal

– Salim Yahia Kissi, Rabéa Ameur-Boulifa et Yassamine Seladji. **Identifying Security Vulnerabilities in Source Code with Safety Verification**. *International Journal of Critical Computer-Based Systems* (IJCCBS), 2025, pp. 45–70.*(Online, 26 May 2026)* doi:10.1504/IJCCBS.2025.10075749.

Outil développé

– `AETHER` (*Architecture-aware Execution and THEoretical Reasoning*). Outil d'analyse statique de programmes `C/C++` offrant aux développeurs un cadre permettant de vérifier si un programme présente des failles de sécurité induites par des problèmes de sûreté dans un environnement d'exécution particulier, en s'appuyant sur l'infrastructure `Clang/LLVM` et le solveur SMT Z3.
Dépôt : `https://github.com/NazimKYS/AETHER`

Articles de conférences internationales

– Salim Yahia Kissi, Yassamine Seladji et Rabéa Ameur-Boulifa. **Detection of Security Vulnerabilities Induced by Integer Errors**. In *Proceedings of the 16th International Conference on Software Technologies* (ICSOFT 2021), SciTePress, INSTICC, 2021, pp. 177–184. doi:10.5220/0010551301770184.

– Salim Yahia Kissi, Rabéa Ameur-Boulifa et Yassamine Seladji. **Security Vulnerabilities Detection Through Assertion-Based Approach**. In *Theoretical Aspects of Software Engineering: 16th International Symposium* (TASE 2022), Cluj-Napoca,

Romania, July 8–10, 2022. Springer-Verlag, Berlin, Heidelberg, 2022, pp. 381–387. doi:10.1007/978-3-031-10363-6_25.

– Salim Yahia Kissi, Yassamine Seladji et Rabéa Ameur-Boulifa. **Detection of Leaks Through Exception Mechanisms**. In *International Conference on Advanced Aspects of Software Engineering* (ICAASE 2022), Constantine, Algeria, September 17–18, 2022. IEEE, 2022, pp. 1–8. doi:10.1109/ICAASE56196.2022.9931592.

– Salim Yahia Kissi, Yassamine Seladji et Rabéa Ameur-Boulifa. **Support for Detecting Integer Overflow Vulnerability**. In *Journées 2022 du GT "Méthodes Formelles pour la Sécurité"*, GdR Sécurité Informatique, mars 2022, pp. 34–35.*(Extended abstract)* doi:10.13140/RG.2.2.24226.71363.

Table des Matières

Table des Figures

Liste des Tableaux

Liste des Algorithmes

Liste des Abbréviations

ABI – *Application Binary Interface* 75, 76, 78
ACSL – ANSI/ISO C Specification Language 51
AETHER – Architecture-aware Execution and THEoretical Reasoning 13, 24, 88, 89, 103, 104, 106
AMD – Advanced Micro Devices 79
ASLR – *Address Space Layout Randomization* 75
AST – *Abstract Syntax Tree* 13, 24, 31, 36, 52, 63, 66, 84, 89, 91–94, 106, 109

BV – Bit-Vector 14, 81–83

CDCL – Conflict-Driven Clause Learning 44
CFG – *Control Flow Graph* 37, 45
CFI – Control Flow Integrity 37
CI – Intégration Continue 47
CIL – C Intermediate Language 51
CPU – *Central Processing Unit* 110
CTL – Logique Computationnelle des Arbres 45
CVE – Common Vulnerabilities and Exposures 20, 21, 57
CWE – *Common Weakness Enumeration* 36, 77

DCE – Dead Code Elimination 32
DPLL – Davis–Putnam–Logemann–Loveland 44
DSE – Dynamic Symbolic Execution 41
DTA – Dynamic Taint Analysis 39

EVA – Evolved Value Analysis 52

FOL – First-Order Logic 43, 44

GCC – GNU Compiler Collection 51, 65, 76, 78, 79, 108
GPU – Graphics Processing Unit 110

ILP – Instruction Level Parallelism 34
IR – Intermediate Representation 32, 34
ISO – *International Organization for Standardization* 78

JSON – JavaScript Object Notation 84, 86

LLVM – Low Level Virtual Machine 24, 48, 88, 89, 109
LLVM-IR – LLVM Intermediate Representation 48, 49, 51, 72
LTL – Logique Temporelle Linéaire 45

MMU – *Memory Management Unit* 75
MSVC – Microsoft Visual C++ 65, 76, 78, 79, 108

OS – Operating System 75, 76, 78, 79, 84

POSIX – Portable Operating System Interface 49, 75

QF_LIA – Quantifier-Free Linear Integer Arithmetic 82
QF_NIA – Quantifier-Free Non-Linear Integer Arithmetic 14, 81–83

RTE – Run-Time Errors 52

SMT – *Satisfiability Modulo Theories* 5, 6, 11, 12, 15, 23, 24, 33, 40–42, 44, 49, 53, 55, 59, 61, 62, 67, 69–72, 74, 79–83, 85–87, 103, 108
SSA – *Static Single Assignment* 5, 6, 11, 15, 32–34, 55, 59, 61, 62, 71, 87–89, 94–100, 103, 106, 109

TDD – Test-Driven Development 47

V&V – Vérification et Validation 46

1 Introduction

Sommaire

1.1 Préambule

Nous vivons une époque où le logiciel occupe une place centrale dans tous les aspects de notre vie quotidienne et professionnelle. Aucun secteur n'échappe à cette évolution profonde : santé, transport, finance, énergie, ou encore défense. L'avènement des technologies intelligentes marque une transition majeure : nous sommes passés d'un monde oú **l'homme assistait la machine** dans sa prise de décision, à un monde oú **la machine assiste désormais l'homme**. Aujourd'hui, nous atteignons une nouvelle ère : celle où **les machines assistent d'autres machines** dans leur prise de décision, illustrant ainsi l'essor des systèmes autonomes et interconnectés.

Cette évolution se manifeste à travers plusieurs exemples emblématiques : les voitures autonomes, les drones automatisés, ou encore les systèmes intelligents, qui s'imposent progressivement dans notre quotidien et modifient en profondeur notre rapport à la technologie. Cependant, cette dépendance accrue au logiciel s'accompagne de défis majeurs, tant sur le plan technique que sociétal, qui se multiplient à mesure que la complexité des systèmes croit.

Parmi ces défis, deux préoccupations majeures émergent : la **sûreté** et la **sécurité**. Bien qu'elles soient complémentaires, ces notions répondent à des problématiques distinctes. Une **propriété de sûreté** garantit qu'aucun incident néfaste ne se produise, en prévenant les erreurs accidentelles ou les comportements imprévus, même en cas de défaillance matérielle ou de conditions extrêmes [75]. En revanche, une **propriété de sécurité** a pour objectif de protéger le système contre des menaces **intentionnelles**, empêchant l'exploitation malveillante du système, qu'il s'agisse d'**attaques**, d'intrusions ou de mauvaises manipulations [75].

Dans le contexte des défis croissants liés à la sûreté et à la sécurité des systèmes embarqués, les récents incidents mettent en lumière l'ampleur des risques associés à ces technologies modernes. En 2015, deux chercheurs en cybersécurité ont **pris le contrôle à distance** d'une voiture *Jeep Cherokee* via son système multimédia *Uconnect*, manipulant des fonctions critiques telles que **le freinage et l'accélération** [37]. Ils ont d'abord identifié la cible en exploitant sa connexion au réseau Sprint, puis ont profité d'une vulnérabilité dans la puce *Uconnect* pour exécuter du code malveillant. Grâce à un *firmware* personnalisé installé à distance sur la puce *Renesas V850*, qui sert d'interface avec les modules physiques du véhicule, ils ont pu prendre le contrôle de fonctions essentielles comme la direction et le freinage. Cette faille a conduit au rappel de 1,4 million de véhicules, soulignant ainsi les enjeux de **sécurité** qui en découlent. Dans un autre registre, en 2018, la découverte des vulnérabilités **Meltdown** [64] et

Spectre [58] a révélé des failles critiques dans les processeurs modernes [1], affectant des milliards d'appareils. Ces attaques exploitent des mécanismes d'optimisation matérielle, tels que l'exécution spéculative, pour contourner l'isolation mémoire et permettre à un attaquant de récupérer des informations sensibles, telles que des mots de passe ou des clés cryptographiques. Ces vulnérabilités ont démontré que **la frontière entre sûreté et sécurité est poreuse** : une faiblesse architecturale, initialement conçue pour optimiser les performances, peut se transformer en une faille exploitable, compromettant ainsi l'intégrité des systèmes.

Ces événements mettent en évidence à quel point la sûreté et la sécurité sont devenues des exigences indissociables dans la conception des systèmes embarqués critiques. Une vulnérabilité de sécurité peut compromettre la sûreté d'un système, tout comme une faiblesse en matière de sûreté peut être exploitée à des fins malveillantes et ouvrir la voie à des cyberattaques. Leur interindépendance rend indispensable une approche intégrée, où les deux dimensions sont considérées conjointement dès les premières phases de conception.

1.2 Motivation

Garantir à la fois la robustesse face aux erreurs involontaires et la résilience face aux attaques intentionnelles constitue désormais un enjeu majeur. Cette double exigence justifie le recours à des méthodes et des outils capables de traiter conjointement les problématiques de sûreté et de sécurité.

Dans un contexte où les systèmes embarqués gagnent en complexité, les défis associés à leur fiabilité, leur sûreté et leur sécurité ne cessent de croître. Bugs logiciels, vulnérabilités de sécurité et défaillances critiques continuent de représenter des menaces sérieuses, y compris dans des secteurs stratégiques.

Pour mieux cerner l'ampleur de cette menace, nous avons analysé l'évolution des vulnérabilités répertoriées sur la plateforme CVE (Common Vulnerabilities and Exposures) [34]. La Figure 1.1 illustre cette progression en retraçant le nombre total de vulnérabilités signalées entre 2015 et 2024. Cette analyse révèle une augmentation exponentielle au cours des dernières années. Depuis 2015, plus de 100 000 vulnérabilités ont été enregistrées, et rien qu'en 2022, plus de 25 000 nouvelles failles ont été identifiées — soit une hausse de 22% par rapport à 2021.

Cette explosion du nombre de vulnérabilités s'explique par une conjonction de facteurs. D'une part, l'essor de l'Internet des objets (IoT) a considérablement multiplié le nombre

de systèmes embarqués déployés dans des contextes variés, élargissant ainsi la surface d'attaque et les opportunités d'exploitation pour les attaquants. D'autre part, la communauté de la cybersécurité a gagné en maturité, tirant parti de l'expérience accumulée pour perfectionner les techniques d'analyse et de détection. Cela s'est traduit par une augmentation significative du nombre de vulnérabilités signalées.

Parmi les exemples les plus emblématiques, les vulnérabilités Meltdown et Spectre, évoquées dans le préambule, illustrent les risques critiques posés par les failles ancrées au plus profond de l'architecture matérielle. Exploitant des mécanismes d'optimisation comme l'exécution spéculative, ces attaques permettent d'accéder à des données sensibles telles que des mots de passe ou des clés cryptographiques. Elles ont mis en lumière les limites des approches classiques de vérification, souvent focalisées exclusivement sur les couches logicielles, et peu sensibles aux subtilités des architectures matérielles sous-jacentes.

Ces constats soulignent la nécessité de **repenser les méthodes d'analyse et de vérification des systèmes embarqués**, en intégrant de manière **explicite les interactions entre le logiciel et le matériel**. Une compréhension fine de ces interactions devient indispensable pour anticiper, détecter et prévenir des vulnérabilités complexes avant qu'elles ne puissent être exploitées.

C'est dans cette perspective que s'inscrit notre travail, qui vise à améliorer les approches existantes ou à proposer une nouvelle méthode **prenant en compte de façon explicite les liens étroits entre logiciel et matériel**, afin de renforcer la sécurité et la sûreté des systèmes embarqués dès leur conception.

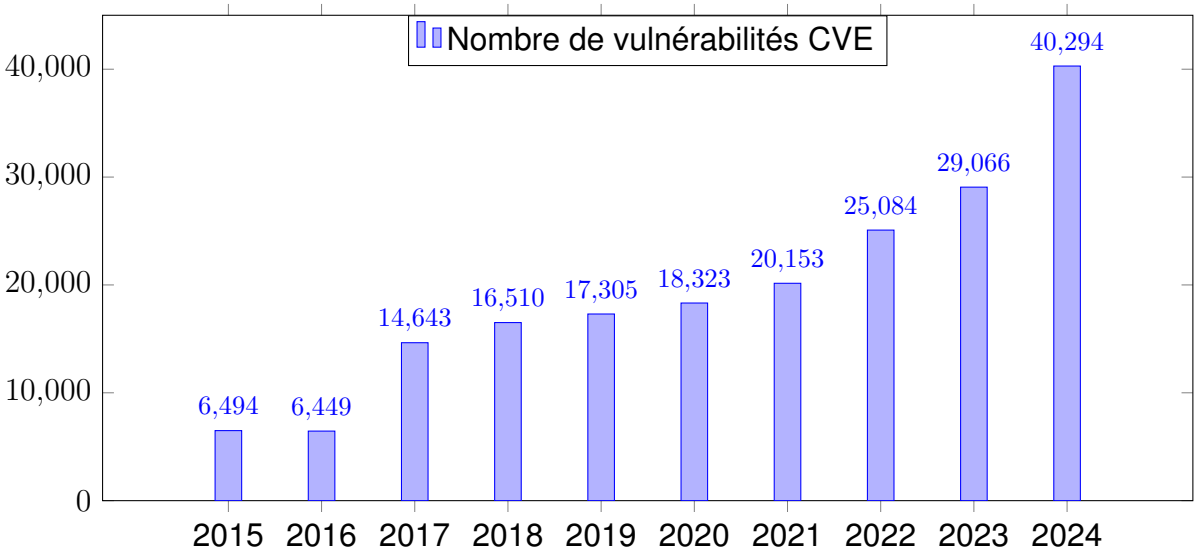

Figure 1.1: évolution du nombre de vulnérabilités CVE entre 2015 et 2024[1].

1.3 Contexte et positionnement

La **sûreté** et la **sécurité** partagent un objectif commun : prévenir les situations dangereuses et en limiter les conséquences. Toutefois, elles se distinguent par la nature des événements qu'elles cherchent à éviter et par les causes sous-jacentes qu'elles adressent [13]. Alors que la sûreté vise à empêcher les erreurs involontaires ou les défaillances accidentelles, la sécurité se concentre sur la prévention des attaques malveillantes et des comportements intentionnellement nuisibles [75, 13].

Bien que traditionnellement traitées séparément, la convergence de ces deux dimensions dans le contexte des logiciels embarqués soulève des problématiques d'une complexité particulière. En effet, certaines erreurs logicielles ou bugs initialement perçus comme des atteintes à la sûreté, peuvent également ouvrir des brèches exploitables sur le plan de la sécurité. C'est précisément sur cette zone d'intersection, où les deux notions se rejoignent, que se concentre notre travail.

Dans les systèmes embarqués, la complexité ne réside pas uniquement dans le code source, mais aussi dans les spécificités de l'environnement d'exécution, notamment les interactions étroites entre le logiciel et le matériel. Ce couplage fort rend les analyses classiques insuffisantes : un défaut ne découle pas nécessairement d'une erreur dans le code lui-même, mais peut émerger de comportements subtils induits par l'architecture matérielle ou les conditions d'exécution. Ainsi, la détection et la prévention des vulnérabilités exigent des approches d'analyse capables de prendre en compte à la fois les aspects logiciels et les contraintes matérielles.

Notre recherche s'inscrit dans cette perspective. Elle vise à explorer des **approches formelles intégrant explicitement les caractéristiques matérielles et environnementales dans les processus de vérification et d'analyse des programmes embarqués**. En modélisant rigoureusement le comportement du système dans un environnement donné, notre objectif est de garantir – à l'aide de preuves mathématiques – le respect de propriétés critiques de sûreté et de sécurité. Ce positionnement ouvre la voie à de nouvelles perspectives en analyse statique, en offrant une compréhension approfondie des interactions complexes qui, autrement, pourraient échapper aux méthodes traditionnelles.

[1] Source:https://www.cvedetails.com/browse-by-date.php

1.4 Objectifs et contributions

La sûreté et la sécurité des systèmes embarqués reposent sur une combinaison délicate de facteurs étroitement interconnectés : la qualité du code source, les interactions avec l'environnement matériel, ainsi que les transformations introduites au cours du processus de compilation. Ces éléments, lorsqu'ils ne sont pas rigoureusement maîtrisés, peuvent introduire des vulnérabilités critiques, dans des contextes où la moindre erreur peut avoir des conséquences graves.

Face à ces enjeux, **l'objectif central de cette thèse est d'identifier des vulnérabilités de sécurité qui émergent de failles de sûreté**, en vérifiant si un logiciel embarqué respecte un ensemble rigoureux de propriétés de sûreté et de sécurité, tout en tenant compte de son environnement d'exécution. Si les problématiques de sûreté restent pleinement intégrées à notre démarche, nous nous focalisons particulièrement sur celles susceptibles d'avoir un impact direct sur la sécurité.

Contributions principales. Les apports de cette thèse s'articulent autour de quatre axes majeurs :

1. **Développement d'une approche formelle intégrant conjointement le logiciel et l'impact du matériel sur son exécution.**
   Nous proposons une méthode d'analyse et de vérification basée sur les solveurs SMT (*Satisfiability Modulo Theories*), qui ramène la détection de vulnérabilités à un problème de satisfiabilité. Cette approche se distingue par sa capacité à révéler des vulnérabilités souvent ignorées par les outils traditionnels, en intégrant de manière explicite les interactions entre le logiciel et son environnement matériel.
2. **Formalisation des effets de l'environnement d'exécution.**
   Nous introduisons une formalisation précise de propriétés vérifiables qui capturent l'impact du matériel sur le comportement logiciel. Cette formalisation permet de détecter les divergences entre le comportement théorique d'un programme et son exécution réelle, en tenant compte des contraintes spécifiques à l'architecture cible. À titre d'exemple, nous nous intéressons notamment aux dépassements d'entiers, fréquents dans les architectures embarquées.
3. **Construction d'une base de connaissances architecturales.**
   Nous proposons la création d'une base de données structurée regroupant des modèles formels et des formules caractérisant les environnements d'exécution. Cette base intègre progressivement de nouveaux patterns associés aux spécifications matérielles et aux vulnérabilités connues. Conçue pour être extensible et

mutualisable, elle facilite l'adoption de notre approche dans d'autres contextes ou par d'autres chercheurs.

4. **Implémentation de l'outil AETHER — une chaîne d'analyse automatisée.** Nous avons implémenté l'outil AETHER (*Architecture-aware Execution and THEoretical Reasoning*), qui matérialise concrètement l'ensemble de l'approche, allant de l'analyse statique à la génération de code SMT. Cette chaîne inclut l'extraction des formules depuis la base de connaissances, la construction des conditions de chemin (path conditions), et la formulation du problème sous forme de contraintes logiques. Elle permet de produire des résultats précis, indiquant la présence ou l'absence de violations des propriétés de sûreté ou de sécurité.

1.5 Organisation de la thèse

Le **chapitre 2** explore les approches existantes pour l'analyse et la vérification des programmes. Il examine en détail les techniques actuelles, leurs avantages respectifs ainsi que leurs limites. Cette analyse permet de mieux comprendre les lacunes dans l'état de l'art et justifie l'approche proposée dans cette thèse.

Le **chapitre 3** introduit la partie théorique de notre approche. Il décrit les concepts fondamentaux, les outils mathématiques et les modèles utilisés pour concevoir notre approche formelle. Ce chapitre présente également l'algorithme qui automatise les différentes étapes de l'approche : construction de la condition de chemin, intégration des contraintes architecturales, et génération du code SMT.

Dans le **chapitre 4**, nous présentons la base de connaissances développée comme élément central de notre approche. Nous expliquons son rôle, sa construction et son exploitation pratique. Cette base regroupe les spécifications des architectures matérielles, des environnements d'exécution et des modèles d'assertions nécessaires pour détecter certaines classes de vulnérabilités. Elle est conçue pour être partageable et extensible, facilitant ainsi son adoption par la communauté scientifique.

Le **chapitre 5** présente AETHER, l'outil qui matérialise concrètement l'ensemble de l'approche. Reposant sur l'infrastructure Clang/LLVM (Low Level Virtual Machine) pour la construction de l'AST (*Abstract Syntax Tree*) et l'analyse statique, AETHER orchestre toutes les étapes du pipeline en exploitant la base de connaissances jusqu'à la génération du code SMT et l'interrogation du SMT solveur. Ce chapitre inclut également une

étude de cas concrète — une vulnérabilité d'escalade de privilèges par débordement arithmétique — illustrant la faisabilité et la pertinence de la solution proposée.

2

État de l'art : Analyse et Vérification des Programmes

Sommaire

2.1 Introduction

Dans ce chapitre, nous présentons les différentes approches et techniques de **vérification de programmes**, qu'elles soient **statiques** ou **dynamiques**, et montrons comment elles participent à la **détection de bugs** et à l'identification de **violations de bonnes pratiques de sécurité** au cours du développement.

En pratique, les développeurs n'ont pas toujours une vision complète de l'impact que peut avoir **l'environnement d'exécution** sur le comportement réel d'un programme. Des choix liés à la **compilation**, à **l'architecture matérielle** ou au **système d'exploitation** peuvent entraîner l'apparition de bugs subtils ou de comportements inattendus, parfois difficiles à diagnostiquer [2, 93]. Ce manque de recul sur ces facteurs explique la persistance de certaines vulnérabilités malgré l'usage d'outils classiques de test ou d'analyse.

Il est donc essentiel de mettre en évidence les éléments qui influencent ce comportement. Nous commencerons par examiner le **processus de compilation**, qui marque la transition du code source vers une forme exécutable. Les autres facteurs liés à l'environnement d'exécution seront abordés dans les chapitres ultérieurs.

Les sections suivantes introduisent ensuite les **principales méthodes d'analyse de programmes**, puis les différentes **approches de vérification**, qu'elles reposent sur des techniques formelles ou sur des tests. Enfin, nous présenterons un aperçu des **outils phares** utilisés par la communauté, en insistant sur ceux qui ont inspiré notre propre démarche et éclairent les choix méthodologiques développés dans la suite du manuscrit.

Ce chapitre presente quelques **défis actuels de la vérification de programmes**, en particulier ceux liés à l'intégration des aspects matériels et des environnements d'exécution. Ces constats permettent de mieux comprendre les limites des approches existantes et de situer notre contribution dans un contexte à la fois pratique et scientifique.

2.2 Compilation des programmes : de l'abstraction à l'exécution

Analyser des programmes de manière rigoureuse ne se limite pas à l'examen du code source lui-même. Il est également nécessaire de prendre en compte les éléments avec lesquels il interagit ainsi que le fonctionnement global de l'environnement d'exécution [2].

Le compilateur, dans sa description simplifiée, est un programme qui prend en entrée un code source et produit un programme exécutable sous forme binaire [2]. Cependant, le réduire à une simple **boite noire** Figure 2.1 masque la complexité de son fonctionnement. En réalité, de nombreuses étapes interviennentt pour transformer le code écrit par le développeur en un ensemble d'instructions compréhensibles par la machine [2].

Dans cette section, nous explorerons, étape par étape, les mécanismes internes du compilateur, afin de dévoiler comment il effectue cette conversion essentielle.
Cette analyse nous offre un recul précieux pour anticiper et identifier les problèmes, en démontrant que leurs origines ne se trouvent pas exclusivement dans le code source, mais aussi dans l'ensemble des interactions avec le système d'exploitation, le matériel et les autres composants de l'environnement d'exécution.

Figure 2.1: Illustration d'un compilateur en boite noire

2.2.1 Structure générale du compilateur

Structure du compilateur. Jusqu'à présent, nous avons considéré le compilateur comme une boite noire capable de transformer un programme source en un programme cible sémantiquement équivalent. En réalité, cette boite est composée de plusieurs modules, chacun jouant un rôle spécifique dans la chaîne de compilation.
Dans la littérature, deux grandes approches coexistent pour décrire cette structure. La première distingue deux composants principaux illustrée par la figure 2.2 : le **frontend** et le **backend** du compilateur [2]. Le **frontend** est chargé de l'analyse du code source (analyse lexicale, syntaxique, sémantique), tandis que le **backend** s'occupe de la génération du code machine et des optimisations spécifiques à l'architecture cible.
La seconde approche propose une décomposition plus fine, en introduisant un composant intermédiaire illustrée par la figure 2.3 : le **middle-end** [4]. Dans ce modèle, le *frontend* se limite à l'analyse et à la production d'une représentation intermédiaire,

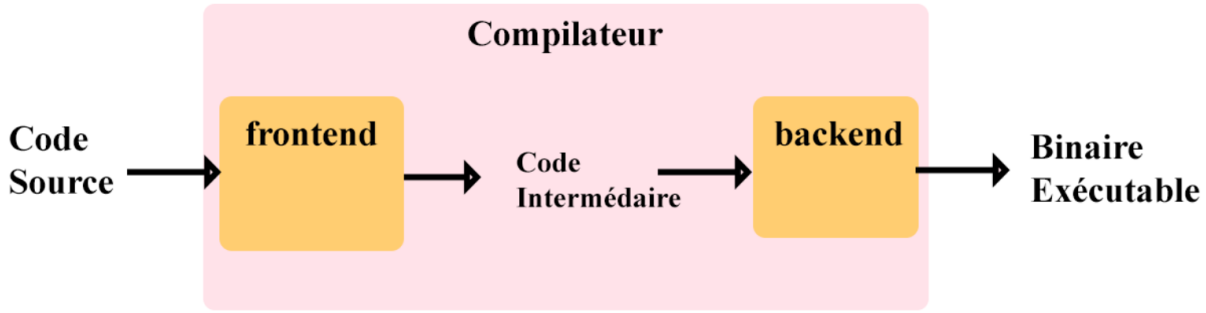

Figure 2.2: Illustration d'un compilateur avec deux composants internes

que le **middle-end** prend en charge pour effectuer des optimisations indépendantes de la machine cible. Ce n'est qu'ensuite que le **backend** intervient pour adapter le code aux spécificités de l'architecture.

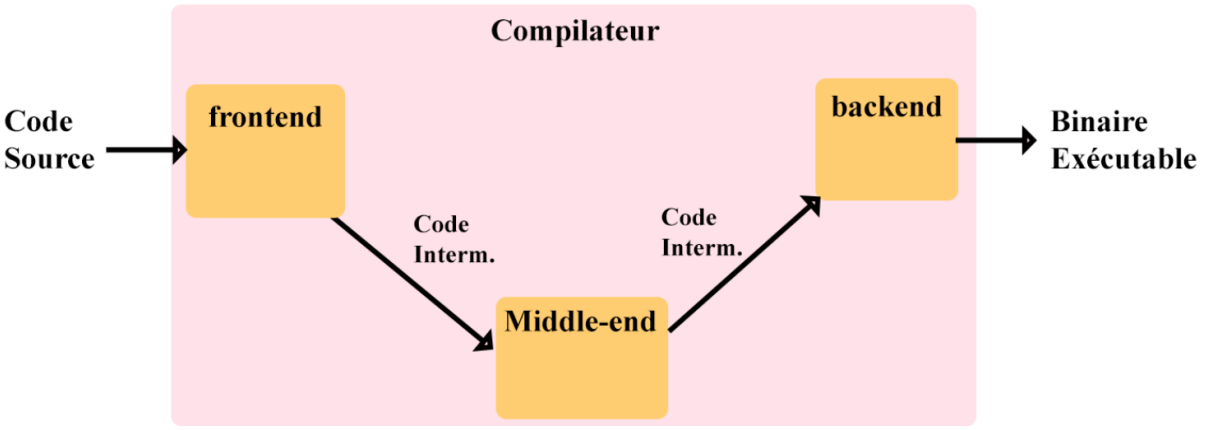

Figure 2.3: Illustration d'un compilateur avec trois composants internes

Ces deux modèles ne s'opposent pas mais reflètent des niveaux de granularité différents. Le second modèle peut être vu comme une subdivision du *frontend* en deux parties: le *frontend* proprement dit et le middle-end. Cette distinction devient particulièrement utile lorsqu'on s'intéresse aux optimisations [2], qui peuvent être soit génériques, soit spécifiques au matériel.

Il est courant de regrouper les phases d'analyse (lexicale, syntaxique, sémantique et génération de code intermédiaire) dans un *frontend* [2], et les phases d'optimisation et de génération de code dans un *backend*. Des outils comme Lex (ou Flex) et Yacc facilitent la mise en œuvre de ces phases en automatisant la génération d'analyseurs lexicaux et syntaxiques [2].
Tout au long de cette sous-section, nous allons illustrer les différentes étapes du *frontend* de la compilation sur un simple programme 2.1 qui demande à l'utilisateur

de saisir un entier, le multiplie par deux, puis affiche le résultat. Son objectif est simple et sa structure semble correcte. Pourtant, cette simplicité apparente masque plusieurs points d'attention, tant du point de vue de la compilation que des bonnes pratiques de programmation.

Listing 2.1: Exemple qui calcule le double d'un nombre donné par un utilisateur

```
1   #include <stdio.h>
2   int main() {
3       int y;
4       int x = 2;
5       printf("Enter one integer: ");
6       scanf("%d ", &y);
7       int result = x * y;
8       printf("le double de %d est: %d\n", y, result);
9       return 0;
10  }
```

2.2.2 Frontend : analyse du programme source

Analyse lexicale. La première passe du compilateur est l'analyse lexicale. Elle consiste à parcourir le flux de caractères du code source pour identifier des unités lexicales (appelées **tokens**) : mots-clés, identificateurs, constantes, opérateurs, symboles de ponctuation, etc.
Maintenant si on considere l'instruction `int result = x * y;` se trouvant à la ligne 7 du Listing 2.1, sa decomposition est illustré par la figure 2.4:

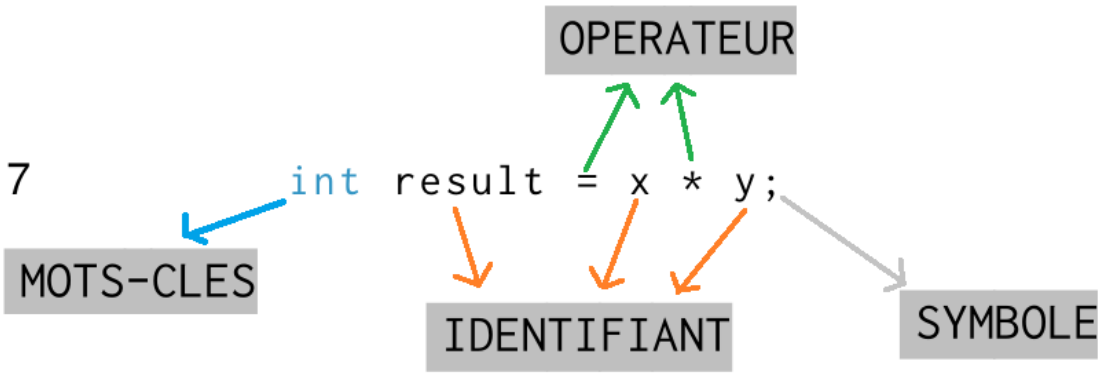

Figure 2.4: Identification des unités lexicales (tokens) du code source.

En effet, elle est convertie en : `int` → `MOTS-CLES` ; `result` → `IDENTIFIANT` ; `=` → `OPERATEUR` ; `x` → `IDENTIFIANT` `*` → `OPERATEUR` ; `y` → `IDENTIFIANT` `;` → `SYMBOLE`.
Cette phase est également responsable de la suppression des espaces et des commentaires, et de la détection d'erreurs comme : caractères invalides (@, #, etc.), chaînes de caractères mal fermées ("Hello) et les identifiants incorrects (2variable).

Analyse syntaxique *parseur*. À cette étape, le compilateur vérifie que les jetons obtenus précédemment sont agencés selon les règles grammaticales du langage. Il construit pour cela un arbre syntaxique abstrait noté AST qui reflète la structure hiérarchique du programme.
L'AST simplifié de notre programme pourrait ressembler à ceci :

```
main()
|— Declaration: int y
⋮
|— Declaration: int result = x * y
|    |__ Init value:
|        |__ BinaryOp: *
|            |— Left: x
|            |__ Right: y
|__ Return(0)
```

Lors de cette phase, le compilateur détecte des erreurs de syntaxe telles que l'oubli de points-virgules, des parenthèses ou accolades manquantes, ou encore des expressions mal formées. à partir du flux de jetons, l'analyseur syntaxique construit une structure arborescente, l'arbre syntaxique, qui représente la structure grammaticale du programme. Cela permet de vérifier que le code respecte la grammaire du langage.

Analyse sémantique : valider le sens du programme. Après les étapes lexicale et syntaxique, l'analyse sémantique vérifie la **cohérence logique** du programme, au-delà de sa simple structure grammaticale. Elle s'appuie principalement sur : **l'arbre syntaxique**, qui fournit une représentation structurée du code, et **la table des symboles**, qui centralise les informations sur les identificateurs (types, portées, déclarations, etc.) [2]. Le contrôle de type et la gestion des portées en constituent les aspects essentiels, permettant de détecter des erreurs telles que **l'utilisation d'identificateurs non déclarés,**

des opérations sur des types incompatibles ou des appels de fonctions avec un nombre ou un type d'arguments incorrect.
Cette phase prépare également la génération de code intermédiaire, notamment via des annotations sémantiques intégrées à l'arbre syntaxique. Toutefois, elle se limite aux règles du langage et n'assure pas la correction fonctionnelle : **un programme peut être sémantiquement valide tout en contenant des erreurs logiques.**
D'où l'intérêt d'une **vérification sémantique renforcée**, capable d'anticiper des anomalies liées à la gestion des types ou aux conversions implicites. Cette vigilance précoce contribue à réduire les bugs coûteux et à améliorer la fiabilité logicielle dès la compilation.

Ainsi, la phase *frontend* s'achève par la construction d'une **représentation intermédiaire** IR (Intermediate Representation) du programme source. Cette IR, qu'elle prenne la forme **d'arbres syntaxiques abstraits** ou de **code à trois adresses** (*three-address code*), constitué d'instructions de la forme `x = y Op z` [2], constitue le pont entre l'analyse du code et les étapes ultérieures. Elle offre une structure interne claire et **indépendante de la machine cible**, facilitant ainsi les optimisations au *middle end* ainsi que la génération du code au *backend*.

2.2.3 Middle-end & Backend : optimisation & génération de l'exécutable

L'architecture moderne d'un compilateur divise traditionnellement la synthèse en deux grandes étapes logiques : le *middle-end*, centré sur les optimisations indépendantes de la machine, et le *backend*, chargé de la traduction vers la machine cible et des optimisations dépendantes de l'architecture [2].

Le Middle-end : optimisations indépendantes de la machine. Le *middle-end* correspond à l'étape où différentes passes d'optimisation s'appliquent sur la représentation intermédiaire du programme. L'objectif principal est d'améliorer la structure et les performances du code, tout en restant indépendant de la cible matérielle [2]. Parmi les transformations courantes figurent la propagation de constantes (*constant folding*), l'inlining agressif, le dépliage de boucles (*loop unrolling*), l'élimination de code mort DCE (Dead Code Elimination), ou encore la transformation en forme SSA (*Static Single Assignment*) [2].
Le degré et la nature des optimisations effectuées à ce stade dépendent des options de compilation choisies, telles que `-O2`, `-O3` ou `-Ofast`, qui activent un ensemble de passes plus ou moins agressives.

Enfin, l'un des avantages majeurs de la transformation SSA est qu'elle facilite l'application de méthodes formelles pour raisonner sur le comportement du programme. C'est pourquoi nous lui consacrons une partie spécifique dans cette section.

Transformation SSA. La transformation SSA est une étape clé du *middle-end* d'un compilateur. Elle garantit que chaque **variable du programme** (c'est-à-dire un emplacement mémoire nommé pouvant être réaffecté au cours de l'exécution) est définie une seule fois en remplaçant chacune de ses réaffectations par une **variable SSA**. Une variable SSA est représentée par un **identifiant unique** (ou nom versionné), qui désigne une unique définition et, par conséquent, une valeur immuable [30]. Ainsi, une variable du programme x réaffectée plusieurs fois est transformée en une succession de variables SSA, représentées par les identifiants x_1, x_2, etc., chacun correspondant à une définition unique [62].

Listing 2.2: Code source classique

```
1 x = a + 3;
2 x = x + 1;
```

Listing 2.3: Représentation SSA

```
1 x1 = a + 3;
2 x2 = x1 + 1;
```

L'immuabilité des identifiants SSA constitue un atout majeur, non seulement pour les optimisations classiques des compilateurs (propagation de constantes, élimination de code mort, etc.) [62], mais également pour les méthodes formelles, qui s'appuient sur des représentations dépourvues de **effets de bord** (*side effects*) afin de raisonner logiquement sur les programmes [30]. En supprimant les réaffectations et les alias implicites, la représentation SSA rend explicites les dépendances de données, ce qui facilite la construction de formules logiques ou de modèles formels destinés à des outils tels que les solveurs SMT [62].
La SSA intègre également le flot de contrôle au moyen des **fonctions** Φ (ou nœuds `Phi`), qui fusionnent les différentes définitions d'une même variable du programme lorsque plusieurs chemins d'exécution convergent. Par exemple :
La fonction Φ associe un nouvel identifiant SSA à la variable du programme en sélectionnant la définition (x_1 ou x_2) correspondant au chemin d'exécution effectivement emprunté. Elle préserve ainsi l'unicité des définitions tout en représentant correctement les convergences du flot de contrôle.

Listing 2.4: Exemple de branchements conditionnels en C

```
1 if (cond) {
2     x = 1;
3 } else {
4     x = 0;
5 }
6 z = x * 3;
```

Listing 2.5: Représentation SSA des branchements conditionnels

```
1 if (cond) {
2     x1 = 1;
3 } else {
4     x2 = 0;
5 }
6 x3 = Phi(x1, x2);
7 z1 = x3 * 3;
```

En résumé, la transformation SSA fournit une représentation intermédiaire dans laquelle chaque variable du programme est remplacée par une succession d'identifiants SSA, chacun associé à une unique définition. Cette représentation rend explicites les dépendances de données, facilite les optimisations du compilateur et constitue une base particulièrement adaptée à la formalisation mathématique nécessaire aux méthodes formelles ainsi qu'à la génération de contraintes logiques.

Backend : Génération de code machine. Le *backend* est la phase finale de la compilation [2]. Il prend en entrée une représentation intermédiaire optimisée et la traduit en code cible (assembleur ou machine) adapté à l'architecture visée. Contrairement au *middle-end*, cette étape est **fortement dépendante de la machine**, car elle doit tirer parti de ses spécificités telles que **le jeu d'instructions**, **les registres disponibles** ou **les modes d'adressage**.

Le générateur de code repose principalement sur trois tâches cruciales :

- **La sélection des instructions** (*Instruction Selection*), qui choisit les séquences d'instructions machine appropriées pour implémenter chaque instruction IR.
- **L'allocation des registres** (*Register Allocation*), qui gère efficacement un nombre limité de registres pour contenir les valeurs et éviter des accès mémoire coûteux.
- **L'ordonnancement des instructions** (*Instruction Scheduling*), qui organise leur séquence d'exécution pour exploiter au mieux le parallélisme matériel ILP (Instruction Level Parallelism) tout en respectant les dépendances de données.

Une fois ces étapes réalisées, le compilateur procède à l'émission du code cible, sous forme de code assembleur ou de **code machine exécutable**. Le backend joue ainsi un rôle déterminant : il assure la traduction finale du programme tout en optimisant les performances selon la machine cible. Bien que la génération optimale de code soit un problème indécidable en général, l'objectif est de produire des exécutables corrects, efficaces et adaptés aux architectures modernes [2].

En résumé, le processus de compilation traduit un programme source en un programme cible par une série d'étapes successives, permettant non seulement d'exploiter la richesse des langages de haut niveau, mais aussi d'optimiser le code pour l'exécution sur une architecture matérielle donnée.

2.3 Méthodes d'analyse de programme

L'analyse de programme est une étape fondamentale dans le développement logiciel, en particulier pour les systèmes embarqués oú la sûreté et la sécurité jouent un rôle critique [15, 5]. Elle vise à détecter les erreurs, identifier les vulnérabilités et garantir que les programmes respectent les exigences fonctionnelles et non fonctionnelles [59, 44]. Grâce à une variété de méthodes et d'outils, l'analyse de programme permet d'améliorer la qualité, la robustesse et la fiabilité des logiciels [59, 36].

Avant d'entrer dans le détail des approches, il est utile de situer les méthodes d'analyse de programmes dans une taxonomie plus générale. La classification proposée par l'auteur [24] Figure 2.5, illustre la diversité des méthodes d'analyse selon deux dimensions fondamentales : la nature de l'analyse (statique ou dynamique) et le degré d'automatisation (manuel ou automatique). Cette perspective met en évidence la complémentarité des approches et sert de clé de lecture pour la suite de cette section. À partir de ce schéma, nous détaillerons dans les sections suivantes les principales méthodes, en commençant par l'analyse statique, puis l'analyse dynamique, avant d'aborder les approches hybrides [68, 54].

2.3.1 Analyse statique

L'analyse statique, est une méthode qui consiste à examiner le code source ou le code binaire d'un programme sans l'exécuter, afin de détecter des erreurs, des vulnérabilités ou des violations de spécifications dès les premières phases du développement [76]. Elle joue un rôle crucial dans la formalisation et l'assurance de propriétés critiques, notamment la sécurité et la sûreté.

L'analyse statique constitue une méthode clé pour détecter un large éventail de bugs dans les logiciels, en prenant en compte les propriétés de sécurité, de sûreté et celles définies par l'utilisateur. Ces méthodes d'analyse varient principalement en termes de précision (taux des fausses alarmes faible élevé), de capacité à s'adapter à de grands programmes et des théories sous-jacentes utilisées. Le domaine a connu

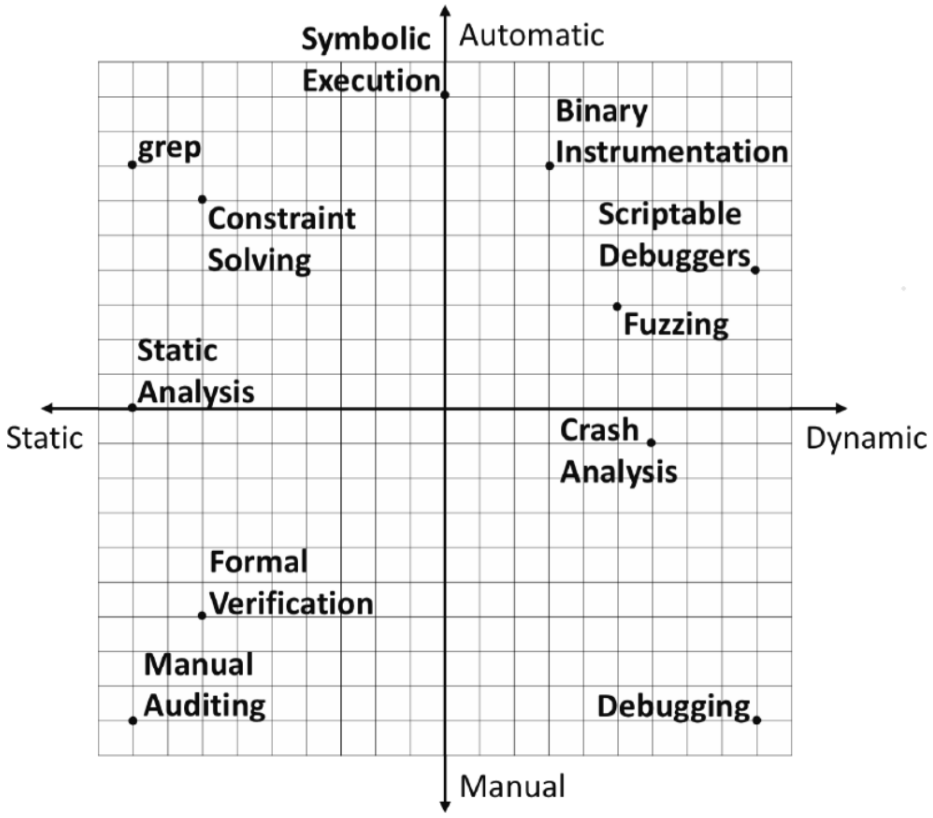

Figure 2.5: Méthodes d'analyse de programmes selon les métriques : statique vs dynamique, manuel vs automatique [24].

des avancées considérables, notamment avec l'introduction d'outils qui s'appuient sur des formules mathématiques et logiques pour raisonner sur le comportement des programmes sans les exécuter.

Analyse syntaxique par motifs (*Pattern matching*). L'analyse syntaxique par motifs vise à identifier, dans la structure du programme, des constructions susceptibles d'introduire des erreurs ou des vulnérabilités [44]. Elle commence généralement par la transformation du code source en un flux de jetons via un analyseur lexical, puis en un arbre syntaxique abstrait AST [54, 95]. Les motifs dangereux sont ensuite recherchés soit directement sur le flux de jetons [54], soit sur l'AST, en s'appuyant sur des bases de signatures prédéfinies, souvent alignées sur des référentiels tels que le CWE (*Common Weakness Enumeration*) [29, 53, 65].
Bien que rapide et simple à mettre en œuvre [44, 54], cette méthode présente des limites notables : elle ne tient pas compte du contexte sémantique du programme, ce qui entraîne un taux élevé de faux positifs [54, 44] et une fiabilité limitée des alertes

produites. Elle reste néanmoins pertinente dans les premières étapes d'analyse, pour une détection précoce de motifs connus[53].

Analyse de flux de contrôle. L'analyse de flux de contrôle a pour objectif de **modéliser les chemins d'exécution possibles d'un programme** ou d'une fonction particulière [89]. Elle repose sur la construction d'un graphe de flux de contrôle *Control Flow Graph* (CFG) [95, 44], une structure orientée dont les nœuds représentent les blocs de code (instructions séquentielles) et les arêtes, les transitions conditionnelles possibles entre ces blocs. Le CFG constitue une abstraction essentielle utilisée dans de nombreuses techniques d'analyse statique pour représenter le comportement potentiel d'un programme.
Les approches fondées sur le CFG sont fréquemment combinées avec d'autres formes d'analyse (*Dataflow*, *tainting*, *slicing*, etc.) [73] afin de vérifier des propriétés de sûreté ou de sécurité, de détecter des bugs, ou encore de signaler des violations de bonnes pratiques [89]. Ce couplage permet notamment de raffiner les diagnostics en tenant compte de la structure logique du programme.
Certaines techniques s'appuient spécifiquement sur le CFG pour renforcer la sécurité d'exécution, comme l'intégrité du flux de contrôle CFI (Control Flow Integrity) [94, 22], qui vise à garantir que le programme ne dévie pas de son graphe de contrôle légitime, en bloquant ainsi certaines classes d'attaques. Des outils comme *MOPS* [95] exploitent cette approche en modélisant le programme sous forme d'automates afin de vérifier des propriétés de sécurité temporelles, tandis que *Fortify* l'intègre dans son moteur pour détecter des séquences d'opérations potentiellement dangereuses [95].
Enfin, l'analyse de flux de contrôle reste une composante précieuse mais partielle, qui gagne en pertinence lorsqu'elle est intégrée dans un cadre d'analyse plus global [18, 49, 22].

Analyse de flux de données L'analyse de flux de données vise à déterminer, pour chaque point d'un programme, des informations sur les valeurs possibles des variables et les dépendances entre instructions, dans le but de **prédire statiquement le comportement dynamique du programme** [44]. Elle repose sur la construction d'un graphe de flux de contrôle (CFG), sur lequel un algorithme itératif est appliqué pour résoudre des équations de flux. Ces équations définissent, pour chaque nœud, des ensembles de faits, comme les variables vivantes ou les définitions atteignables, qui sont mis à jour jusqu'à convergence.

Cette analyse permet ainsi d'identifier des propriétés utiles comme les définitions atteignables, les expressions disponibles ou les variables vivantes. Elle peut être menée de manière intra-procédurale ou inter-procédurale, et est souvent combinée à d'autres techniques telles que l'analyse de flot de contamination (*taint analysis*), qui suit la propagation de données sensibles ou non fiables [31, 91], ou encore l'analyse de points-à (*points-to analysis*) pour la gestion fine des pointeurs [22].
Des outils tels que *CodeSonar* ou l'analyseur statique de *Clang* [8] exploitent ces techniques pour détecter des vulnérabilités liées à la propagation de données non contrôlées. Toutefois, en raison de son caractère conservatif, cette approche peut inclure des chemins inaccessibles à l'exécution, ce qui peut entraîner des faux positifs. à l'inverse, des hypothèses incorrectes sur le comportement des pointeurs peuvent conduire à des faux négatifs.

2.3.2 Analyse dynamique

L'analyse dynamique consiste à exécuter un programme dans un environnement réel ou simulé afin d'observer son comportement effectif [43, 84]. Contrairement à l'analyse statique, qui examine le code sans l'exécuter, elle se focalise sur ce que le programme fait réellement plutôt que sur ce qu'il pourrait faire [43]. Cette approche permet de détecter des bugs et vulnérabilités qui ne se manifestent que dans certaines conditions d'exécution [84], de **déjouer des techniques d'obfuscation** et d'analyser des binaires sans accès au code source [3].

L'analyse dynamique **repose sur la collecte d'informations issues de traces**, d'événements ou d'états observés au cours de l'exécution : accès mémoire, trafic réseau, appels système ou encore dumps du tas [84, 43]. Ces données fournissent des éléments factuels permettant de confirmer ou d'infirmer des hypothèses sur le comportement attendu, tout en réduisant le risque de faux positifs [50, 80].

Toutefois, cette approche **ne couvre que les chemins effectivement exécutés**, ce qui limite sa capacité à explorer l'intégralité de l'espace d'exécution [84, 3]. Pour atténuer cette limite et enrichir l'observation, plusieurs techniques spécifiques sont mises en œuvre, comme la surveillance et instrumentation [43], l'analyse dynamique de flot de données *(dynamic taint analysis)* [84] et le fuzzing [67, 68].

Surveillance et instrumentation La surveillance dynamique consiste à intégrer dans le programme ou son environnement des mécanismes destinés à collecter des données d'exécution [84, 43]. L'instrumentation peut être réalisée au niveau du **code source, du**

bytecode ou du binaire [43], et permet de suivre de manière fine le comportement du programme pendant son fonctionnement. Les informations collectées peuvent couvrir différents aspects :

- **Suivi d'événements fonctionnels et structurels :** L'enregistrement des appels de fonctions, des appels de systèmes [50], ou des interactions avec des API critiques (par exemple, des API non sécurisées pour les *threads* [63]), des accès mémoire [68], des synchronisations de *threads* afin de diagnostiquer des problèmes de concurrence ou de comportement anormal [50, 43].
- **Détection d'erreurs mémoire :** L'objectif est l'identification des accès illégaux à la mémoire (spatiaux et temporels, incluant les débordements et sous-débordements de tampon) [68], des fuites de mémoire, ou des conditions de concurrence [63]. Ces outils ont la capacité de signaler des problèmes précis tels que les accès hors limites *(out-of-bounds access)*, les utilisations après libération *(use-after-free)* [67], les utilisations de mémoire non initialisée, les doubles libérations (double-free), ou les écritures concurrentes non protégées. Des outils comme *AddressSanitizer* **(ASan)**, *MemorySanitizer* **(MSan)**, et *ThreadSanitizer* **(TSan)** [68] sont des exemples qui détectent ces types d'erreurs. Cependant, ils impliquent souvent une surcharge significative en raison de l'instrumentation et de l'analyse en temps réel [68, 67].
- **Profilage des performances et des ressources** : mesure de l'utilisation du processeur, de la mémoire, des entrées/sorties et d'autres ressources matérielles [43], afin d'identifier les goulots d'étranglement et d'optimiser les sections critiques [84]. Des outils comme *perf* fournissent des métriques quantitatives précieuses. Bien que cette approche ne vise pas directement la correction fonctionnelle, elle contribue à améliorer l'efficacité et la scalabilité du système.

Analyse dynamique de flux de données Dynamic Taint Analysis (DTA).
La DTA est une technique qui suit le flux d'exécution d'un programme pour **identifier la propagation de données potentiellement non fiables** ou malveillantes (*tainted flows*) [84].
Elle consiste à marquer les données à leur origine (*sources*) — par exemple des entrées utilisateur, des données réseau ou des informations sensibles — et à observer leur propagation jusqu'à des points critiques (*sinks*) tels que des appels système ou des écritures mémoire [84]. À chaque instruction, les marques (*taints*) sont propagées selon des règles prédéfinies, généralement à l'aide de mécanismes d'instrumentation [88]. Cette méthode permet de détecter des vulnérabilités liées à des flux de données mal

contrôlées et se combine efficacement avec d'autres techniques. Intégrée à l'analyse statique, elle permet de concentrer l'instrumentation sur des chemins pertinents. **Associée au *fuzzing***, elle sert à cibler les portions d'entrée influençant des opérations sensibles, améliorant ainsi l'exploration de l'espace d'exécution. Elle présente toutefois certaines limites : la définition précise des sources et des puits dépend fortement de l'application [84], et l'analyse peut être coûteuse en ressources.

2.3.3 Exécution symbolique : une approche hybride

L'exécution symbolique est une technique d'analyse de programme fondamentale conçue pour explorer systématiquement les chemins d'exécution potentiels d'un programme [9, 32]. Au lieu d'utiliser des valeurs d'entrée concrètes, l'exécution symbolique **remplace les variables d'entrée par des variables symboliques**, comme α_i, qui représentent des ensembles de valeurs plutôt que des instances uniques [9].
Le moteur d'exécution symbolique maintient un **état symbolique**, incluant une structure qui mappe les variables du programme à des expressions symboliques, et une **formule booléenne de premier ordre appelée contrainte de chemin** (π) pour chaque chemin d'exécution exploré [9].
Lors de l'exploration du programme, notamment aux points de branchement conditionnels (par exemple, des instructions `if` ou `while`), l'exécution symbolique **construit et met à jour ces contraintes de chemin** en y ajoutant les conditions (ou leur négation) des branches prises, et en **dupliquant l'exécution** pour explorer toutes les alternatives faisables [14].
Pour déterminer la faisabilité d'un chemin ou **générer des valeurs concrètes** qui satisfont les contraintes accumulées, un **solveur SMT**, tel que Z3 ou CVC5, est utilisé [32, 9]. Ce solveur permet de trouver des affectations concrètes aux variables symboliques qui rendent la contrainte de chemin satisfaisable, produisant ainsi de nouvelles entrées de test.

L'exemple ci-dessous illustre ce mécanisme : la colonne de gauche montre le code, et la colonne de droite sa représentation symbolique.

Exemple de code analysé

```
1  int foo(int x, int y) {
2      if (x > 0) {
3          if (y > 0)
4              return 1;
5              //Chemin 1
6          else
7              return 2;
8              //Chemin 2
9      } else {
10         return 3;
11         //Chemin 3
12     }
13 }
```

Exploration symbolique :

Input : x = α, y = β

- Chemin 1 : $\pi_1 = (\alpha > 0 \wedge \beta > 0)$
- Chemin 2 : $\pi_2 = (\alpha > 0 \wedge \beta \leq 0)$
- Chemin 3 : $\pi_3 = (\alpha \leq 0)$

Exemples de solutions fournies par le solveur SMT :

- $(\alpha = 1, \beta = 1)$ pour le chemin 1
- $(\alpha = 1, \beta = -1)$ pour le chemin 2
- $(\alpha = -1, \beta = 0)$ pour le chemin 3

Cet exemple minimaliste illustre le principe : les entrées x et y sont remplacées par les variables symboliques α et β. Chaque branche du programme correspond alors à une contrainte de chemin, dont la faisabilité est validée par le solveur SMT.

En combinant cette représentation symbolique avec la génération d'entrées par SMT solver, l'exécution symbolique permet une **exploration exhaustive et systématique des chemins d'exécution**. Toutefois, la gestion de l'explosion combinatoire des chemins et **l'interaction avec l'environnement d'exécution** restent des défis majeurs [9].

L'exécution symbolique se décline en plusieurs variantes, chacune visant à optimiser son efficacité face aux défis inhérents à l'analyse de programme, notamment l'explosion des chemins d'exécution et la gestion d'environnements complexes.

L'exécution symbolique dynamique, également appelée *concolique* Dynamic Symbolic Execution (DSE), est une approche pratique qui combine l'exécution concrète et symbolique [9, 32]. Contrairement à l'exécution symbolique pure, qui tente d'explorer simultanément tous les chemins en dupliquant l'exécution à chaque branche conditionnelle [9, 20], l'exécution concolique limite son exploration à un chemin concret, tout en construisant en parallèle les contraintes symboliques associées [32].
Le principe est itératif : le programme est d'abord exécuté avec une entrée concrète, ce qui permet de collecter une contrainte de chemin correspondant au parcours suivi [9, 32]. Pour explorer d'autres chemins, l'analyse modifie une ou plusieurs conditions de branchement rencontrées, puis confie la recherche d'une entrée satisfaisante à un solveur SMT (tel que Z3, CVC5 ou Boolector). La nouvelle entrée générée force alors

le programme à emprunter un chemin inédit lors de la prochaine exécution.
Cette stratégie permet de limiter l'explosion combinatoire des chemins [83], problème majeur de l'exécution symbolique classique. Elle se révèle particulièrement utile pour :

- Générer automatiquement des jeux de tests ciblés et découvrir des vulnérabilités difficiles à atteindre [83, 74, 9].
- Gérer les environnements complexes (bibliothèques externes, appels système, code non analysable symboliquement) en concrétisant les valeurs concernées [20, 74, 9].
- Traiter plus efficacement des contraintes difficiles pour les SMT solvers (comme l'arithmétique non linéaire ou les fonctions transcendantales) [9].

2.4 Méthodes de vérification

Après avoir présenté les différentes approches d'analyse de programmes, il convient désormais d'aborder la question de leur vérification. Alors que l'analyse se limite à observer et décrire le comportement potentiel d'un programme, la vérification vise à établir si ce comportement est conforme à des spécifications définies, qu'elles soient fonctionnelles, de sûreté ou de sécurité [12].
Cette étape constitue un prolongement naturel de l'analyse, en passant de l'identification d'indices ou d'approximations à la démonstration de conformité. Elle est particulièrement cruciale dans les contextes critiques où une simple défaillance logicielle peut entraîner des conséquences graves [48]. Dans de tels cas, les tests traditionnels ne suffisent pas [90], car, comme le rappelait Dijkstra, *" les tests ne peuvent prouver l'absence d'erreurs "*.
Deux grandes approches dominent la vérification des programmes. Les méthodes formelles, fondées sur les mathématiques et la logique [72], sont particulièrement adaptées pour vérifier des propriétés spécifiques au niveau des programmes, en offrant une rigueur mathématique et des garanties plus fortes. Elles incluent notamment le *Model checking*, le theorem proving et l'usage de SMT solvers [71, 42]. Les méthodes de test, plus empiriques et largement utilisées en pratique [90, 48], consistent à exécuter le programme sur un sous-ensemble représentatif de cas, fournissant ainsi une validation pratique mais intrinsèquement non exhaustive.

2.4.1 Méthodes formelles

Les méthodes formelles regroupent un ensemble de techniques fondées sur les mathématiques et la logique, visant à développer et à raisonner rigoureusement sur des sys-

tèmes complexes ou critiques, qu'il s'agisse de logiciels, de matériels ou d'architectures hybrides combinant les deux [72]. Elles jouent un rôle central dans l'analyse de programmes en permettant la détection précoce de bugs, d'erreurs d'exécution et de violations de propriétés liées à la sécurité ou à la sûreté [11].
Contrairement aux approches de test classiques qui ne permettent d'explorer qu'un sous-ensemble des cas possibles, les méthodes formelles offrent une exploration systématique et une garantie de correction fondée sur des preuves mathématiques. Elles permettent ainsi de mettre en évidence des erreurs difficilement détectables autrement et d'apporter des garanties élevées sur la conformité des implémentations par rapport aux spécifications formelles.
La suite de cette section présente les principales techniques relevant des méthodes formelles utilisées dans l'analyse de programmes.

Fondements logiques et automatisation Il existe dans un premier niveau la **logique propositionnelle** ou logique d'ordre zéro, qui manipule des énoncés atomiques ou phrases déclaratives reliés par des opérateurs booléens tels que la conjonction ($\wedge$), la disjonction ($\vee$), la négation ($\neg$) ou l'implication ($\rightarrow$) [72, 71]. Ce systeme permet de modéliser des relations logiques simples, mais son expressivité reste limitée puisqu'il ne prend pas en compte les variables, fonctions, relations ou quantifications nécessaires pour traiter des problèmes complexes [72]. Par exemple, considérons un système de circulation : soit R : "feu rouge" et S : "voiture stoppée". La règle de sûreté *"si le feu est rouge, alors la voiture doit s'arrêter"* s'exprime simplement par l'implication $R \rightarrow S$.
Pour dépasser cette limite, on introduit la logique du premier ordre FOL (First-Order Logic), qui est une extension de la logique propositionnelle et est généralement plus expressive [72]. La FOL enrichit la logique propositionnelle avec des variables représentant des objets, des fonctions, des prédicats, ainsi que des quantificateurs universel ($\forall$) et existentiel ($\exists$). Ainsi, la règle précédente peut être affinée afin de prendre en compte les véhicules prioritaires (ambulances, police, pompiers). Notons $Stop(v)$: "le véhicule v s'arrête", $Exception(v)$: "le véhicule v est prioritaire", et $Rouge$: "le feu est rouge". La règle s'exprime alors comme suit :

$$\forall v, (Rouge \wedge \neg Exception(v) \rightarrow Stop(v)).$$

Cette formalisation illustre la capacité de la logique FOL, qui est la base logique de la majorité des systèmes de prouveurs de théorèmes [72], à raisonner sur des classes d'objets et à exprimer des conditions plus réalistes. Néanmoins, la vérification de la

satisfiabilité des FOL reste **semi-décidable** : c'est à dire il existe un algorithme qui s'arrête toujours et répond *True / False* si la formule est valide, mais rique de tourner indéfiniment si la formule n'est pas valide, ce qui en limite l'automatisation complète [72].

Les solveurs SAT et SMT Le problème de **satisfiabilité booléenne (SAT)** est une question fondamentale en informatique théorique et pratique [39, 42]. Il consiste à déterminer s'il existe une affectation de valeurs qui satisfait une formule booléenne donnée. Par exemple, la formule $(x_1 \wedge x_2) \vee \neg x_3$ est satisfaisable si $x_1 =$ vrai, $x_2 =$ vrai et $x_3 =$ faux. Une méthode simple comme le brute force, consiste à énumérer toutes les combinaisons possibles de valeurs (table de vérité), mais cette approche devient rapidement impraticable car le nombre de cas croît exponentiellement avec le nombre de variables (2^n). Reconnu comme le premier problème démontré **NP-complet**, SAT constitue un pilier de la complexité algorithmique [42].
Les **solveurs SAT** modernes vont bien au-delà de l'énumération naïve. Ils reposent sur des algorithmes complets basés sur la recherche par retour arrière (*backtracking*), dont l'exemple classique est l'algorithme DPLL (Davis–Putnam–Logemann–Loveland). Celui-ci procède par assignations successives de variables, puis revient en arrière lorsqu'une contradiction est détectée. L'efficacité de cette approche a été considérablement renforcée par l'introduction du Conflict-Driven Clause Learning (CDCL), qui permet au solveur d'apprendre des clauses de conflit pour éviter de revisiter les mêmes impasses et ainsi réduire drastiquement l'espace de recherche [81, 71].
Alors que le SAT se limite à la logique propositionnelle, les **SMT solvers** en généralisent le principe [71]. Ils s'attaquent à la satisfiabilité de formules exprimées en logique du premier ordre, où les variables et relations appartiennent à des théories mathématiques spécifiques (par exemple : l'arithmétique linéaire sur les entiers et les réels, les tableaux, les bit-vecteurs, ou encore les nombres à virgule flottante).
La majorité des SMT solvers, comme **Z3** [33] ou **CVC4**/**CVC5** [10], utilisent l'approche **DPLL(T)** [71]. Son principe est de déléguer la partie purement booléenne de la formule à un moteur SAT, tandis que des procédures spécialisées (*theory solvers*) vérifient localement la cohérence des contraintes issues des théories considérées. Cette collaboration permet de combiner l'efficacité des solveurs SAT avec la richesse des théories, faisant des SMT solvers des outils incontournables pour la vérification formelle [11].

Model checking et interprétation abstraite En complément de ces approches, le *Model checking* et l'interprétation abstraite jouent des rôles cruciaux dans vérification formelle. Le *Model checking* est une méthode qui vérifie si un système donné satisfait

une propriété désirée [71]. En effet, le système doit etre modélisé sous la forme d'un **automate à états finis**, où chaque état représente une configuration possible et chaque transition décrit une évolution du système.La rôle du *Model checking* consiste alors à déterminer si cet automate satisfait certaines propriétés formulées dans un **langage logique formel**, tel que la Logique Temporelle Linéaire (LTL) [11], si on souhaite vérifier la propriétée **tout au long d'un chemin d'execution,** ou la Logique Computationnelle des Arbres (CTL) [11], si on souhaite vérifier la propriétée **à partir d'un état** sur les differents chemins d'execution possibles.
À partir de cette modélisation, le model checking explore systématiquement l'espace des états et confronte chaque trajectoire d'exécution aux formules logiques spécifiées [11]. En cas de violation, l'outil fournit un **contre-exemple**, c'est-à-dire une trace d'exécution illustrant concrètement le défaut, ce qui en facilite l'analyse et la correction. Cette capacité à générer des contre-exemples constitue un atout majeur par rapport à d'autres techniques de vérification. Toutefois, le *Model checking* est confronté au problème classique de l'**explosion combinatoire** [71], le nombre d'états pouvant croître de manière exponentielle avec la taille du système [71].

L'**interprétation abstraite** est une technique d'analyse statique qui permet de raisonner sur les comportements possibles d'un programme en regroupant plusieurs exécutions concrètes dans un scénario abstrait [26]. Au lieu de manipuler les valeurs exactes des variables, elle utilise des représentations approximatives (par exemple, des intervalles) afin de capturer un ensemble de trajectoires d'exécution en une seule analyse [11].
Ces abstractions sont propagées dans le graphe de contrôle du programme **CFG** au moyen d'une fonction reliant valeurs concrètes et valeurs abstraites [11]. Par un processus itératif, l'analyse converge lorsque les valeurs abstraites cessent de changer, c'est-à-dire à l'atteinte d'un **point fixe** [26]. Une erreur signalée à ce stade correspond donc à un comportement potentiellement réalisable dans le programme concret.
L'intérêt de l'interprétation abstraite réside dans son compromis entre précision et faisabilité [26]. L'exploration exacte des exécutions est en effet souvent **intractable**, voire impossible, en raison de la taille potentiellement infinie de l'espace des états [11]. L'abstraction réduit cet espace à une taille finie et manipulable, tout en préservant les propriétés pertinentes pour l'analyse. Ainsi, il devient possible de détecter statiquement des erreurs telles que des **dépassements de bornes**, des **débordements arithmétiques** ou des **conditions inaccessibles**.

Enfin, l'alliance de la logique formelle, des solveurs automatiques, du *Model checking* et de l'interprétation abstraite forme le socle des approches modernes d'analyse de programmes, chaque méthode apportant des garanties spécifiques et complémentaires.

2.4.2 Méthodes des test

Le **test logiciel** (*software testing*) constitue une étape indispensable du cycle de vie du développement [48]. Il vise à vérifier que le comportement observé d'un programme correspond aux attentes et qu'il satisfait les exigences fonctionnelles et non fonctionnelles définies [48]. Contrairement aux méthodes formelles, le test repose principalement sur une démarche empirique, basée sur l'exécution effective du logiciel dans un ensemble de scénarios représentatifs [28].
L'objectif du test est d'*exécuter le programme dans l'intention de trouver des erreurs* [48], ce qui en fait un instrument essentiel de **l'assurance qualité** et du processus de **Vérification et Validation (V&V)** du logiciel [48]. La philosophie sous-jacente est simple : **un test qui échoue suffit à démontrer qu'un logiciel ne fonctionne pas correctement**. En revanche, aucun ensemble fini de tests ne peut garantir l'absence de défauts, car il est impossible de couvrir exhaustivement tous les scénarios d'exécution possibles d'un programme non trivial [28].
Cette limite s'explique par la nature même des défauts logiciels et par la complexité croissante des systèmes modernes. Les défauts logiciels **proviennent principalement d'erreurs de conception** qui demeurent latentes jusqu'à leur activation dans des conditions particulières [28]. De plus, l'explosion combinatoire des cas d'entrée rend la détection exhaustive quasi-impossible, même pour des programmes de taille et de complexité modérées. Ainsi, bien que le test constitue une méthode pragmatique et largement utilisée, il ne peut, à lui seul, fournir de garanties complètes sur la correction d'un logiciel [28].

Techniques de test Il existe plusieurs techniques de test, chacune pouvant être appliquée à différents niveaux au cours du **cycle de développement logiciel** [48]. Les principales techniques se déclinent comme suit :

- **Test unitaire (Unit Testing)** : Il se concentre sur la vérification isolée de chaque *module*[1] individuel (standardisé IEEE 1008-1987(R2009)) [28]. Réalisé en général par les développeurs eux-mêmes, il permet d'assurer la **conformité fonctionnelle** de chaque unité de base.
- **Test d'intégration (Integration Testing)** : Il vise à évaluer **l'interaction entre des modules** développés séparément [28]. Cette étape est particulièrement critique, car de nombreux défauts émergent lors de la **mise en relation de composants** initialement testés de manière indépendante.

[1] Dans le contexte du développement logiciel, un *module* désigne généralement une fonction ou une unité logique définie par le développeur au niveau du **code source**.

- **Test système (System Testing)** : Il consiste à valider le **comportement global du logiciel** par rapport aux **exigences fonctionnelles et non fonctionnelles** spécifiées [48, 28], en reproduisant des scénarios d'utilisation représentatifs.
- **Test d'acceptation (Acceptance Testing)** : Il représente la phase de **validation finale**, au cours de laquelle on vérifie la **conformité du produit** aux attentes de l'utilisateur avant son déploiement effectif [28].

Ces différents niveaux sont **complémentaires** et permettent d'assurer une couverture progressive allant des unités les plus élémentaires jusqu'au système complet.

D'autres parts, les approches de génération de cas de test se distinguent principalement par le degré de connaissance qu'elles requièrent de la structure interne du programme. Les techniques de ***boîte noire*** (*Black Box Testing*) s'appuient exclusivement sur les spécifications externes, sans tenir compte du code source [28, 48]. Elles consistent à fournir des entrées au système et à analyser les sorties produites afin de vérifier sa conformité aux exigences fonctionnelles [48]. Parmi les méthodes couramment utilisées figurent le *Fuzzing* (test aléatoire), le test par graphe de cause à effet, ainsi que l'analyse des valeurs limites (*Boundary Value Analysis*) [28].

À l'opposé, les techniques de ***boîte blanche*** (*White Box Testing*) exploitent la structure interne et la logique du programme [48]. Leur objectif est de vérifier non seulement ce que le programme fait, mais également comment il le fait [28], afin de détecter des erreurs subtiles susceptibles d'échapper aux tests purement fonctionnels. Ces approches mobilisent notamment des techniques telles que le *Control Flow Testing*, le *Basis Path Testing* ou encore le *Loop Testing* [48].

Entre ces deux extrêmes, les approches de ***boîte grise*** (*Grey Box Testing*) combinent une connaissance partielle de la structure interne avec des objectifs fonctionnels, dans le but d'accroître la pertinence et l'efficacité des tests lorsque l'accès au code est partiel ou limité [48].

Enfin, certaines pratiques transversales viennent renforcer l'efficacité de ces approches dans les cycles de développement modernes. Le ***test de régression*** permet de vérifier la stabilité du logiciel après des modifications [28]. De plus, des stratégies telles que le **Test-Driven Development (TDD)** et l'**Intégration Continue (CI)** positionnent le test au cœur même du processus de développement, en favorisant une détection rapide et continue des régressions et anomalies.

Pour conclure cette section, bien que les tests constituent un outil essentiel pour améliorer la qualité logicielle, leur portée reste limitée par leur caractère non exhaustif. Dans les contextes critiques, où des preuves sur le bon comportement du logiciel

sont nécessaires, les méthodes formelles s'imposent grâce à leur rigueur logique et mathématique. La combinaison de ces deux approches permet ainsi d'allier validation pratique et assurance de conformité, offrant une couverture complémentaire particulièrement efficace.

2.5 Outils d'analyse et de vérification

Cette section présente deux outils d'analyse et de vérification parmi les plus avancés et largement utilisés à l'état de l'art. Nous nous intéressons en particulier à **KLEE**, qui repose sur l'exécution symbolique pour explorer dynamiquement les chemins d'exécution, et à **Frama-C**, qui propose un cadre d'analyse statique modulaire basé sur des spécifications formelles.
L'objectif ici est de mettre en évidence les différences majeures entre ces deux outils et de situer, par la suite, notre approche par rapport à celles-ci.

2.5.1 KLEE : Moteur d'execution symbolique dynamique

KLEE est un moteur d'exécution symbolique dynamique, développé sur la base de la plate-forme de compilation LLVM, destiné à la vérification et au test automatique de programmes en C/C++. Son objectif est de générer automatiquement des cas de test et de révéler des bugs en explorant systématiquement les chemins d'exécution possibles d'un programme. l'utilisateur peut inserer des assertions ou/et annoter les variables à analyser ou laisser **KLEE** explorer de manière autonome l'espace des exécutions. Cette approche rend possible la détection de diverses erreurs, notamment les violations d'assertions, les corruptions mémoire ou les exceptions non gérées.

Architecture et fonctionnement de KLEE

Le processus typique d'analyse avec **KLEE** comporte deux étapes. Premièrement, l'utilisateur annote les variables d'entrée souhaitées comme symboliques ou ajouter des assertions dans le code source. Deuxièmement, le programme C est compilé en une representation intermédiaire LLVM-IR (LLVM Intermediate Representation) en utilisant la commande `clang -S -emit-llvm -o`. Enfin, **KLEE** interprète ce LLVM-IR à travers une architecture modulaire dont les composants clés sont illustrés Figure 2.6.

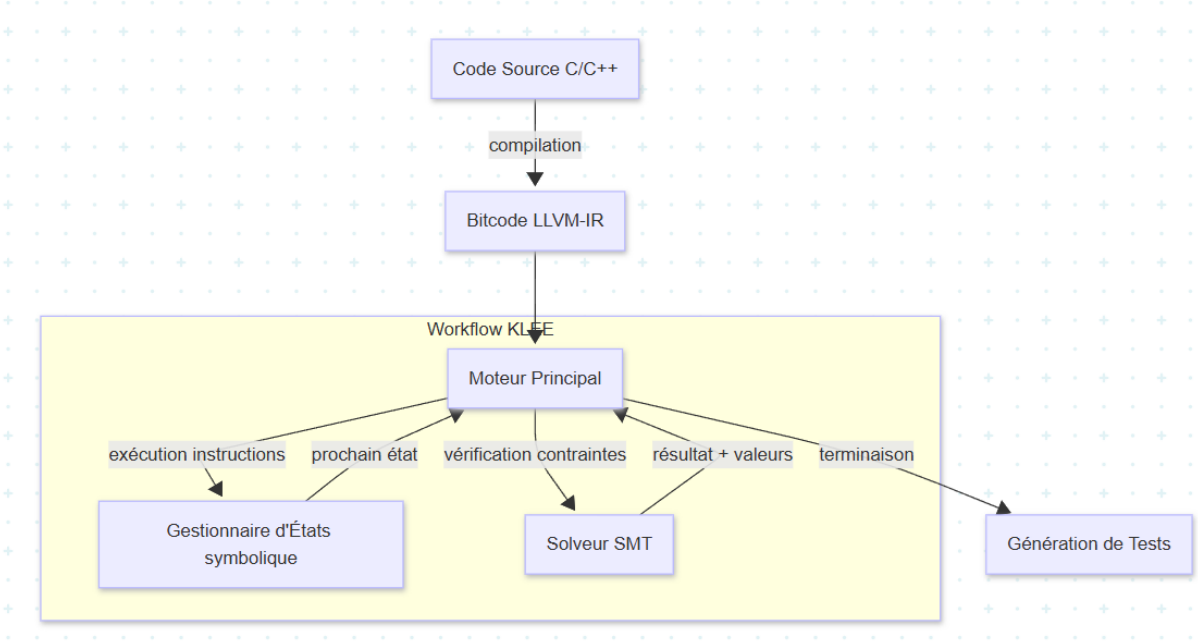

Figure 2.6: Workflow et architecture modulaire de **KLEE**

Moteur d'exécution symbolique : Il est chargé d'interpréter les instructions du LLVM-IR, de maintenir les états d'exécution et de dupliquer (*fork*) ces états lorsqu'une condition symbolique apparaît. Chaque état conserve un pointeur d'instruction, un modèle mémoire et un ensemble de contraintes accumulées. Les conditions issues de l'exploration symbolique sont sont transmises à **l'interface des SMT solvers**.

Interface avec les SMT solvers. Ce module a pour rôle de communiquer les formules logiques (contraintes) à des SMT solvers externes (tels que STP, Z3 ou CVC4). Le solveur détermine la satisfiabilité de ces contraintes. Les états correspondant à des chemins impossibles (contraintes insatisfaisables) sont éliminés, tandis que les chemins valides (contraintes satisfaisables) sont poursuivis. Le solveur peut également fournir des valeures concrètes pour les variables symboliques.

Le gestionnaire d'états. ce composant maintient une file des états actifs et sélectionne le prochain état à explorer selon différentes heuristiques (par exemple, recherche en largeur d'abord ou guidée par la couverture de code).

Environnement d'exécution. Afin de rendre l'outil utilisable dans des environnements réalistes, **KLEE** intègre également une version instrumentée de la bibliothèque standard C (uClibc) ainsi qu'un modèle pour les appels **POSIX (Portable Operating System**

Interface) [21]. Ceux-ci permettent de modéliser des interactions avec le système (fichiers, réseau, etc.) de manière symbolique.
Au terme de l'analyse, **KLEE** fournit un ensemble de sorties comprenant :

- Des fichiers de test concrets (`.ktest`) ;
- Des rapports d'erreurs détaillant les violations détectées ;
- Des statistiques de couverture de code.

Illustration pratique Les variables d'entrée d'un programme peuvent être transformées en variables symboliques grâce à la fonction `klee_make_symbolic` comme montrer dans Listing 2.6 ligne 3. Cette opération permet à **KLEE** de traiter leurs valeurs comme indéterminées et de les explorer selon différents scénarios possibles. L'exemple suivant illustre ce mécanisme :

Listing 2.6: Specifier une variable symbolique avec **KLEE**

```
1 int main() {
2   int x;
3   klee_make_symbolic(&x, sizeof(x), "x");
4   if (x > 0) { /* Chemin A */ }
5   else { /* Chemin B */ }
6 }
```

Dans cet exemple, la variable `x` est considérée comme symbolique, ce qui permet à l'exécution symbolique d'explorer les chemins où la condition est vraie, ainsi que ceux où elle est fausse.
Lorsque **KLEE** rencontre une instruction conditionnelle, il crée autant d'états que de branches possibles, chacun associé à une condition de chemin représentant les décisions accumulées au cours de l'exécution.

Terminaison. Enfin, l'exploration symbolique ne se poursuit pas indéfiniment. Elle peut s'achever lorsque tous les chemins atteignables ont été parcourus, ce qui demeure rare pour des programmes complexes. Plus fréquemment, l'exécution est interrompue en raison de **contraintes de ressources**, telles que la mémoire disponible ou la durée maximale d'analyse. Elle peut également s'arrêter lorsqu'une **violation d'assertion spécifiée par l'utilisateur** est détectée, ce qui permet de révéler des comportements indésirables du programme.

Depuis sa présentation initiale dans [21], **KLEE** a eu un impact considérable aussi bien dans la recherche académique que dans l'industrie. Son architecture modulaire et ses optimisations (par exemple le cache de requêtes au solveur) ont inspiré de nombreux

travaux ultérieurs, faisant de **KLEE** une référence incontournable dans le domaine de la vérification automatique et du test de programmes.

2.5.2 Frama-C

Frama-C est un outil d'analyse statique open source pour les programmes C, conçu avec une architecture modulaire accompagné avec un langage de spécification dédié [38]. L'objectif principal de Frama-C est la vérification formelle des programmes complexes ou critiques, c'est-à-dire la conformité des implémentations aux spécifications formelles [25]. Ces spécifications sont soit introduites par l'utilisateur, soit générées par certains plugins lors de l'analyse.
Il permet de prouver diverses propriétés, telles que l'absence d'erreurs d'exécution (*runtime errors*) ou des propriétés fonctionnelles [41]. Au-delà de la simple détection de défauts, Frama-C vise à garantir l'absence de bugs.

Architecture et fonctionnment de la plateforme Frama-C. Le noyau de Frama-C repose sur une version modifiée de CIL (C Intermediate Language), une infrastructure de traitement du langage C qui convertit les programmes en une représentation intermédiaire normalisée. Cette représentation, comparable dans son rôle à celui du *Java Bytecode* ou du LLVM-IR, a été étendue par Frama-C pour intégrer les annotations ACSL (ANSI/ISO C Specification Language).
Le processus d'analyse et de la verification traverse plusieures phase intermédiaires illustrées dans la figure 2.7.
Dans un premier temps, Frama-C prend en entrée un programme C, accompagné ou non d'annotations ACSL. Ces annotations, insérées directement dans le code sous forme de commentaires spéciaux (`/*@ ... */` ou `//@ ...`), constituent un langage de spécification formel permettant de définir des propriétés fonctionnellescomme : des contrats de fonctions, des assertions ou encore des invariants de boucles [19]. Elles complètent ainsi le code en fournissant une description logique des comportements attendus.

Phase de préparation des sources. Avant toute analyse, le code et ses annotations subissent une phase de préparation des sources composée de trois étapes successives :

- le *pré-traitement*, qui utilise par défaut le préprocesseur GCC (GNU Compiler Collection) pour traiter les fichiers `.c`, `.h` et prend en charge les annotations ACSL (`-pp-annot`) ainsi que des macros dédiées comme `__FC_MACHDEP_XXX` selon l'architecture cible spécifiée par `-machdep` [25].

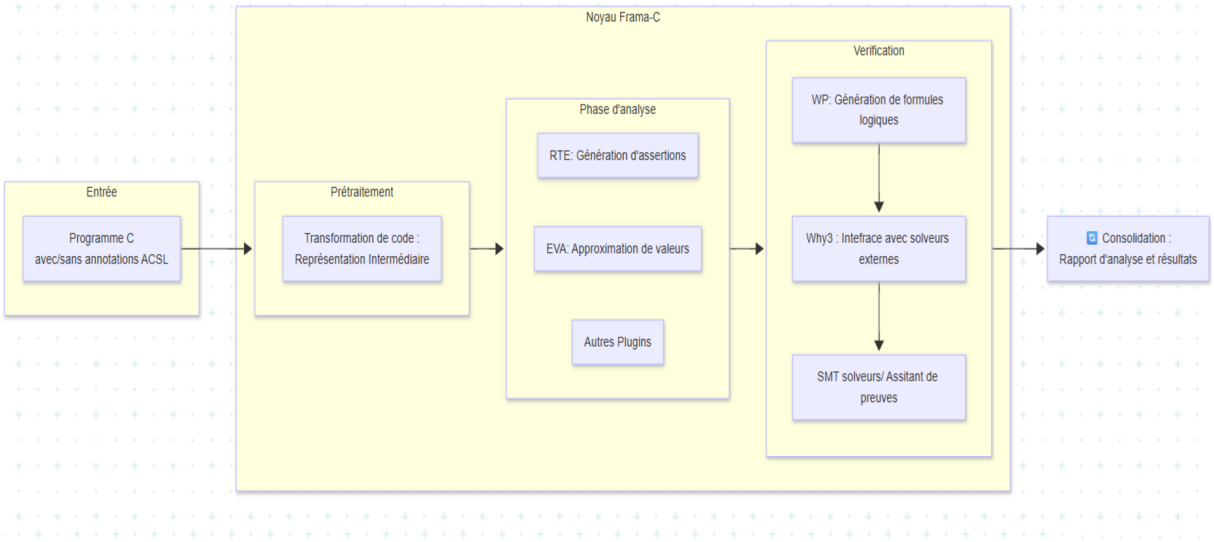

Figure 2.7: Workflow et architecture modulaire de **Frama-c**

- la *fusion*, qui effectue l'analyse syntaxique, la vérification des types et la mise en correspondance entre code et spécifications [25], avec possibilité de signaler ou d'interrompre l'exécution en cas d'erreur d'annotation;
- la *normalisation*, qui applique des transformations locales préservant la sémantique afin de produire une version équivalente mais structurée pour l'analyse. Cette version normalisée est représentée sous la forme d'un AST [41], lequel constitue la base unifiée sur laquelle opèrent l'ensemble des plugins de la plateforme [25].

Phase d'analyse. Une fois l'AST obtenu, différents plugins interviennent pour appliquer leurs techniques d'analyse et renforcer le processus de vérification [41].

Run-Time Errors (RTE). Insère des assertions pour détecter les **erreurs d'exécution courantes** (division par zéro, débordement, pointeurs invalides, etc.) [41]. Le plugin RTE gère notamment les divisions par zéro, les dépassements de capacité (overflows) et les déréférencements de pointeurs invalides. Il vise à prouver l'absence de ces erreurs.

Evolved Value Analysis (EVA). Calcule une **sur-approximation des valeurs possibles** par **interprétation abstraite**, ce qui permet de prouver l'absence d'erreurs d'exécution [55]. EVA est un plugin qui utilise l'interprétation abstraite pour calculer les domaines de valeurs de chaque variable du programme [41], étant particulièrement efficace pour vérifier l'absence d'erreurs numériques d'exécution. Il infère une

sur-approximation des valeurs possibles pour chaque variable à chaque point du programme.

D'autres plugins spécialisés peuvent intervenir selon les besoins, complétant l'analyse avec des perspectives variées [41, 25]. Frama-C est une plateforme extensible et collaborative qui vise à combiner diverses techniques d'analyse via des plugins pour garantir l'absence de bugs.

Phase de vérification. Certaines analyses, en particulier le plugin *WP* (Weakest Precondition), traduisent les annotations et propriétés du programme en formules logiques [19, 41]. *WP* génère des **conditions de vérification (VCs)** à partir des annotations du code. Ces conditions de vérification sont ensuite transmises à **Why3**, qui fait l'interface avec des solveurs SMT solvers comme Alt-Ergo, Z3 ou CVC5 [19], ou avec des assistants de preuve interactifs tels que Coq [41]. Cette étape permet d'obtenir des **garanties formelles sur la correction du code**.

Enfin, le noyau de Frama-C centralise les résultats produits par les différents plugins, générant un rapport d'analyse unifié [25]. Ce mécanisme d'intégration assure une **vision cohérente de l'état de vérification du programme**, qu'il s'agisse de propriétés prouvées, d'alertes ou de preuves incomplètes [25].

2.6 Conclusion

Dans ce chapitre, nous avons présenté les fondements nécessaires à la compréhension de l'analyse et de la vérification des programmes. Nous avons d'abord examiné le **processus de compilation**, qui constitue le point de passage essentiel entre l'abstraction du code source et son exécution sur une architecture matérielle donnée. Cette étape met en lumière l'importance de l'environnement d'exécution — compilation, système d'exploitation, architecture — dans la manifestation de comportements parfois inattendus.

Nous avons ensuite détaillé les **principales méthodes d'analyse** : statique, dynamique et hybride (exécution symbolique). Chacune présente des avantages et des limites complémentaires. L'analyse statique permet une exploration globale et précoce, sans exécution, mais peut produire des résultats approximatifs. L'analyse dynamique observe le comportement réel du programme, au prix d'une couverture limitée aux chemins effectivement explorés. L'exécution symbolique et concolique, quant à elles, combinent ces deux approches pour accroître la profondeur et la précision de l'analyse.

Sur cette base, nous avons introduit les **méthodes de vérification**, en distinguant les approches formelles — offrant des garanties mathématiques fortes — et les approches fondées sur les tests, plus empiriques mais indispensables dans la pratique. Enfin, nous avons présenté deux outils emblématiques, *KLEE* et *Frama-C*, qui illustrent la mise en œuvre concrète de ces techniques dans des contextes réels d'analyse et de vérification de programmes en C.

Ces éléments constituent le **socle conceptuel** sur lequel s'appuie notre approche. Dans le chapitre suivant, nous préciserons comment ces méthodes sont articulées et adaptées dans notre travail pour traiter des problématiques de sécurité et de sûreté dépendantes de l'environnement d'exécution.

3

Approche de vérification formelle sensible au contexte d'exécution

Sommaire

3.1 Introduction

Dans ce chapitre, nous présentons l'approche que nous proposons pour identifier de manière proactive les vulnérabilités potentielles dans les programmes C. Nous nous concentrons particulièrement sur les vulnérabilités liées aux défauts de sûreté, qu'elles soient introduites par les développeurs durant la phase de développement ou générées par l'environnement d'exécution lors du déploiement.

Nous commençons par un aperçu global de l'approche, puis nous détaillons chacune de ses phases ainsi que ses composants principaux. Nous expliquons également les choix stratégiques qui ont orienté notre démarche. Enfin, nous concluons en analysant les limites de notre approche et en explorant les extensions potentielles.

3.2 Vue d'ensemble de l'approche

L'approche adoptée repose sur l'utilisation de méthodes formelles, transformant la détection de vulnérabilités en un problème de satisfiabilité. Cette démarche permet ainsi d'exploiter des outils de vérification formelle pour garantir la sécurité du programme. En effet, comme le souligne la littérature, les vulnérabilités peuvent découler d'erreurs de codage et/ou d'une méconnaissance de la sémantique du langage, voire des subtilités d'interprétation propres aux différentes architectures. Bien que certaines de ces vulnérabilités soient parfois bien documentées, leur compréhension reste souvent difficile pour les ingénieurs, dont le travail n'est pas toujours simple.

C'est pourquoi nous avons opté pour une approche permettant aux développeurs de vérifier la présence de vulnérabilités, en prenant en compte le contexte d'exécution, les spécificités des architectures et les différents modes d'exécution. Cette information, collectée à partir de diverses sources telles que les bonnes pratiques [70, 23, 79], sera consignée dans une base de données et utilisée comme donnée d'entrée pour la vérification que nous entreprenons.

Plusieurs paramètres sont susceptibles d'influencer le comportement d'un programme en C/C++. Ces paramètres sont principalement liés aux différents composants de l'environnement d'exécution, notamment :

- **Le compilateur** : Les différences entre les compilateurs ou leurs versions peuvent provoquer des variations dans la performance, le comportement ou l'interprétation du code. Aussi l'utilisation des **options de compilation**. En effet, des choix tels que les optimisations, la gestion des exceptions ou d'autres paramètres

influencent directement l'exécution du programme.

- **Les bibliothèques externes** : L'utilisation de bibliothèques peut introduire des vulnérabilités, comme exemples celles signalées dans CVE-2014-01601[1] et CVE-2021-293902[2]. L'utilisation de bibliothèques peut aussi modifier les performances des programmes selon les implémentations.
- **Le système d'exploitation**: Les spécificités des systèmes (allocation de mémoire, gestion des ressources, appels système) impactent significativement le comportement du programme.
- **Le processeur**: Le jeu d'instructions, le parallélisme, la gestion du cache et d'autres caractéristiques matérielles influencent également les performances et le comportement.

Les paramètres des composants de la plateforme d'exécution doivent être soigneusement pris en compte dans l'analyse pour garantir la fiabilité, la portabilité et la sécurité du programme dans des environnements variés.

Notre approche d'analyse repose sur trois éléments fondamentaux qui constituent les entrées nécessaires à l'analyse :

- **(i) Le code source du programme à analyser** : Le programme constitue la base de l'analyse. Son étude permet d'identifier les mécanismes internes, les interactions possibles avec l'environnement et les points sensibles en termes de sécurité et de sûreté.
- **(ii) Les caractéristiques de l'environnement d'exécution cible** : Cela inclut des informations telles que le système d'exploitation, l'architecture du processeur, les bibliothèques utilisées, ou encore les configurations du compilateur. Ces éléments influencent directement le comportement du programme, en particulier lorsqu'il est déployé dans un environnement spécifique.
- **(iii) Les contraintes de sécurité ou de sûreté** : Ces contraintes définissent les exigences que le programme doit respecter, comme l'absence d'erreurs critiques, la résistance aux attaques ou la conformité à des standards industriels.

Notre méthodologie s'articule autour de deux composantes principales, chacune étant élaborée à des étapes distinctes de l'analyse :

1. Construction d'une base de connaissances : Cette base est construite de manière continue et demeure pérenne. Elle regroupe des assertions de sûreté et des propriétés vérifiables, adaptées à divers environnements d'exécution. Elle s'enrichit

[1] Célèbre vulnérabilité de la bibliothèque *OpenSSL* Heartbleed, permettant aux attaquants de lire des données sensibles à partir de la mémoire des systèmes.

[2] La bibliothèque *libpng*, utilisée pour la gestion des fichiers image PNG, présentait une vulnérabilité de dépassement de tampon pouvant entraîner l'exécution de code arbitraire.

à partir de multiples sources : corpus de règles de sécurité, standards tels que *MISRA* [70] ou *CERT* [79], ainsi que des analyses spécifiques à des configurations matérielles et logicielles. Les assertions abordent différents aspects, comme la gestion de la mémoire, l'accès aux ressources critiques, et les comportements liés aux interruptions et au temps.

2. Méthode d'analyse et de vérification formelle. Cette méthode est appliquée à chaque nouvelle analyse de programme. Elle utilise le code du programme, les assertions issues de la base de connaissances pour vérifier le respect des contraintes spécifiées par l'utilisateur. Elle repose sur des techniques formelles robustes, l'analyse statique et les preuves logiques. Son objectif est de détecter tout écart potentiel entre le comportement attendu du programme et son comportement réel dans l'environnement d'exécution cible.

La Figure 3.1 illustre en détail les différentes phases de la méthodolgie d'analyse.

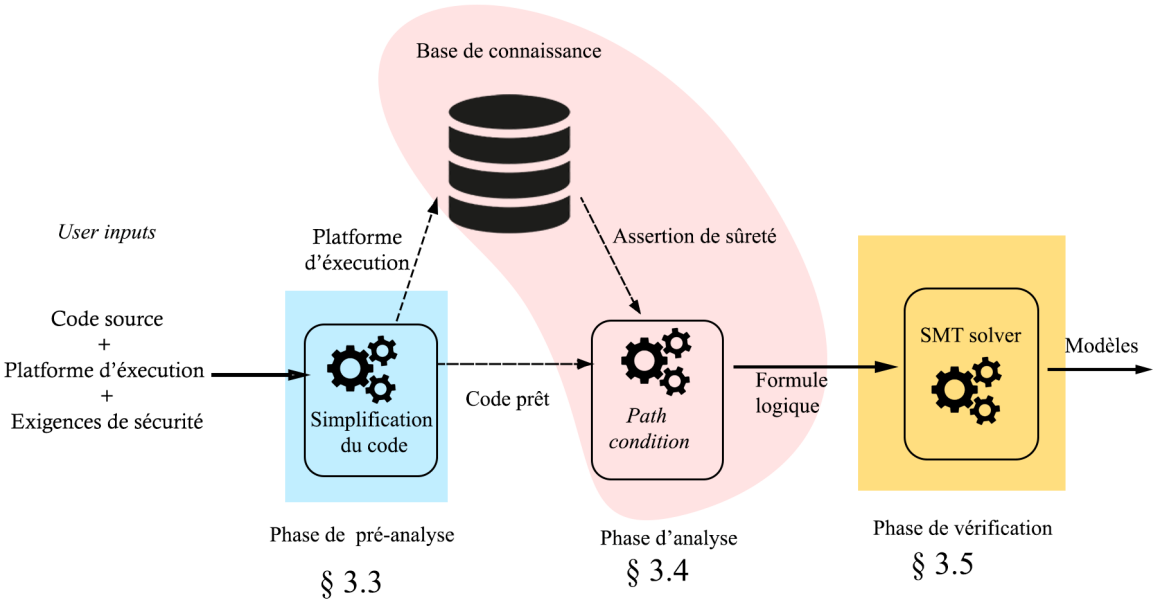

Figure 3.1: Vue d'ensemble de l'approche d'analyse formelle.

3.3 Phase de pré-analyse

L'une des premières questions qui s'est posée a été le choix du format de programme à analyser. En effet, les analyses peuvent être menées à différents niveaux : code source, e.g [61, 76], assembleur, e.g [78] ou binaire, e.g [17]. Ce choix est déterminant,

car chaque niveau exige une expertise particulière pour déchiffrer les transformations appliquées et repérer les points de vulnérabilité. Afin de rendre le processus plus accessible et efficace, nous avons privilégié une analyse directe du code source, tirant parti des techniques et outils d'analyse statique pour simplifier le travail et optimiser la détection des vulnérabilités.
Avant de commencer l'analyse du code, un prétraitement est nécessaire pour préparer le code à une vérification formelle efficace. Ce prétraitement repose sur l'utilisation d'outils d'analyse statique afin d'identifier les portions de code liées aux contraintes de sécurité spécifiées par l'utilisateur et de simplifier les structures complexes. Ces transformations normalisent le code, le rendant plus accessible à une analyse approfondie et rigoureuse. Deux étapes clés structurent ce processus :

1. Déroulement des boucles (loop unrolling) : Cette étape consiste à déplier les boucles sur un nombre prédéfini d'itérations (w). Cela réduit la complexité introduite par les boucles, facilitant leur analyse statique et augmentant la précision des outils.
2. Analyse structurelle, **transformation de type SSA** : Cette transformation, introduite par [30], attribue une valeur unique à chaque variable dans le code. Cela simplifie la génération des contraintes pour les solveurs SMT en rendant les dépendances et les flux de données plus explicites.

Ce pré-traitement est crucial pour garantir que le code est adapté à une analyse formelle approfondie. En simplifiant et en structurant le code de manière optimale, il permet de vérifier avec rigueur les contraintes de sécurité et de garantir la conformité aux normes de sûreté.

3.3.1 Déroulement des boucles

Les boucles constituent un défi majeur dans l'analyse des programmes en raison de leur capacité à introduire une complexité potentiellement infinie dans les flux d'exécution. Leur nature répétitive et conditionnelle complique la modélisation et la vérification formelles, car elles peuvent générer un nombre indéterminé d'états à explorer. Sans traitement spécifique, les boucles rendent les outils d'analyse statique inefficaces ou imprécis, compromettant ainsi la rigueur de l'évaluation des contraintes de sécurité.

Pour surmonter les défis liés à l'analyse des boucles, plusieurs techniques ont été développées afin de réduire leur complexité tout en préservant la précision des résultats [92, 60, 66].

Parmi ces approches, on trouve le déroulement de boucle *(loop unrolling)*, qui simplifie

les structures en les dépliant sur un nombre fixe d'itérations, l'inférence d'invariants *(loop invariant inference)* pour extraire des propriétés invariantes et le résumé de boucle *(loop summarization)*, qui condense la logique des boucles en une abstraction globale. Ces méthodes, combinées à des transformations comme la transformation de type SSA, jouent un rôle clé dans l'optimisation et la fiabilité de l'analyse formelle des programmes. Notre approche repose principalement sur la technique de *loop unrolling*, qui remplace une boucle par une série d'instructions répétées, éliminant ainsi sa structure itérative. En déroulant la boucle un nombre d'itérations défini par l'utilisateur, cette méthode simplifie l'analyse du programme et facilite la détection de comportements indésirables.

Inspirés par les travaux de [6], nous appliquons dans notre approche une transformation qui convertit une boucle *while* en une série de conditions *if*, suivie de la boucle restante. Comme illustré dans la transformation suivante, qui explicite le dépliage de la première itération :

while$(cond)$ $\{\mathscr{P}\}$ se transforme en *if* $(cond)$ $\{\mathscr{P}$ *while*$(cond)\{\mathscr{P}\}\}$

où $cond$ représente de la condition de la boucle et $\mathscr{P}$ le corps de celle-ci.
Le dépliage peut être effectué jusqu'à w fois, w étant le nombre d'itérations spécifié par l'utilisateur.

Listing 3.1: Exemple d'une boucle while

```
1
2 int i = 0;
3 while(i<100){
4   //do stuff ...
5   i++;
6 }
```

Exemple 1 *Le résultat du dépliage avec $w = 5$ de la boucle présentée dans le Listing 3.1 est donné dans le Listing 3.2. Dans ce code du Listing 3.2, chaque itération est représentée explicitement, ce qui simplifie l'analyse statique de la boucle tout en réduisant la complexité.*
Enfin, puisque nous traitons un nombre fini d'itération, nous n'avons pas inclus le while dans la dernière itération, il est possible d'étendre cette partie plus tard pour calculer l'invariant ou autre propriété de boucle.

Listing 3.2: Déroulement de la boucle de l'exemple du Listing 3.1 sur 5 itérations (w=5)

```
1
2  int i=0;
3  if( i < 100){ // Iteration 0
4    //do stuff ...
5    i++;
6  }
7  if( i < 100){ // Iteration 1
8    //do stuff ...
9    i++;
10 }
11 if( i < 100){ // Iteration 2
12   //do stuff ...
13   i++;
14 }
15 if( i < 100){ // Iteration 3
16   //do stuff ...
17   i++;
18 }
19 if( i < 100){ // Iteration 4
20   //do stuff ...
21   i++;
22 }
```

3.3.2 Transformation de type SSA

Dans notre approche, la transformation en SSA joue un rôle clé dans la pré-analyse. Elle simplifie la gestion des variables et des affectations en garantissant que chaque variable est définie une seule fois dans le programme. En éliminant les **effets de bord**, la SSA assure que chaque variable conserve une valeur unique tout au long de l'exécution, facilitant ainsi le raisonnement formel et répondant à l'exigence **d'immutabilité** des solveurs SMT. Sans cette transformation, l'intégration des solveurs SMT serait très complexe.

Les principaux avantages de la SSA dans notre cadre :

- **Assignation unique :** Chaque variable est assignée une seule fois, avec une nouvelle version créée à chaque réassignation. Cela évite les ambiguïtés sur les valeurs des variables et simplifie le suivi des dépendances.
- **Simplification du flot de contrôle :** La SSA assure que chaque variable a une valeur constante sur chaque chemin d'exécution, simplifiant ainsi l'encodage en logique SMT.
- **Encodage SMT efficace :** Chaque variable correspond à une seule valeur,

facilitant la traduction en formules logiques. Les solveurs SMT peuvent alors traiter les variables de manière plus simple.

- **Fonctions phi pour jonctions de contrôle :** Les fonctions phi combinent différentes valeurs d'une variable après un `if-else`, rendant l'encodage du flot de contrôle plus fluide.
- **Réduction des effets de bord :** L'élimination des réassignations multiples réduit les effets de bord, rendant les formules SMT plus claires et prévisibles.

En résumé, la transformation SSA simplifie la gestion des variables et du flot de contrôle, ce qui facilite la traduction du programme en formules SMT.

Traitement des appels de fonction: Dans notre méthodologie, nous n'adoptons pas une approche hiérarchique pour le traitement des fonctions, c'est-à-dire que nous ne procédons pas à leur dépliage pour analyser les appels. Concernant les appels de fonctions, nous nous concentrons sur leurs valeurs de retour, en particulier sur les types associés à ces valeurs. Seuls les types de valeurs de retour des fonctions sont pris en compte, et ces valeurs sont représentées en utilisant un codage spécial, tenant compte à la fois du signe (S pour signé, U pour non signé) et de la taille de l'entier. Par exemple, des notations telles que `symbSint8, symbUint8, symbSint16, . . . , symbUint64` sont introduites pour coder les valeurs selon des plages spécifiques définies dans la norme[3] :

- `symbSint8` : $[-2^7, 2^7 - 1]$
- `symbUint8` : $[0, 2^8 - 1]$
- `symbSint16` : $[-2^{15}, 2^{15} - 1]$
- `symbUint16` : $[0, 2^{16} - 1]$
- `symbSint32` : $[-2^{31}, 2^{31} - 1]$
- `symbUint32` : $[0, 2^{32} - 1]$
- `symbSint64` : $[-2^{63}, 2^{63} - 1]$
- `symbUint64` : $[0, 2^{64} - 1]$

Ces notations permettent de représenter avec précision les types de retour des fonctions, garantissant que l'analyse formelle prenne en compte à la fois le signe et la taille des valeurs manipulées. En utilisant ces conventions, nous pouvons traiter les appels de fonction dans notre formalisation tout en respectant les spécifications de la norme C/C++.

[3] https://en.cppreference.com/w/c/language/arithmetic_types

3.4 Phase d'analyse

Une fois le code préparé pour l'analyse, la seconde phase consiste à formaliser le comportement du programme en générant des contraintes logiques basées sur les conditions d'exécution et les spécifications de sécurité. L'objectif est de déterminer si, pour un chemin d'exécution donné, une configuration du programme peut respecter ou violer les contraintes de sécurité. Cette analyse se déroule en trois étapes principales : (1) Construction de la condition de chemin; (2) Intégration des contraintes architecturales; (3) Génération de la formule logique.

3.4.1 Construction de la condition de chemin d'exécution

La condition de chemin d'exécution (*path condition*) est une expression logique qui décrit les contraintes nécessaires pour qu'un chemin spécifique dans un programme soit emprunté lors de son exécution. Elle joue un rôle clé dans la modélisation des différents chemins possibles d'un programme et constitue une base essentielle pour la vérification formelle.

Dans notre méthodologie, le calcul de la condition de chemin repose sur l'utilisation de l'arbre syntaxique AST généré par le compilateur Clang. L'AST représente chaque instruction du programme sous forme de nœuds organisés en une structure arborescente.

Nous parcourons l'AST en ciblant le nœud correspondant à l'instruction analysée, puis remontons vers la racine de la fonction. Lors de ce parcours, les conditions des branches conditionnelles rencontrées (comme celles issues des if, while, etc.) sont empilées. Le résultat final est une expression logique formée par la conjonction de toutes les conditions empilées.

Cette condition de chemin ainsi obtenue sert de base pour la génération de la formule logique utilisée dans les étapes suivantes de l'analyse. Les aspects techniques détaillés de ce processus seront développés dans le chapitre 5, consacré à l'implémentation.

Exemple 2 *Considérons le code du Listing 3.3. Nous cherchons à déterminer la condition de chemin menant à l'exécution de l'instruction* `assert(...)` *située dans la ligne 7 marquée par* ***chemin 2*** *.*

Commentaire explicatif :
L'exécution du programme débute par l'évaluation de la condition `if (x > 0)`*. L'accès au bloc* `else` *implique que cette condition soit évaluée à faux, ce qui se traduit par la contrainte logique* $\neg(x > 0)$*. À l'intérieur de ce bloc, une seconde condition est ensuite*

testée, à savoir `if (y < 10)`*. L'exécution de l'instruction* `assert(...)` *n'est possible que si cette condition est vérifiée, ce qui ajoute la contrainte* $(y < 10)$ *au chemin d'exécution considéré.*

La condition de chemin associée au chemin 2 correspond alors à la conjonction des contraintes accumulées le long de ce chemin, et s'exprime par la formule suivante :

$$(y < 10) \wedge \neg(x > 0).$$

Listing 3.3: Exemple de génération de condition de chemin

```
1  void foo(int x,int y){
2  if( x>0 ){
3    doSomthing();// Chemin 1
4    ...
5  }else{
6    if(y<10){
7      assert(...); // Chemin 2
8    }
9  ...
10 }
11 }
```

3.4.2 Contraintes architecturales

Notre méthodologie se distingue par l'intégration des contraintes architecturales dans notre analyse. Nous partons du principe que les bugs ne proviennent pas uniquement du code, mais peuvent également résulter de l'architecture d'exécution. Pour cela, nous avons élaboré un ensemble de formules spécifiques aux différentes architectures d'exécution.

Certaines de ces formules portent sur des propriétés liées au débordement mémoire. Ce phénomène se produit lorsque la valeur d'une variable ou d'un calcul dépasse la capacité de stockage allouée, un comportement souvent imprévisible, en particulier dans le cas des calculs sur des entiers signés.

Dans cette section, nous introduisons une première modélisation du phénomène de débordement, suffisante pour capturer son intuition et permettre son intégration dans notre approche. Une formalisation mathématique plus rigoureuse, adaptée à l'encodage SMT, sera présentée ultérieurement dans la Section 4.3.3.

Pour cela, nous modélisons le débordement à l'aide d'un modèle sémantique basé sur

la taille mémoire n. La valeur d'un entier signé x, codée sur n bits, peut être représentée sous différentes formes, telles que la valeur nominale ou la valeur effectivement stockée après troncature, notée $\bar{x}$. Ce modèle, introduit et exploité dans nos travaux antérieurs [57], permet de capturer le comportement de débordement des entiers. Cette relation est formalisée par l'équation suivante :

$$x = 2^n \times k + \bar{x} \tag{3.1}$$

oú k est un entier relatif et n représente la capacité mémoire pour encoder l'entier. Dans les architectures modernes, n prend généralement les valeurs 8, 16, 32 ou 64. Si $k = 0$, alors $\bar{x} = x$, ce qui signifie que la valeur est correcte. En revanche, si $k \neq 0$, un débordement est survenu.

Pour identifier les situations oú une représentation devient incorrecte, que ce soit lors d'un calcul arithmétique ou d'une affectation, nous utilisons l'assertion suivante :

$$(x = 2^n \times k + \bar{x}) \wedge (k \neq 0)$$

Cette assertion, dérivée de l'équation du modèle sémantique, est instanciée avec les valeurs de n provenant de la table préalablement élaborée.

Exemple 3 *L'instanciation de cette assertion pour le calcul d'un résultat de type long selon l'architecture X86_64, telle que définie dans la table Table 3.1, génère la formule suivante :*

$$(x = 2^{64} \times k + \bar{x}) \wedge (k \neq 0)$$

Table 3.1: Portion de la table concernant la taille memoire du type Int à travers different environement d'execution

Integer type	**Target Arch.**	**OS ; Compiler**	**Compiler options**	**Size *bits***
⋮	⋮	⋮	⋮	⋮
long	TMS320C28x	Compiler v22.6.0.LTS Windows ; MSVC	$flags$	32
	X86 / X86_64	Linux ; Clang/GCC macOS ; Clang	$m32 \in flags$	
	X86_64	Linux ; Clang/GCC macOS ; Clang	$m32 \notin flags$	64
⋮	⋮	⋮	⋮	⋮

Comme le souligne [35]: *"La situation est complexe, car tous les débordements ne sont pas des bugs. De meilleurs outils doivent être développés, mais une compréhension approfondie des problèmes liés à ces erreurs n'existe pas encore"*.

Cependant, notre analyse ne vise pas à traiter tous les débordements mémoire. Elle se concentre exclusivement sur ceux qui sont susceptibles de compromettre les contraintes de sécurité, apportant ainsi une réponse ciblée et pertinente à ces problématiques critiques. La table 3.1 illustre clairement l'influence des différents paramètres d'un environnement d'exécution sur la taille mémoire allouée aux entiers. Elle sera utilisée pour instancier l'espace mémoire (n) dans l'équation précédemment décrite (3.1).

3.4.3 Modélisation logique du problème

La dernière étape de notre approche consiste à générer la formule logique qui combine la condition de chemin (*PC*) et la contrainte de sécurité (*SC*), dans un context d'execution (*EC*) particulier, de manière à détecter toute affectation des variables qui satisfasse *PC* tout en violant *SC*. Formellement, cette combinaison s'exprime ainsi :

$$EC \vdash PC \wedge \neg SC$$

Parcours inverse et calcul des variables impliquées: Pour assurer la cohérence de cette formule, nous identifions les variables nécessaires par un parcours inverse de l'AST depuis l'instruction cible au niveau du programme. Ce processus se décompose en plusieurs étapes :

1. Nous initialisons l'ensemble des variables V en y incluant toutes celles utilisées dans la condition de chemin (*PC*) et dans la contrainte de sécurité (*SC*).
2. En parallèle, nous définissons l'ensemble D, contenant les définitions de chaque variable $v_i \in V$.
3. Ensuite, nous enrichissons V en ajoutant les variables présentes dans les définitions D_j de chaque variable déjà identifiée. Cette opération se poursuit de manière récursive jusqu'à ce que toutes les variables impliquées directement ou indirectement dans *PC* ou *SC* soient identifiées dans V.

Cette collecte exhaustive des variables permet de couvrir toutes les interactions potentielles entre *PC* et *SC* dans le contexte du programme.

Enfin, nous intégrons les contraintes architecturales aux expressions arithmétiques présentes dans SC et D, assurant ainsi une modélisation précise du comportement du programme.

Intégration des contraintes architecturales : Une fois les variables identifiées, les contraintes architecturales (comme les tailles des types de données, les options de compilation, etc.) seront exploitées pour l'intégration des contraintes architecturales comme expliqué dans la sous-section 3.4.2 pour que la formule logique reflète précisément le comportement du programme dans l'environnement cible.

3.5 Phase de vérification

La phase de vérification constitue l'étape finale de notre approche, visant à identifier les scénarios d'exécution potentiels oú les contraintes de sécurité pourraient être violées dans l'environnement spécifié par l'utilisateur.
Pour cela, nous générons un code au format SMT-LIB2, compatible avec divers solveurs SMT tels que CVC5, utilisé dans notre cas. Ce format permet d'appliquer la vérification formelle avec divers solveurs, assurant ainsi une couverture accrue en cas d'échec de résolution par un solveur spécifique. Le cas échéant, les résultats produits incluent des modèles fournissant des exemples d'affectations de variables entraînant une violation de sécurité.

3.5.1 Génération du code SMT-LIB

Les sections précédentes ont établi les trois ingrédients du problème de vérification : la condition de chemin PC, les contraintes de sécurité SC et le contexte d'exécution EC. La présente section décrit comment ces éléments sont assemblés en un code SMT-LIB2 soumis au solveur. Cette transformation est formalisée par l'Algorithme 1, qui reçoit en entrée le programme source P, la ligne correspondante à l'instruction cible ℓ, les contraintes contextuelles *FCxt*, les contraintes de sécurité *FSec* et la description de l'architecture $\mathcal{A}_p$, et produit en sortie un code SMT-LIB2 prêt à l'emploi. Le déroulement de cet algorithme sur un exemple, ainsi que des détails supplémentaires, sont présentés dans notre article [77].

Notations. Avant de présenter l'algorithme, nous introduisons les notations qu'il emploie.

- **Programme transformé $P^\sharp$.** Préalablement à l'analyse, les boucles du programme P sont déroulées jusqu'à une borne w fournie par l'utilisateur, et les

fonctions appelées sont remplacées par des variables symboliques typées (Section 3.3.2). Le programme résultant, sans boucle et aplati, est noté $P^\sharp$.

- **Ensemble des instructions cibles déroulées $\lambda^{\ell}_{P,w}$.** Lorsque l'instruction cible ℓ se trouve à l'intérieur d'une boucle, chaque occurrence de ℓ dans le programme déroulé $P^\sharp$ constitue un point d'analyse distinct. $\lambda^{\ell}_{P,w}$ désigne l'ensemble de ces occurrences après w déroulages.
- **Ensemble de variables *vars*(t).** Pour un terme t, *vars*(t) retourne l'ensemble des variables qui y apparaissent.
- **Opérateur de substitution $[\![\mathrm{T}]\!]_V$.** La notation $[\![\mathrm{T}]\!]_V$ remplace chaque variable de T par sa forme la plus récente dans l'état symbolique V ; si une variable est absente de V, la substitution est sans effet.
- **Opérateur de traduction $\widehat{\ }$.** La fonction $\widehat{F}$ convertit une formule logique F en sa représentation SMT-LIB2 en notation préfixe, en substituant les connecteurs logiques par les mots-clés correspondants du format.
- **Opérateur de concaténation $\oplus$.** Dans la génération de code, les chaînes SMT-LIB2 sont construites par concaténation ; $\oplus$ désigne cet opérateur.

Structure de l'algorithme. L'Algorithme 1 opère en deux phases principales.

Phase 1 — Construction de la formule d'exécution *FExec*. La formule *FExec* modélise la condition sous laquelle l'instruction cible ℓ est atteinte, tout en violant les propriétés de sécurité. Elle résulte de la conjonction de la condition de chemin (calculée par `computeCondPath`) et de la négation des contraintes de sécurité, substituées dans l'état symbolique courant :

$$\mathit{FExec} \leftarrow \texttt{computeCondPath}(P^\sharp, \ell) \ \wedge \ [\![\mathit{FCxt} \wedge \neg \mathit{FSec}]\!]_{\mathit{vars}(\Sigma^{\ell}_{P^\sharp})} \tag{3.2}$$

Lorsque ℓ se situe à l'intérieur d'une boucle, la formule devient une disjonction sur l'ensemble $\lambda^{\ell}_{P,w}$ des occurrences déroulées, chacune représentant un chemin d'exécution possible au sein de la boucle :

$$\mathit{FExec} \leftarrow \bigvee_{\ell^\sharp \in \lambda^{\ell}_{P,w}} \Big(\texttt{computeCondPath}(P^\sharp, \ell^\sharp) \ \wedge \ [\![\mathit{FCxt} \wedge \neg \mathit{FSec}]\!]_{\mathit{vars}(\Sigma^{\ell^\sharp}_{P^\sharp})}\Big) \tag{3.3}$$

Phase 2 — Construction de la formule d'environnement *FEnv*. La formule *FEnv* reflète l'exécution des expressions arithmétiques sur l'architecture cible $\mathcal{A}_p$. Elle est construite comme une conjonction sur l'ensemble des variables de *FExec*, dont chacune est traitée selon la forme que lui attribue le store σ :

- Si $\sigma(v) =$ *valeur* : on pose directement $v = \sigma(v)$, encodant ainsi les contraintes de plage du type (bornes inférieures et supérieures).
- Si $\sigma(v) =$ *Phi* : on invoque la fonction `encodePhi`, qui traduit les chaînes def-use en termes logiques conformément à la sémantique SSA.
- Si $\sigma(v) =$ *exp* (expression arithmétique) : on appelle `encodeArch`(v, *exp*, $\mathcal{A}_p$), qui remplace l'expression par son encodage architectural sensible aux débordements (cf. Algorithme 2).

Une fois *FEnv* construite, toutes les occurrences d'expressions arithmétiques dans *FExec* et *FEnv* sont substituées par leur encodage architectural, garantissant la cohérence des deux formules.

Phase 3 — Génération du code SMT-LIB. Les formules *FExec* et *FEnv* sont ensuite traduites en instructions SMT-LIB2 : déclaration de la logique et des options (dont `set-logic`), déclaration de chaque variable comme entier non borné (`declare-fun` v `() Int`), assertions des deux formules, puis appel au solveur (`check-sat` et `get-model`). Un résultat *sat* fournit un contre-exemple concret ; un résultat *unsat* constitue une preuve formelle d'absence de violation.

Encodage architectural — fonction `encodeArch`. Le cœur de l'encodage architectural est la fonction `encodeArch` (Algorithme 2). Elle reçoit une variable v, l'expression arithmétique *exp* qui lui est associée, et l'architecture cible $\mathcal{A}_p$. Elle crée d'abord une variable fraîche $\bar{x}_i$, destinée à capturer la valeur résiduelle en cas de débordement. Elle interroge ensuite la base de connaissances $\mathcal{KD}_A$ via la fonction `lookUp` pour obtenir la formule de débordement $FOver_{size}$, qui encode la condition $(\textit{exp} = 2^n \times k + \bar{x}_i) \wedge (k \neq 0)$ avec n déterminé par l'architecture. Le résultat retourné est une expression *if-then-else* :

$$\texttt{ite}\Big(FOver_{size},\ \bar{x}_i,\ \textit{exp}\Big)$$

Si la condition de débordement est satisfaite, la variable prend la valeur résiduelle $\bar{x}_i$ (la valeur effectivement stockée en mémoire après troncature) ; sinon elle prend la valeur mathématique exacte de l'expression.

3.5.2 Résolution par SMT solveurs

Une fois le code SMT-LIB généré, il est exécuté par un solveur SMT, comme CVC5. Le format SMT-LIB2 permet également l'utilisation de solveurs alternatifs, tels que Z3 ou Boolector, pour une flexibilité accrue.

Algorithm 1: Algorithme de génération du code SMT-LIB — COMPUTE_SMTCODE

Input: Programme P, instruction cible ℓ, formule de contexte *FCxt*, formule de sécurité *FSec*, profil d'architecture $\mathcal{A}_p$
Output: Code SMT-LIB *SMTCode*
Data: Table d'environnement σ, ensemble des chemins $\lambda^{\ell}_{P,w}$

Fonction `Compute_SMTCode`$(P, \ell,$ *FCxt*, *FSec*, $\mathcal{A}_p)$**:**
  if ℓ *est dans une boucle* **then**
    $w \leftarrow$ `read()` `// Borne de déroulage fournie par l'utilisateur`
    $\mathit{FExec} \leftarrow \bigvee_{\ell^\sharp \in \lambda^{\ell}_{P,w}} \Big(\texttt{computeCondPath}(P^\sharp, \ell^\sharp) \wedge [\![\mathit{FCxt} \wedge \neg \mathit{FSec}]\!]_{\mathit{vars}(\Sigma^{\ell^\sharp}_{P^\sharp})} \Big)$
  else
    $\mathit{FExec} \leftarrow \texttt{computeCondPath}(P^\sharp, \ell) \wedge [\![\mathit{FCxt} \wedge \neg \mathit{FSec}]\!]_{\mathit{vars}(\Sigma^{\ell}_{P^\sharp})}$
  `// Construction de la formule d'environnement`
  $\mathit{FEnv} \leftarrow \bigwedge_{\substack{v \in \mathit{vars(FExec)} \\ \text{t.q. } \sigma(v) = \mathit{valeur}}} \Big(v = \sigma(v) \Big) \wedge \bigwedge_{\substack{v \in \mathit{vars(FExec)} \\ \text{t.q. } \sigma(v) = \mathit{Phi}}} \Big(v = \texttt{encodePhi}(\sigma(v)) \Big)$
  $\wedge \bigwedge_{\substack{v \in \mathit{vars(FExec)} \\ \text{t.q. } \sigma(v) = \mathit{exp}}} \Big(\texttt{encodeArch}(v, \sigma(v), \mathcal{A}_p) \Big)$;
  `// Substitution des expressions arithmétiques dans les deux formules`
  $\mathit{FExec}\{\mathit{exp}/\texttt{encodeArch}(\_, \mathit{exp}, \mathcal{A}_p)\}$
  $\mathit{FEnv}\{\mathit{exp}/\texttt{encodeArch}(\_, \mathit{exp}, \mathcal{A}_p)\}$
  `// Génération du code SMT-LIB`
  SMTCode.add("(set-logic " $\oplus$ *Théorie* $\oplus$ ")")
  SMTCode.add("(set-option " $\oplus$ *Options* $\oplus$ ")")
  forall $v \in \mathit{vars}(\mathit{FExec}) \cup \mathit{vars}(\mathit{FEnv})$ **do**
    SMTCode.add("(declare-fun " $\oplus$ v $\oplus$ "() Int)")
  SMTCode.add("(assert " $\oplus$ $\widehat{\mathit{FExec}}$ $\oplus$ ")")
  SMTCode.add("(assert " $\oplus$ $\widehat{\mathit{FEnv}}$ $\oplus$ ")")
  SMTCode.add("(check-sat)")
  SMTCode.add("(get-model)")

Le solveur SMT analyse la formule logique et fournit un rapport de résultats selon trois cas :

- **SAT** (Satisfiable) : Un scénario satisfait la condition de chemin tout en violant la contrainte de sécurité ; le solveur fournit les valeurs concrètes des variables responsables de cette violation.
- **UNSAT** (Unsatisfiable) : Aucune affectation des variables ne conduit à une viola-

Algorithm 2: Fonction d'encodage architectural — ENCODEARCH

Input: Variable v, expression arithmétique *exp*, profil d'architecture $\mathcal{A}_p$
Data: Base de connaissances architecturales $\mathcal{KD}_A$
Output: Expression encodée tenant compte de la sémantique de débordement de l'architecture cible

```
1 Fonction encodeArch(v, exp, A_p):
2     x̄_i ← fresh()                    // Variable fraîche pour la valeur résiduelle
3     FOver_size ← lookUp(type(v), exp, A_p, KD_A, x̄_i)   // Formule de débordement
4     return ite(FOver_size, x̄_i, exp)        // Sélection selon l'occurrence d'un
        débordement
```

tion, indiquant que la sécurité est assurée.

- **UNKNOWN** : Le solveur ne parvient pas à déterminer la satisfiabilité, possiblement en raison de la complexité du problème.

Cette phase de vérification valide ainsi la conformité du programme aux contraintes de sécurité définies dans l'environnement d'exécution.

3.6 Conclusion

Dans ce chapitre, nous avons présenté une approche de vérification formelle des programmes en tenant compte de l'impact de l'environnement d'exécution sur leurs comportements. L'utilisation de techniques telles que le déroulement de boucle *loop unrolling* et la transformation en SSA, combinées avec les solveurs SMT, permet de garantir que les contraintes de sécurité définies par l'utilisateur sont vérifiées de manière rigoureuse.

Les avantages de cette approche sont multiples. Tout d'abord, elle permet d'intégrer l'impact du matériel sur le comportement du programme, un aspect souvent négligé par les outils d'analyse statique classiques. En se concentrant sur le corps des fonctions, elle réduit également les faux positifs, offrant ainsi des résultats plus précis et exploitables pour les développeurs. Cela facilite son intégration dans les processus de revue de code, rendant l'outil pratique à utiliser dans les cycles de développement quotidiens. Un autre avantage majeur de cette méthode est sa **modularité** et sa **flexibilité**. L'approche est composée de trois blocs principaux (prétraitement du code source, analyse formelle, et vérification) et peut-être étendue pour traiter d'autres aspects, tels que la gestion des pointeurs ou la concurrence. Cela permet aux utilisateurs, développeurs et théoriciens d'adapter l'approche en fonction des propriétés spécifiques du programme à vérifier. Ainsi, cette modularité permet d'élargir le champ d'analyse

au-delà des opérations arithmétiques.
De plus, un autre avantage important est notre base de connaissances, qui englobe les assertions de sûreté collectées et développées, elle sera accessible publiquement à la communauté, permettant aux outils existants d'exploiter ces assertions de sûreté pour enrichir leurs propres analyses et contribuer à une amélioration globale de la sûreté et la sécurité des logiciels.

Cependant, **certaines limites** subsistent. L'incapacité à traiter les boucles infinies, la récursivité et l'absence d'analyse inter-procédurale sont des facteurs qui limitent l'applicabilité de l'approche dans des contextes oú ces mécanismes sont essentiels. Toutefois, ces limitations peuvent être surmontées par des améliorations ciblées.
Les perspectives d'amélioration de cette méthode sont prometteuses. Concernant la gestion de la récursivité, une solution envisageable serait de la traiter de la même manière que les boucles, en dépliant le corps de la fonction un nombre fixe de fois, un concept déjà bien établi sous le nom de *function inlining*. Cela permettrait de rendre l'analyse des fonctions récursives plus accessible.
Pour surmonter l'absence d'analyse inter-procédurale, il serait envisageable d'exploiter une forme intermédiaire, comme LLVM-IR, qui permet une meilleure visualisation des interactions entre fonctions. Toutefois, cette solution nécessiterait le développement d'un module capable de mapper les instructions du code source à celles de LLVM-IR, afin de conserver la précision de l'analyse. De plus, pour améliorer la gestion des états *UNKNOWN*, une autre piste consisterait à combiner plusieurs solveurs SMT, puisque ces derniers partagent le même langage de spécification SMT-LIB2. Cela pourrait réduire les cas indéterminés et augmenter la robustesse des résultats.

En résumé, l'approche méthodologique proposée présente de nombreux atouts, notamment sa capacité à intégrer les comportements matériels et sa flexibilité dans l'ajout de nouveaux modules d'analyse. Bien que certaines limites existent, telles que la gestion des boucles infinies, de la récursivité et l'absence d'analyse inter-procédurale, les pistes d'amélioration et les solutions envisagées permettent d'envisager une extension de cette méthode vers des applications encore plus vastes et complexes. Cette approche offre ainsi une base solide pour améliorer la sécurité et la fiabilité des programmes, tout en étant adaptable aux besoins variés des systèmes modernes.

4

Construction et exploitation de la base de connaissances

Sommaire

4.1 Introduction

Le comportement d'un programme C dépend étroitement de son environnement d'exécution : architecture matérielle, système d'exploitation, compilateur et options de compilation influencent directement la représentation des données, la taille des types entiers et les règles de promotion arithmétique. Pour cette raison, l'analyse formelle d'un programme ne peut être pleinement fiable si elle ignore ces paramètres. Une vérification qui supposerait que le type `long` occupe toujours 64 bits produirait des résultats erronés dès lors que le programme est destiné à s'exécuter sous Windows, où ce même type n'occupe que 32 bits — ce qui modifie radicalement la frontière arithmétique à partir de laquelle un débordement est possible.

La base de connaissances développée dans notre approche répond précisément à cet enjeu. Notée $\mathscr{KD}_A$, elle permet d'intégrer explicitement l'impact de l'environnement d'exécution dans le processus d'analyse, en fournissant au moteur de vérification les tailles effectives des types entiers propres à la plateforme cible. Elle constitue ainsi le lien formel entre la description de l'environnement d'exécution et les assertions logiques que le solveur SMT doit évaluer.

Cette base de connaissances, son rôle, sa structure et sa place dans la chaîne d'analyse sont présentés en détail dans ce chapitre. Nous décrivons d'abord les trois composantes de l'environnement d'exécution — système d'exploitation, compilateur et architecture matérielle — en montrant comment chacune contribue à la variabilité du comportement arithmétique des programmes. Nous présentons ensuite la conception de $\mathscr{KD}_A$: les sources qui ont alimenté sa construction, la structuration des données qu'elle encode, et la formalisation mathématique du débordement entier qu'elle permet d'instancier. Nous décrivons enfin comment cette base est interrogée durant l'analyse et comment elle participe à la génération des assertions de sûreté soumises au solveur.

4.2 Environnement d'exécution

La largeur effective d'un type entier ne dépend pas uniquement du langage, mais de l'environnement dans lequel le programme est compilé et exécuté. Cette section identifie les trois facteurs qui composent cet environnement et qui conditionnent le choix du modèle de données : le système d'exploitation, le compilateur et ses options, et l'architecture matérielle cible.

4.2.1 Système d'exploitation

Le système d'exploitation constitue la couche de médiation entre le programme et le matériel. Il peut être défini, simplement, comme un programme chargé d'exploiter les ressources matérielles — **processeur**, **mémoire**, **stockage** et **périphériques** — afin de permettre l'exécution des programmes utilisateurs. Il joue ainsi le rôle de **chef d'orchestre** de la machine [85] : c'est lui qui attribue, contrôle et arbitre l'accès à ces ressources tout au long du cycle de vie d'un processus.

Dans notre contexte, l'influence du système d'exploitation sur le comportement des programmes se manifeste à travers deux mécanismes principaux. Le premier est l'**abstraction des ressources matérielles** : le programme n'interagit pas directement avec le processeur ou la mémoire, mais accède à des interfaces standardisées — appels système, descripteurs de fichiers, bibliothèques — qui masquent la complexité du matériel sous-jacent et méditialisent tout accès aux composants critiques. Le second est la **mémoire virtuelle et l'isolation** : chaque processus dispose d'un espace d'adressage indépendant, structuré en segments (code, données, pile, tas) et géré par la MMU (*Memory Management Unit*) [85], ce qui confine les erreurs internes au périmètre du processus.

Du point de vue de la vérification formelle, c'est la dimension *Application Binary Interface* (ABI) qui nous intéresse le plus directement. Elle spécifie les conventions d'appel, l'organisation de la mémoire, la taille des types fondamentaux C et la représentation des objets dans un exécutable [82]. Selon l'ABI retenue, un même type C peut avoir une taille différente : le type `long` occupe 8 octets sur les systèmes Linux et macOS 64 bits (modèle LP64), mais seulement 4 octets sur Windows 64 bits (modèle LLP64), et ce quelle que soit l'architecture du processeur sous-jacent [27]. Deux environnements d'exécution utilisant le même matériel peuvent ainsi produire des comportements arithmétiques différents pour un programme C identique, ce qui illustre la nécessité de disposer d'une connaissance explicite de l'ABI lors de la vérification.

Le système d'exploitation contribue enfin à la sécurité de l'exécution par plusieurs mécanismes complémentaires : l'*Address Space Layout Randomization* (ASLR) qui randomise l'agencement de l'espace mémoire, le bit NX/DEP qui empêche l'exécution de données hors de segments prévus à cet effet [85, 69, 7], les politiques de contrôle d'accès (permissions POSIX, SELinux), et la séparation stricte entre modes utilisateur et noyau. Ces protections forment une couche de défense importante, mais elles ne compensent pas les vulnérabilités arithmétiques qui surviennent avant même l'intervention de l'OS (Operating System) — c'est précisément là que notre approche intervient.

4.2.2 Compilateur et options de compilation

Le compilateur traduit le programme source en code machine exécutable. Il est le premier acteur à interpréter les types et les opérations arithmétiques du programme, et ses choix conditionnent directement les tailles des données, les règles de promotion et les comportements indéfinis.

Le standard C [47] définit des règles de conversion entre types entiers, dites *conversions arithmétiques usuelles* : lorsque deux opérandes de types différents sont combinés dans une expression, l'opérande de rang inférieur est promu au rang de l'opérande supérieur avant que l'opération ne soit effectuée. Ces règles s'appliquent uniformément, mais leur effet concret dépend des tailles effectives des types sur la plateforme — tailles que le compilateur détermine en accord avec l'ABI de l'OS cible [82]. Ainsi, une règle de promotion dont le résultat semble anodin sur une plateforme 64 bits peut déclencher un débordement silencieux sur une plateforme 32 bits, à cause d'un type `long` de largeur réduite.

Les **options de compilation** constituent un deuxième levier d'influence [82]. L'option `-m32` force le compilateur à générer un code 32 bits et à adopter le modèle ILP32, même sur un hôte 64 bits : le type `long` est alors de 32 bits, et les règles de débordement changent en conséquence. L'option `-m64` impose à l'inverse un modèle 64 bits. D'autres drapeaux tels que `-O2` ou `-fwrapv` peuvent modifier le comportement des débordements signés — par défaut défini comme comportement indéfini par le standard, mais stabilisé en sémantique de complémentation à deux par `-fwrapv`. Ces paramètres doivent être pris en compte explicitement dans l'analyse, et non inférés de l'architecture de la machine d'analyse.

Plusieurs compilateurs majeurs sont documentés dans $\mathcal{KD}_A$: GCC et Clang pour Linux et macOS, MSVC (Microsoft Visual C++) et MinGW pour Windows, et le compilateur TI C/C++ pour les microcontrôleurs TMS320C28x [87]. Chacun peut produire des tailles de types différentes même pour un même programme source compilé sur la même machine physique, selon les options fournies.

4.2.3 Architecture matérielle

L'architecture matérielle détermine la largeur native des registres du processeur, les règles d'alignement des données en mémoire et, en interaction avec le compilateur et l'OS, le modèle de données effectivement adopté. On distingue principalement les architectures 64 bits à usage général (x86_64, AArch64, RISC-V 64), les architectures 32

bits (x86, ARM 32 bits) et les architectures embarquées (AVR, MSP430, TMS320C28x) dont les conventions diffèrent sensiblement des plateformes de bureau.

La correspondance entre architecture matérielle et modèle de données n'est pas bijective. Une architecture x86_64 peut faire fonctionner un programme sous le modèle LP64 (Linux), LLP64 (Windows) ou ILP32 (avec l'option `-m32`). Réciproquement, le modèle ILP32 est partagé par les architectures 32 bits traditionnelles et par les compilations 32 bits sur hôte 64 bits [27]. Cette non-bijection est précisément la raison pour laquelle $\mathscr{KD}_A$ ne raisonne pas directement en termes d'architecture, mais introduit la notion de **modèle de données** comme couche d'abstraction intermédiaire, permettant de regrouper les environnements partageant les mêmes tailles de types.

4.3 Conception et structure de la base de connaissances

La conception de $\mathscr{KD}_A$ répond à un objectif central : permettre à l'analyse statique de modéliser précisément l'impact de l'environnement d'exécution sur le comportement réel d'un programme. Pour atteindre cet objectif, il a été nécessaire d'identifier les sources de connaissance pertinentes, de structurer les données qu'elles fournissent, et de formaliser mathématiquement les assertions de sûreté que ces données permettent d'instancier.

4.3.1 Identification et agrégation des sources de connaissance

La construction d'une base de connaissances nécessite l'identification et la sélection de sources de référence documentant à la fois les bonnes pratiques de programmation sécurisée et les caractéristiques des environnements d'exécution. Ces sources relèvent de deux catégories complémentaires.

La première catégorie regroupe les **référentiels de sécurité et de sûreté**. Parmi les plus importants figurent les CWE, qui classifie notamment sous CWE-190 les débordements entiers et sous CWE-680 les conversions entières problématiques [29] ; le **SEI CERT C Secure Coding Standard**, dont la règle INT32-C interdit les débordements sur les entiers signés [79] ; et les directives **MISRA C/C++**, dont la règle 7.1 de MISRA C:2025 [70] encadre l'utilisation des expressions entières. Ces corpus constituent des fondations normatives largement adoptées dans l'industrie et la recherche, et ont guidé l'identification des cas de débordement à couvrir en priorité. Malgré leur documentation exhaustive, ces failles demeurent fréquentes dans les bases de code — y compris

dans des dépôts accessibles publiquement — ce qui souligne la nécessité de relier les connaissances théoriques issues de ces référentiels aux contextes d'exécution réels.

La seconde catégorie regroupe les **sources techniques sur les environnements d'exécution** : standards ISO (*International Organization for Standardization*) C [47] et C99 [46], spécifications ABI (System V AMD64 ABI, Windows x64 ABI), documentations des compilateurs GCC, Clang, MSVC et TI [86], et guides de référence des architectures cibles. Ces sources formelles ont été complétées par des **tests empiriques** conduits à l'aide de l'explorateur de compilateurs Godbolt[1], un outil en ligne open source[2], qui donne accès à plus de 200 combinaisons architecture/OS /compilateur et a permis de vérifier systématiquement les tailles de types dans chaque configuration.

L'un des facteurs techniques à l'origine des vulnérabilités documentées par ces référentiels réside précisément dans la dépendance des types entiers primitifs à l'environnement d'exécution. Le type `long`, par exemple, occupe 4 octets sur Windows 64 bits et 8 octets sur Linux 64 bits. Cette variabilité peut provoquer des erreurs lors du portage de code ou fausser les vérifications statiques qui supposeraient une taille fixe. Pour pallier cette non-portabilité, les référentiels de bonnes pratiques préconisent l'usage de types entiers à taille fixe (`int32_t`, `uint64_t`, etc.) définis dans `<stdint.h>` ; mais dans les bases de code existantes, les types natifs restent omniprésents, ce qui renforce la nécessité de les prendre en compte dans l'analyse.

4.3.2 Structuration des données d'environnement d'exécution

À partir de ces sources, nous avons identifié deux grandes catégories d'informations relatives à l'environnement d'exécution. La première regroupe les éléments qui influencent l'**allocation ou la représentation en mémoire** : taille des types entiers, règles d'alignement, modèle de pointeurs, représentation des entiers signés. La seconde concerne les facteurs susceptibles d'affecter le **comportement temporel** : certains modèles d'architecture, des optimisations agressives du compilateur ou des comportements dépendant du matériel. Notre analyse se concentre principalement sur la première catégorie, car c'est elle qui conditionne directement la détection de débordements entiers.

Le Tableau 4.1 présente un extrait de notre base de données, recueillant les tailles des types entiers fondamentaux du C collectées dans différents environnements d'exécution. Il illustre la diversité des configurations couvertes : l'architecture X86 et son extension

[1] https://godbolt.org/
[2] https://github.com/compiler-explorer/compiler-explorer

X86_64 (processeurs Intel et AMD (Advanced Micro Devices)), ainsi que le processeur embarqué TMS320C28x de Texas Instruments [86]. La colonne *Options de compilation* met en évidence l'effet de l'option `-m32` sur le type `long` : même sur hôte 64 bits, cette option bascule vers le modèle ILP32 et réduit sa largeur à 32 bits.

Table 4.1: Taille du type `integer` selon les différents environnements d'exécution

Integer type	Target Arch.	OS ; Compiler	Compiler options	Size *bits*
char	X86 / X86_64 TMS320C28x	OS 32/64 ; C/C++ Comp. Compiler v22.6.0.LTS	$flags$	8 16
short	X86 / X86_64 TMS320C28x	OS 32/64 ; C/C++ Comp. Compiler v22.6.0.LTS	$flags$	16
int	X86 / X86_64 TMS320C28x	OS 32/64 ; C/C++ Comp. Compiler v22.6.0.LTS	$flags$	32 16
long	TMS320C28x	Compiler v22.6.0.LTS	$flags$	32
	X86 / X86_64	Windows ; MSVC Linux ; Clang/GCC macOS ; Clang	$m32 \in flags$	
	X86_64	Linux ; Clang/GCC macOS ; Clang	$m32 \notin flags$	64
long long	X86 / X86_64 TMS320C28x	OS 32/64 ; C/C++ Comp. Compiler v22.6.0.LT S	$flags$	64
int*N*_t	Supp. targets	Supp. OS ; Supp. Compilers	$flags$	*N*

L'observation centrale que ce tableau met en évidence est la suivante : la même architecture physique (X86_64) peut produire des tailles de type radicalement différentes selon le système d'exploitation et les options de compilation. Ce constat justifie que la base de connaissances ne soit pas indexée par l'architecture seule, mais par une combinaison de paramètres incluant l'OS, le compilateur et les options — c'est-à-dire par l'environnement d'exécution complet.

4.3.3 Modélisation formelle des propriétés

Comme introduit dans la Section 3.4.2, le phénomène de débordement peut être modélisé par une relation entre la valeur mathématique d'un calcul et la valeur effectivement stockée en mémoire. Nous proposons ici une formalisation rigoureuse de ce modèle, adaptée à la génération de contraintes logiques pour les solveurs SMT.

Disposer des tailles de types est une condition nécessaire mais non suffisante pour la vérification formelle. Il faut encore traduire ces tailles en assertions logiques que le solveur SMT peut évaluer. Cette traduction repose sur une formalisation mathématique du phénomène de débordement entier, que nous développons ici.

Presque tous les processeurs modernes utilisent la représentation en **complément à deux** pour les entiers signés, spécifiquement conçue pour que le débordement produise un *wrap-around*. Dans cette représentation sur n bits, un entier appartient à l'intervalle $[-2^{n-1},\ 2^{n-1}-1]$ et chaque valeur de cet intervalle possède une représentation unique. Si une opération arithmétique produit un résultat x supérieur à $2^{n-1}-1$, un débordement survient : seuls les n bits de poids faible sont conservés, et la valeur observable dans la variable est le résidu de la division euclidienne de x par 2^n.

Pour formaliser cette relation, notons $\bar{x}$ la valeur effectivement stockée après débordement éventuel. Nous reprenons l'égalité fondamentale introduite précédemment et en donnons ici une interprétation formelle complète :

$$\boldsymbol{x = 2^n \times k + \bar{x}} \tag{4.1}$$

où k est un entier naturel et $n \in \{8, 16, 32, 64\}$ est la largeur en bits du type, déterminée par l'environnement d'exécution via $\mathcal{KD}_A$. Lorsque $k = 0$, le résultat x est entièrement représentable dans n bits : il n'y a pas de débordement et $\bar{x} = x$. Lorsque $k \neq 0$, la valeur stockée $\bar{x} = x - 2^n \cdot k$ est différente du résultat mathématique exact : le débordement s'est produit et le programme produit une valeur potentiellement erronée.

Cette formule paramétrisée se décline en deux assertions logiques selon le contexte de l'opération, en s'appuyant sur les règles de conversion du standard C [47].

Assertion pour une opération arithmétique pure. Soit une opération binaire $s_1 \diamond s_2$ sur deux entiers signés, et v la valeur de son résultat, $v = [\![s_1 \diamond s_2]\!]$. Par la règle des *conversions arithmétiques usuelles*, l'opérande de rang inférieur est promu au rang supérieur avant l'opération. La largeur effective du résultat est donc celle du type de rang le plus élevé parmi les deux opérandes. La valeur v satisfait :

$$\boldsymbol{(v = 2^n \times k + \bar{v}) \wedge (k \neq 0)} \tag{4.2}$$

où $n = \lambda p.\ \texttt{size\_max\_rank}(\texttt{type}(s_1),\ \texttt{type}(s_2),\ \diamond,\ p)$ et p désigne l'ensemble des paramètres de la plateforme cible interrogée dans $\mathcal{KD}_A$.

Assertion pour une opération arithmétique avec affectation. Soit une instruction de mise à jour $x = \mathit{exp}$ et $v = [\![\mathit{exp}]\!]$ la valeur de l'expression de droite. Par la règle de conversion à l'affectation du standard C, si le rang de v est supérieur à celui de x, la valeur est tronquée au rang de x. La largeur déterminante est donc le minimum des rangs du résultat et de la variable cible. La variable x satisfait :

$$(\boldsymbol{x} = \boldsymbol{2}^{\boldsymbol{n}} \times \boldsymbol{k} + \bar{\boldsymbol{x}}) \wedge (\boldsymbol{k} \neq \boldsymbol{0}) \tag{4.3}$$

où $n = \lambda\, p.\ \texttt{size\_min\_rank}(\texttt{type}(x),\ \texttt{type}(v),\ p)$. Il est important de noter que dans ce second cas, n est calculé à partir des types à la fois de l'expression résultante et de la variable assignée — ce qui reflète fidèlement la règle sémantique selon laquelle *si l'opérande de droite est de rang inférieur, il sera promu au rang de l'opérande de gauche*.

Ces deux assertions forment la base sur laquelle sont construites les vérifications de débordement dans notre approche. Pour tester si un débordement est possible, il suffit de soumettre au solveur SMT la satisfiabilité de l'une de ces formules avec $k \neq 0$: une réponse `sat` atteste qu'il existe une valuation des entrées pour laquelle le débordement se produit effectivement.

4.3.4 Choix de la théorie SMT : arithmétique entière non linéaire

Le choix de la théorie logique soumise au solveur SMT est une décision qui conditionne à la fois la correction et la scalabilité de l'analyse. Deux théories se présentent naturellement pour raisonner sur les débordements entiers : la théorie des **vecteurs de bits** Bit-Vector (BV) et l'**arithmétique entière non linéaire** Quantifier-Free Non-Linear Integer Arithmetic (QF_NIA) [52].

La théorie BV est la plus naturelle pour modéliser les opérations machine [45] : elle représente les entiers comme des chaînes de bits de largeur fixe et capture nativement le comportement wrap-around. Cependant, les solveurs BV reposent généralement sur la technique du **bit-blasting** [51], qui consiste à décomposer chaque opération sur n bits en un ensemble de clauses booléennes. Cette réduction entraîne une croissance *exponentielle* du nombre de variables et de clauses avec la largeur n, ce qui dégrade très rapidement les performances pour des largeurs de 64 ou 128 bits et rend la théorie BV peu adaptée aux problèmes impliquant des multiplications de grandes valeurs.

La théorie QF_NIA, en revanche, traite les entiers comme mathématiquement non bornés [16]. Son avantage est une complexité quasi-indépendante de la largeur de bit : les performances ne se dégradent pas lorsque n augmente. Son inconvénient

apparent est précisément cette absence de bornes : un solveur entier standard raisonne sur les entiers de $\mathbb{Z}$ et ne modélise pas le wrap-around machine. La formule (4.1) est exactement ce qui comble cette lacune. En introduisant la variable fraîche k et la relation $x = 2^n \times k + \bar{x}$, on réintroduit explicitement dans le domaine entier la sémantique machine du débordement, sans recourir à la représentation bit-à-bit.

Il est important de noter que cette formule contient un produit entre deux variables symboliques, $2^n \times k$, ce qui en fait une formule **non linéaire**. Le terme 2^n est certes une constante une fois la plateforme résolue depuis $\mathcal{KD}_A$, mais k est une variable fraîche introduite pour chaque opération : le produit reste non linéaire au sens de QF_NIA. C'est précisément la raison pour laquelle la logique QF_NIA est sélectionnée dans le script SMT généré par l'outil, et non une logique linéaire telle que Quantifier-Free Linear Integer Arithmetic (QF_LIA).

Pour illustrer concrètement la différence entre ces approches, considérons un problème simple : trouver deux entiers $a > 1$ et $b > 1$, différents de 13, dont le produit vaut 13. Sur les entiers mathématiques $\mathbb{Z}$, ce problème est *insatisfaisable* : 13 est un nombre premier et ne possède aucune décomposition non triviale. Un solveur entier standard retourne immédiatement `unsat`.

Cependant, en arithmétique machine à n bits, le wrap-around permet l'existence de solutions : il suffit que $a \times b \equiv 13 \pmod{2^n}$ pour que le produit stocké soit 13, même si le résultat mathématique exact ne l'est pas. En encodant ce comportement avec notre formule — c'est-à-dire en cherchant a, b, k et $\bar{x}$ tels que $a \times b = 2^n \times k + \bar{x}$ et $\bar{x} = 13$ — le solveur Z3 sous QF_NIA trouve des solutions satisfaisables en un temps quasi-constant, quel que soit n.

Le Tableau 4.2 quantifie cet avantage. L'encodage entier étendu maintient un temps de résolution quasi-constant à travers les largeurs de 32, 64 et 128 bits (entre 0,10 s et 0,12 s sous Z3), tandis que la théorie BV de Z3 voit son temps multiplié par 32 entre 32 et 128 bits (de 0,167 s à 3,536 s). CVC5 se comporte mieux en BV sur les petites largeurs, mais se dégrade également à 128 bits (0,445 s). Ces résultats ont été obtenus sous Ubuntu 24.04 LTS sur un processeur Ryzen 5 3600X avec 8 Go de RAM [77].

Cette observation est corroborée par la comparaison avec des outils tiers. L'outil **ESBMC** [40], reconnu pour ses capacités d'analyse de débordements entiers via la théorie BV, parvient à détecter des vulnérabilités de ce type lorsqu'il utilise son solveur BV, mais échoue avec son solveur entier. La raison est précisément l'absence d'un encodage explicite du wrap-around : sans la formule (4.1), le solveur entier traite

Table 4.2: Comparaison des temps de résolution entre la théorie BV et l'encodage entier étendu QF_NIA

Largeur	Encodage	Résultat	Temps
Non borné	Entier standard	`unsat`	$<$ 0,01 s
32 bits	Entier étendu (Z3)	`sat`	0,111 s
32 bits	Vecteur de bits (Z3)	`sat`	0,167 s
32 bits	Vecteur de bits (CVC5)	`sat`	0,080 s
64 bits	**Entier étendu (Z3)**	`sat`	**0,112 s**
64 bits	Vecteur de bits (Z3)	`sat`	0,627 s
64 bits	Vecteur de bits (CVC5)	`sat`	0,140 s
128 bits	**Entier étendu (Z3)**	`sat`	**0,102 s**
128 bits	Vecteur de bits (Z3)	`sat`	3,536 s
128 bits	Vecteur de bits (CVC5)	`sat`	0,445 s

toute expression arithmétique $a \times b$ comme un produit d'entiers mathématiques non bornés et ne perçoit pas le débordement. Notre encodage représente donc une extension substantielle du pouvoir expressif du domaine entier pour la détection de vulnérabilités arithmétiques.

En résumé, la formule (4.1) réalise simultanément deux objectifs : elle encode fidèlement la sémantique machine du wrap-around dans un domaine entier non borné, et elle positionne le problème dans QF_NIA plutôt qu'en BV, ouvrant la voie à une vérification scalable pour les largeurs de types élevées. C'est cette formulation qui justifie le choix de la logique QF_NIA dans le script SMT généré par notre outil, dont l'implémentation est décrite au chapitre suivant.

4.4 Exploitation de la base de connaissances

Une fois $\mathcal{KD}_A$ construite, il reste à définir comment elle est interrogée pendant l'analyse et comment ses résultats sont traduits en contraintes formelles. Cette section décrit d'abord le mécanisme d'interaction avec la base — format de stockage et chaîne de résolution — puis la façon dont les largeurs de types obtenues alimentent la génération des assertions de sûreté.

4.4.1 Interaction avec la base de connaissances

Format de stockage

La base de connaissances $\mathcal{KD}_A$ est représentée sous forme d'un fichier JSON (JavaScript Object Notation) structuré en deux niveaux. Le premier niveau définit les **modèles de données** : chaque modèle associe à chaque type entier C fondamental sa largeur en bits. Le second niveau définit les **règles de sélection** : chaque règle spécifie un ensemble de conditions portant sur l'architecture, l'OS, le compilateur, les drapeaux de compilation ou le triplet de compilation, et pointe vers le modèle de données à adopter lorsque ces conditions sont satisfaites. Le Listing 4.1 illustre un extrait de cette structure.

Listing 4.1: Extrait de `KnowledgeBase.json` : modèles de données et règles

```
{
  "dataModels": {
    "LP64":  { "sizes": {"char":8,"short":16,"int":32,"long":64,"longlong":64} },
    "LLP64": { "sizes": {"char":8,"short":16,"int":32,"long":32,"longlong":64} },
    "ILP32": { "sizes": {"char":8,"short":16,"int":32,"long":32,"longlong":64} },
    "ILP16": { "sizes": {"char":8,"short":16,"int":16,"long":32,"longlong":64} }
  },
  "rules": [
    { "flags":  ["-m32"],              "dataModel": "ILP32" },
    { "os":     ["windows"],           "dataModel": "LLP64" },
    { "arch":   ["x86_64","aarch64"],
      "os":     ["linux","macos"],     "dataModel": "LP64"  },
    { "arch":   ["avr","msp430"],      "dataModel": "ILP16" }
  ]
}
```

Ce format présente deux avantages essentiels. D'abord, la séparation entre modèles de données et règles de sélection évite toute redondance : la définition du modèle LLP64 est écrite une seule fois et peut être référencée par autant de règles que nécessaire. Ensuite, la structure JSON permet d'étendre la base à de nouveaux environnements sans recompiler l'outil : il suffit d'ajouter une nouvelle entrée dans le fichier de configuration.

Interrogation de la base de connaissances

Lors du parcours de l'AST, chaque expression arithmétique est analysée pour déterminer son type. Dans le cas d'une affectation telle que `long int uid_sid = (long)userId * serviceId`, le type de la variable cible est `long` et celui de l'expression de droite est également `long`. Le système construit alors une requête en utilisant le contexte

d'exécution courant comme ensemble de filtres : les règles de $\mathcal{KD}_A$ sont évaluées dans l'ordre, et la première dont les conditions correspondent à l'environnement fourni désigne le modèle de données à appliquer. Une fois le modèle sélectionné, la taille en bits du type est extraite et transmise à la fonction de génération d'assertions. Cette taille n, obtenue depuis $\mathcal{KD}_A$, est précisément le paramètre qui apparaît dans les formules (4.2) et (4.3).

4.4.2 Génération des assertions de sûreté et utilisation dans l'approche

La génération concrète des assertions de sûreté constitue l'aboutissement du processus décrit dans ce chapitre. Une fois la largeur n obtenue depuis $\mathcal{KD}_A$ pour une opération donnée, l'outil introduit deux variables fraîches — k (quotient de débordement) et $\bar{x}$ (*xbar*, résidu stocké) — et construit la contrainte SMT correspondant à la formule (4.2) ou (4.3).

Dans la syntaxe Z3 utilisée par notre outil, cette assertion prend la forme d'une expression conditionnelle `If` : si la condition de débordement est satisfaisable (c'est-à-dire si l'opération exacte $s_1 \diamond s_2$ peut s'écrire $2^n \cdot k + \bar{x}$ avec $k > 0$ pour des valeurs d'entrée licites), la variable prend la valeur du résidu $\bar{x}$; sinon elle prend la valeur exacte. Cette formulation permet au solveur de *choisir librement* entre les deux sémantiques et d'explorer les deux branches — débordement ou non — dans un seul appel au solveur.

Pour illustrer concrètement l'utilisation de $\mathcal{KD}_A$ dans l'approche, reprenons l'exemple de la multiplication `(long)userId * serviceId` compilée pour une plateforme Windows x86. L'environnement d'exécution spécifié conduit $\mathcal{KD}_A$ à sélectionner le modèle LLP64, pour lequel `size(long) = 32`. L'assertion de sûreté générée est alors :

$$(\texttt{uid\_sid} = 2^{32} \times k + \bar{x}) \wedge (k \neq 0)$$

avec $n = 32$ fourni directement par $\mathcal{KD}_A$. Si la même analyse avait été conduite avec le modèle LP64 (Linux 64 bits, `size(long) = 64`), l'assertion aurait utilisé 2^{64} comme modulus — un paramètre radicalement différent qui modifie l'espace de solutions du solveur. C'est précisément ce risque de faux négatif architectural que $\mathcal{KD}_A$ permet d'éliminer, en garantissant que les assertions de sûreté sont toujours instanciées avec les paramètres réels de la plateforme cible. La résolution complète de cet exemple — génération du script, exécution du solveur et interprétation du contre-exemple — est présentée au Chapitre 5, Section 5.4.

4.5 Conclusion

Ce chapitre a présenté la base de connaissances architecturales $\mathcal{KD}_A$, élément central de l'approche de vérification formelle développée dans cette thèse. Nous avons montré que l'environnement d'exécution — système d'exploitation, compilateur et architecture matérielle — introduit une variabilité substantielle dans le comportement arithmétique des programmes C, et que cette variabilité ne peut être ignorée sans risquer de produire des analyses incorrectes.

La formalisation proposée, fondée sur la relation $x = 2^n \times k + \bar{x}$ et ses deux instanciations selon le contexte de l'opération, fournit un cadre logique rigoureux pour rendre le débordement observable par un solveur SMT. Les assertions (4.2) et (4.3) constituent le pont formel entre la description de l'environnement et les contraintes soumises au solveur, avec une précision directement conditionnée par le modèle de données résolu depuis $\mathcal{KD}_A$.

La structure de $\mathcal{KD}_A$ en modèles de données et règles de sélection est conçue pour être à la fois précise et extensible : la couverture de nouveaux compilateurs, architectures ou options de compilation ne nécessite que l'enrichissement du fichier JSON, sans modification du moteur d'analyse. Le chapitre suivant décrit comment ce mécanisme est intégré concrètement dans notre prototype, en montrant les fonctions `resolveDataModel` et `getBitWidthForType` qui implémentent la chaîne de résolution depuis l'environnement d'exécution jusqu'aux contraintes SMT générées.

5

Conception et implémentation de l'outil de vérification *AETHER*

Sommaire

5.1 Introduction

Ce chapitre présente l'implémentation concrète de l'outil développé dans le cadre de cette thèse. L'objectif principal est de réaliser une **analyse statique de programmes `C/C++`** et de **vérifier formellement** leur conformité vis-à-vis de propriétés de sûreté et de sécurité définies par l'utilisateur. Ces propriétés sont évaluées dans un **contexte d'exécution précis**, intégrant à la fois des contraintes matérielles (architecture cible, capacités mémoire) et logicielles (bibliothèques externes, appels système).

L'outil `Architecture-aware Execution and THEoretical Reasoning (AETHER)`, publiquement accessible à l'adresse `https://github.com/NazimKYS/AETHER`, s'inscrit dans cette perspective. Il repose sur l'infrastructure `Clang/LLVM` pour analyser statiquement le programme, construire une représentation en forme (SSA) sensible au flot de contrôle, puis générer un script Python/Z3. La résolution de ce dernier permet soit de démontrer l'inatteignabilité d'un point d'analyse, soit de produire un contre-exemple concret. Par ailleurs, les tailles des types entiers sont résolues dynamiquement à l'aide d'une base de connaissances architecturales $\mathcal{KD}_A$, rendant ainsi l'analyse dépendante de l'architecture cible plutôt que de la seule plateforme d'analyse.

Ce chapitre décrit l'implémentation de ce pipeline en s'appuyant directement sur le code source de l'outil. La Section 5.2 présente l'architecture globale ainsi que l'interface de configuration. La Section 5.3 détaille les différentes phases du processus d'analyse ainsi que les mécanismes de traduction en contraintes SMT utilisés pour la vérification formelle. La Section 5.4 illustre le fonctionnement de l'outil à travers un cas d'étude réaliste mettant en évidence ses capacités de détection de vulnérabilités. Enfin, les Sections 5.5 et 5.6 discutent les limites du prototype ainsi que ses principales contributions.

5.2 Architecture et vue d'ensemble de l'outil

Pipeline global

La Figure 5.1 représente l'enchaînement des modules qui composent l'outil, depuis les fichiers d'entrée jusqu'au résultat produit par le solveur.

Le point d'entrée est constitué de trois ressources. Le **programme source C** est le code à analyser. Le fichier **`target.json`** spécifie le point d'analyse, les contraintes utilisateur et l'environnement d'exécution visé. La **base de connaissances** `KnowledgeBase.json` encode les correspondances entre environnements d'exécution et modèles de données

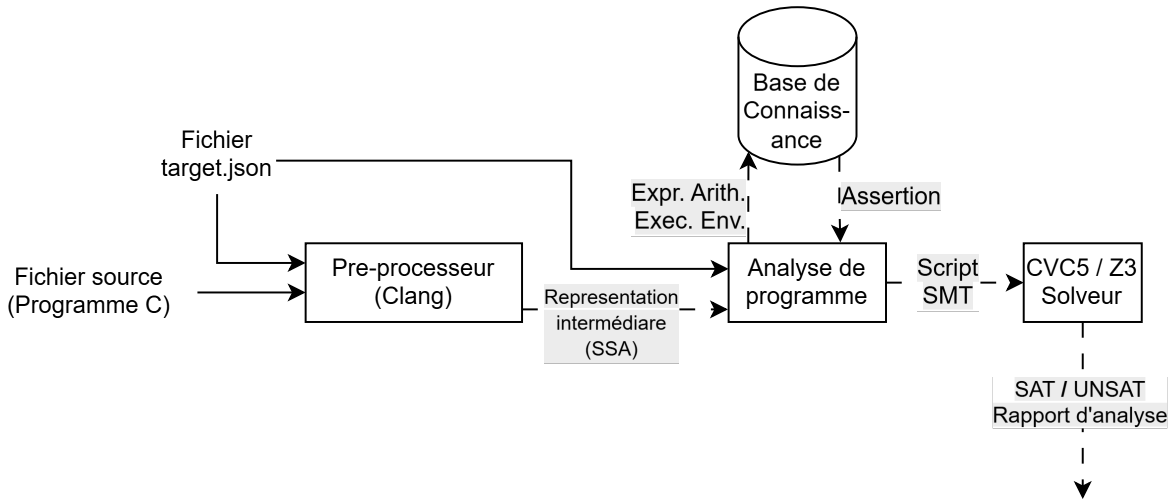

Figure 5.1: Vue d'ensemble du pipeline de l'outil AETHER

(tailles effectives des types entiers). Une fois ces ressources chargées par le module d'entrée (`main.cpp`), le pipeline enchaîne quatre phases : localisation de l'instruction cible dans l'AST, étiquetage SSA du corps de la fonction englobante, calcul de la condition de chemin, et génération du script Python/Z3. Ce script est finalement exécuté par Z3 via `python3 checking.py`.

Sur le plan technologique, deux choix structurent ce pipeline. L'infrastructure `Clang/LLVM`, via sa bibliothèque `libTooling`, fournit un accès programmatique à l'AST avec informations de types, de portée et de localisation source. Z3 est invoqué via son interface Python plutôt qu'en générant directement du code `SMT-LIB2`.

Ce choix est avant tout une question de **commodité syntaxique** : l'interface Python de Z3 permet d'exprimer des contraintes de façon plus lisible et de les construire programmatiquement avec moins de cérémonial que le format `SMT-LIB2`. Il ouvre également la possibilité d'**énumérer l'ensemble des solutions** du problème encodé : en ajoutant après chaque modèle trouvé une contrainte d'exclusion `s.add(Or(v != val_v, ...))` puis en réinvoquant `s.check()`, on peut itérer sur tous les contre-exemples dans une simple boucle Python — ce qui serait plus lourd à implémenter directement en `SMT-LIB2`. Il convient toutefois de souligner qu'il n'existe aucun obstacle technique à remplacer la génération Python par une génération directe de code `SMT-LIB2` : les contraintes encodées sont exactement les mêmes dans les deux cas, et tout solveur compatible `SMT-LIB2` pourrait alors être substitué à Z3.

Interface de configuration

Le fichier `target.json` (Listing 5.1) constitue l'interface entre l'utilisateur et le moteur d'analyse.

Listing 5.1: Fichier de configuration `target.json`

```
*{
  "target": 39,
  "constraints": [
    { "variable": "x", "operator": ">=", "value": 0 },
    { "variable": "n", "operator": "<=", "value": 1000 }
  ],
  "executionEnv": {
    "Arch":     "X86",
    "OS":       "Linux_32",
    "compiler": "clang-18",
    "flags":    ["-m32"]
  }
}
```

Le champ `target` désigne l'instruction à analyser par son numéro de ligne. Il existe plusieurs manières d'identifier une instruction cible à partir du code source ; nous avons volontairement retenu une approche simple et pragmatique, reposant sur l'hypothèse qu'**une seule instruction est présente par ligne de code**. Bien que cette méthode présente certaines limitations — notamment le fait que plusieurs instructions puissent apparaître sur une même ligne — elle permet de faciliter la mise en œuvre initiale de l'analyse et pourra être levée dans des extensions futures du prototype. Les contraintes du champ `constraints`, exprimées en triplets `(variable, opérateur, valeur)`, transcrivent les hypothèses du développeur ; elles seront injectées dans le script Python/Z3 comme assertions `s.add(...)` afin de restreindre l'espace de recherche aux entrées jugées légitimes. L'objet `executionEnv` décrit la plateforme cible ; ses paramètres conditionnent la résolution des tailles de types entiers via $\mathcal{KD}_A$.

Cette résolution s'effectue par deux fonctions complémentaires implémentées dans `main.cpp`. La fonction `resolveDataModel(env)` évalue dans l'ordre les règles de $\mathcal{KD}_A$ et retourne le premier modèle de données — LP64, LLP64, ILP32 ou ILP16 — dont les conditions correspondent à l'environnement fourni. La fonction `getBitWidthForType(type, env)` interroge ensuite ce modèle pour obtenir la largeur en bits d'un type donné ; en l'absence de correspondance, elle délègue à `ASTContext::getTypeSize()`, assurant

une dégradation gracieuse vers les valeurs par défaut de Clang.

5.3 Processus d'analyse

Une fois la configuration chargée, Clang construit l'AST complet du fichier source et la classe `ConditionPathAstConsumer` orchestre les phases décrites ci-après. Les classes et fonctions mentionnées correspondent directement aux entités du code source.

5.3.1 Localisation de l'instruction cible et identification de la fonction englobante

Cette section détaille la stratégie adoptée pour identifier l'instruction cible au niveau de l'AST. **Comment retrouver l'instruction cible au niveau de l'AST ?**
Le numéro de ligne fourni dans `target.json` est exploité par le visiteur `ConditionPathAstVisitor`, dérivé de `RecursiveASTVisitor`, qui parcourt récursivement l'AST et identifie le nœud situé à cette ligne.

La Figure 5.2 présente un schéma simplifié d'un AST généré par Clang pour un programme comportant plusieurs fonctions. À la racine, le nœud `TranslationUnitDecl` représente l'unité de traduction complète correspondant au fichier source analysé.

Chaque `FunctionDecl` encapsule sa signature (paramètres représentés par des nœuds `ParmVarDecl`) ainsi que son corps, modélisé par un nœud `CompoundStmt`. Ce dernier contient les différentes instructions de la fonction sous forme de sous-nœuds hiérarchisés. Les structures conditionnelles sont représentées par des nœuds `IfStmt`, eux-mêmes composés d'une condition (souvent un `BinaryOperator`) qui peut avoir d'autres noeud fils `BinaryOperator` et de deux blocs distincts (`CompoundStmt`) correspondant aux branches `then` et `else`. Les expressions internes, telles que les références à des variables (`DeclRefExpr`) ou les constantes (`IntegerLiteral`), apparaissent comme feuilles de l'arbre.

Cette organisation hiérarchique est essentielle pour notre approche. Elle permet, dans un premier temps, de localiser une instruction cible lors du parcours descendant de l'arbre, puis, dans un second temps, de remonter vers ses ancêtres afin d'identifier la fonction englobante ainsi que les structures de contrôle qui déterminent son contexte d'exécution. La figure illustre ainsi visuellement le double mouvement exploité par notre outil : une exploration descendante pour la localisation, suivie d'une navigation ascendante pour la reconstruction du contexte.

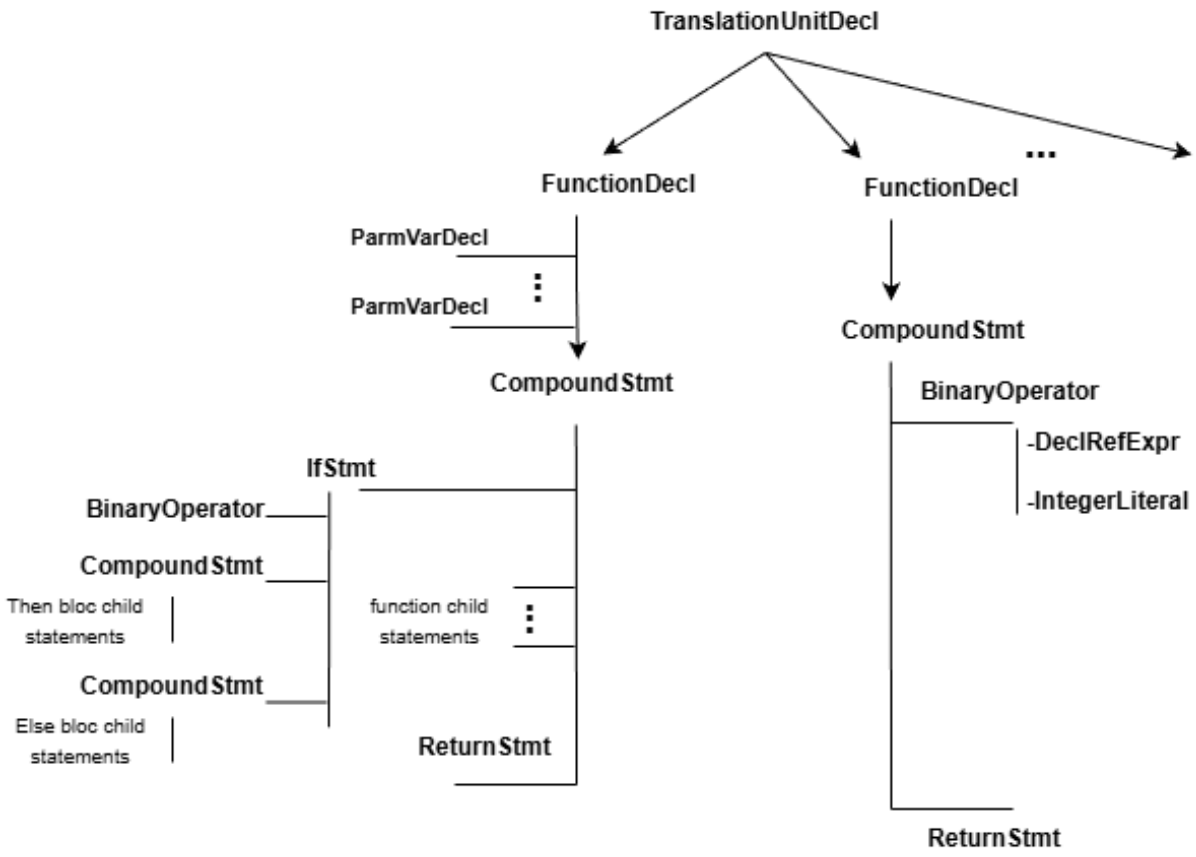

Figure 5.2: Schéma simplifié d'un AST généré par Clang

Le Listing 5.2 illustre le mécanisme adopté pour localiser l'instruction cible au niveau de l'AST.

Listing 5.2: Traversée de l'AST pour l'identification de l'instruction cible

```
bool VisitStmt(Stmt *currentStmt) {
    FullSourceLoc loc = astContext.getFullLoc(currentStmt->getBeginLoc());

    if (astContext.getSourceManager().isInMainFile(loc) &&
        loc.getSpellingLineNumber() == sourceLineOfTargetStmt) {

        if (isa<BinaryOperator>(currentStmt) ||
            isa<CallExpr>(currentStmt)       ||
            isa<DeclStmt>(currentStmt)       ||
            isa<IfStmt>(currentStmt)         ||
            isa<CompoundStmt>(currentStmt)   ||
            isa<ReturnStmt>(currentStmt)) {

            targetPoint = TargetProgramPoint(currentStmt, astContext);
            llvm::outs() << "TARGET FOUND: " << currentStmt->getStmtClassName()
                         << " at line " << sourceLineOfTargetStmt << "\n";
```

```
17            }
18        }
19        return true;
20    }
```

La méthode `VisitStmt` est invoquée lors du parcours récursif descendant de l'arbre syntaxique. À chaque visite d'un nœud `Stmt`, la position source correspondante est récupérée à l'aide de `getFullLoc(currentStmt->getBeginLoc())`. Cette étape permet d'associer chaque nœud de l'AST à sa localisation exacte dans le fichier source.
Un premier filtrage est effectué afin de s'assurer que l'instruction analysée appartient bien au fichier principal (`isInMainFile`). Cela permet d'exclure les instructions issues de fichiers d'en-tête ou générées implicitement par le compilateur, garantissant ainsi que seule la portion de code utilisateur est considérée.

La condition centrale repose sur la comparaison entre le numéro de ligne du nœud courant (`getSpellingLineNumber()`) et le numéro de ligne fourni par l'utilisatur (`sourceLineOfTargetStmt`). Cette comparaison constitue le critère de sélection de l'instruction cible. Conformément à l'hypothèse retenue dans notre prototype — une seule instruction par ligne — cette correspondance permet d'identifier de manière déterministe le point d'analyse.

Un second filtrage typologique est ensuite appliqué via les prédicats `isa<>`. Cette étape vise à exclure les sous-expressions internes (par exemple les opérandes d'une expression binaire) afin de ne retenir que des instructions complètes et sémantiquement pertinentes : affectations (`BinaryOperator`), appels de fonction (`CallExpr`), déclarations (`DeclStmt`), structures conditionnelles (`IfStmt`), blocs (`CompoundStmt`) ou instructions de retour (`ReturnStmt`).

Lorsque ces conditions sont satisfaites, le nœud courant est enregistré comme instruction cible (`globalScopeTargetStmt`). Un objet représentant le point de programme est également instancié (`TargetProgramPoint`), ce qui permettra d'initier les phases ultérieures d'analyse, notamment l'identification de la fonction englobante et le calcul de la condition de chemin.

Ce mécanisme, bien que volontairement simple, fournit une base fonctionnelle robuste pour amorcer le processus d'analyse formelle. Les limitations inhérentes à l'approche (multiples instructions sur une même ligne, macros, expansions préprocesseur) pourront être levées dans des extensions futures du prototype.

Identification de la fonction englobante Une fois l'instruction cible localisée, un parcours ascendant de l'AST est effectué afin d'identifier la fonction englobante dans lequel elle est définie.

Ce parcours consiste à remonter successivement les nœuds parents jusqu'à atteindre un nœud de type `FunctionDecl`. Ce dernier correspond au noeud de la fonction englobante.
L'extraction de cette fonction permet d'obtenir : **sa signature**, **la liste de ses paramètres** et **son corps (bloc principal)**.
Dans la version actuelle de l'outil, l'analyse est strictement intra-procédurale : seules les instructions situées dans la fonction englobante sont prises en compte. Les appels inter-procéduraux ne sont pas inlinés ; la valeur de retour d'un appel est modélisée par une variable symbolique fraîche typée (`SymbInt32`, `SymbInt64`, etc.).
Les paramètres de la fonction sont modélisés comme variables symboliques libres dans la phase de génération du script Python/Z3, avec des contraintes de bornes dépendant de leur type.

Listing 5.3: Remontée ascendante de l'AST pour l'identification de la fonction englobante

```
void findParentStmt(const Stmt *s) {
    if (!s) return;
    const auto &parents = globalAstContext->getParents(*s);
    if (parents.empty()) return;
    const auto &firstParent = *parents.begin();

    if (const FunctionDecl *funcDecl = firstParent.get<FunctionDecl>()) {
        parentFunctionOfProgramPoint = funcDecl;
        return;
    }
    if (const Stmt *parentStmt = firstParent.get<Stmt>()) {
        findParentStmt(parentStmt);
    }
}
```

5.3.2 Parcours de la fonction et étiquetage SSA

Pourquoi adopter une forme SSA sensible au chemin ?

Le corps de la fonction englobante est parcouru séquentiellement par la méthode récursive `visitAllStmts`, depuis le point d'entrée jusqu'à l'instruction cible. Deux structures sont maintenues en parallèle tout au long de ce parcours.

La première est la **table des définitions SSA** `varDefs`, une `unordered_map` qui associe

à chaque **identifiant d'une variable SSA** (format `base_version`, par exemple `x_1`) un objet `DefinitionInfo`. Une variable SSA correspond à une définition unique d'une variable du programme et est représentée par un identifiant versionné. La seconde est la **pile des conditions actives** `globalPathConditions`, une `std::stack<PathCondition>` qui mémorise les gardes conditionnels ouverts à l'instant courant du parcours.

Les **paramètres de la fonction** constituent les premières entrées de `varDefs`. Chaque paramètre du programme est représenté par une première variable SSA, identifiée par la version 0, et est modélisé comme une variable symbolique libre. Pour chaque **déclaration** `DeclStmt` rencontrée (par exemple `int x = 0;`), une première variable SSA, représentée par l'identifiant `x_0`, est créée pour la variable du programme correspondante avec l'expression initiale résolue. Pour chaque **affectation** `BinaryOperator`, la méthode `VisitBinaryOperator` récupère le dernier identifiant de variable SSA associé à la variable cible, incrémente son compteur, puis crée une nouvelle variable SSA, représentée par un nouvel identifiant, qui est enregistrée dans `varDefs`.

Construction des définitions SSA

L'Algorithme 3 formalise le processus de construction de la représentation SSA jusqu'à l'instruction cible.

Chaque affectation rencontrée entraîne la création d'une nouvelle **variable SSA** correspondant à la variable du programme définie par cette affectation. Chaque variable SSA est représentée par un identifiant unique. Ainsi, pour une variable du programme `x`, les variables SSA successives sont représentées par les identifiants `x_0`, `x_1`, `x_2`, etc.

Concrètement, lors du traitement d'une instruction $x = e$, les occurrences des variables du programme apparaissant dans l'expression de droite sont remplacées par les identifiants de leurs variables SSA les plus récentes (ligne 12 de l'Algorithme 3). Le dernier identifiant de variable SSA associé à la variable du programme `x` est ensuite récupéré, son compteur est incrémenté (ligne 13), puis une nouvelle variable SSA, représentée par un nouvel identifiant, est créée (ligne 15).

Si l'affectation apparaît dans un contexte conditionnel actif, la nouvelle définition SSA est enrichie par la conjonction des conditions présentes dans la pile (ligne 14). Cette information est essentielle car elle permettra ultérieurement de générer des expressions SMT de type `ite`, préservant fidèlement la sémantique du flot de contrôle et modélisant le rôle de la fonction ϕ.

Le processus se poursuit jusqu'à l'instruction cible, moment auquel le parcours est interrompu.

Algorithm 3: Construction de la table SSA jusqu'à l'instruction cible

Input: Corps de la fonction

Data: Table SSA vide, pile des conditions active vide

Output: Table SSA mise à jour

```
Initialiser les paramètres comme variables symboliques;
foreach instruction jusqu'à la cible do
    if instruction est if(cond) then
        Empiler cond;
        Parcourir le bloc then;
        Dépiler;
        if bloc else existe then
            Empiler ¬cond;
            Parcourir le bloc else;
            Dépiler;
    if instruction est une affectation x = e then
        Remplacer les variables de e par leurs versions SSA courantes;
        Incrémenter le compteur SSA de x;
        Associer les conditions actives si nécessaire;
        Insérer la nouvelle définition dans la table SSA;
    if instruction est la cible then
        Arrêter le parcours;
```

Que mémorise `DefinitionInfo` pour chaque définition SSA ?

Chaque entrée de `varDefs` est un objet `DefinitionInfo` (Listing 5.4), qui va bien au-delà d'une table de symboles classique.

Listing 5.4: Structure `DefinitionInfo` — cœur du suivi SSA

```
struct DefinitionInfo {
    SSAVariable ssaVar;           // nom et version SSA (ex. x_1)
    unsigned    line;             // ligne dans le code source
    unsigned    stmtID;           // identifiant du noeud AST
    std::string varType;          // type C de la variable (ex. "int")
    std::string exprString;       // expression SSA textuelle
    std::vector<std::string> usedVars;
    std::vector<PathCondition>   globalConditionContext;
    std::string smtDefinitionExpression; // expression Python/Z3 avec
        encodage debordement
```

```
10      bool        isCondtionalDefinition;
11      const clang::Stmt* defStmt = nullptr;
12  };
```

Deux champs sont essentiels pour la génération du script. Le champ `smtDefinitionExpression` contient l'expression arithmétique traduite en notation Python/Z3, avec l'encodage du débordement produit par `handleArithmeticWithOverflow` (décrite à la Section 5.3.4). Le champ `globalConditionContext` conserve les gardes actifs au moment de la définition. C'est lui qui détermine si la définition sera encodée sous forme inconditionnelle ou sous la forme `If(cond, newVal, prevVersion)` — cette dernière structure reproduisant fidèlement la sémantique des nœuds ϕ de la forme SSA standard, sans les construire explicitement. Notons que `If()` est l'opérateur *if-then-else* de l'interface Python de Z3 ; il correspond exactement à l'opérateur `ite(cond, val1, val2)` du format `SMT-LIB2`, et les deux notations encodent la même sémantique.

Comment les structures conditionnelles sont-elles prises en compte ?

Le traitement des `IfStmt` est la clé de voûte du mécanisme. Lorsque `visitAllStmts` rencontre un nœud `IfStmt`, elle réécrit d'abord la condition dans les versions SSA courantes des variables, puis empile le garde dans `gloablPathConditions` avant de parcourir récursivement le bloc `then`. À la sortie, le garde est dépilé. Si un bloc `else` est présent, sa négation est empilée pendant le parcours de ce second bloc, puis dépilée à sa sortie. Ce mécanisme garantit que chaque définition créée à l'intérieur d'un bloc conditionnel porte exactement les gardes actifs au moment de sa création.

Une variable définie dans un bloc conditionnel reçoit une version incrémentée. Si la variable `x` avait la version `x_0` avant le bloc, l'affectation à l'intérieur du `then` produit `x_1`. Sa définition SMT prend alors la forme :

```
x_1 = If(condition, nouvelleValeur, x_0)
```

ce qui exprime que si la condition est vraie, `x` prend la nouvelle valeur ; sinon elle conserve sa valeur précédente `x_0`. Le Tableau 5.1 illustre le contenu de `varDefs` après l'analyse d'un fragment conditionnel.

Table 5.1: Exemple de table `varDefs` après analyse d'un fragment conditionnel

Nom SSA	Ligne	Type	Expression Python/Z3	Contexte conditionnel
`x_0`	2	`int`	`x_0 = 1`	$\emptyset$
`x_1`	5	`int`	`x_1 = If(x_0 > 0, x_0 + n_0, x_0)`	`{x_0 > 0}`

5.3.3 Calcul de la condition de chemin d'exécution

Comment la pile de gardes capture-t-elle le chemin vers la cible ?

La **condition de chemin** est la conjonction des gardes accumulés dans la pile `gloablPathConditions` au moment où `visitAllStmts` atteint l'instruction cible. Chaque garde est encapsulé dans un objet `PathCondition` qui mémorise *deux formulations distinctes* :

- `smtString` — expression Z3 du garde dans les versions SSA courantes (ex. `uid_sid_0 == 0`) ;
- `smtStringWithSafetyAssertion` — version enrichie produite par `handleArithmeticWithOverflow`, qui encode un éventuel débordement dans la condition elle-même, garantissant que même les gardes participent à la détection.

C'est la seconde formulation qui est injectée dans le script Python/Z3 généré.

Le Listing 5.5 montre comment la pile est manipulée lors du traitement d'un `IfStmt` dans `visitAllStmts`.

Listing 5.5: Gestion de la pile de conditions lors du traitement d'un `IfStmt`

```
std::string safetyCondText =
    DefinitionInfo::handleArithmeticWithOverflow(condExpr);

if (const Stmt* thenBody = ifStmt->getThen()) {
    PathCondition pc(condExpr, true, usedVars);
    pc.smtString                    = rewrittenCond;
    pc.smtStringWithSafetyAssertion = safetyCondText;
    gloablPathConditions.push(pc);
    if (visitAllStmts(thenBody)) return true; // retourne vrai si cible atteinte
    gloablPathConditions.pop();
}
if (const Stmt* elseBody = ifStmt->getElse()) {
    PathCondition pc(condExpr, false, usedVars);
    pc.smtString                    = "Not(" + rewrittenCond + ")";
    pc.smtStringWithSafetyAssertion = "Not(" + safetyCondText + ")";
    gloablPathConditions.push(pc);
    if (visitAllStmts(elseBody)) return true;
    gloablPathConditions.pop();
}
```

Ce schéma garantit que, pour tout niveau d'imbrication, la pile contient exactement les gardes nécessaires pour atteindre l'instruction cible par le chemin analysé. La conjonction de ces gardes, exprimée dans l'espace des versions SSA et enrichie de l'encodage du débordement, constitue la précondition de chemin injectée dans le script sous la forme `s.add(garde)` pour chaque garde de la pile.

5.3.4 Génération du script Python/Z3

Comment `exploreFunctionByStmt` orchestre-t-elle la génération du script ?

La génération du script est orchestrée par la méthode `exploreFunctionByStmt` de la classe `TargetProgramPoint`. Cette méthode enchaîne, dans l'ordre, les cinq fonctions de génération suivantes, chacune produisant une section distincte du script `checking.py` :

1. `genVariableDeclarationPyZ3()` — déclarations des variables Z3 ;
2. `genVariableDefinitionPyZ3()` — définitions SSA avec encodage du débordement, triées topologiquement selon les dépendances ;
3. `genConditionPathPyZ3()` — assertions de la condition de chemin ;
4. `genConstraintsPyZ3()` — contraintes utilisateur issues de `target.json` ;
5. `genTypeRangeConstraintsPyZ3()` — bornes de type calculées depuis $\mathcal{KD}_A$.

Avant d'invoquer ces fonctions, `exploreFunctionByStmt` sélectionne les variables pertinentes par **atteignabilité depuis la cible** : seules les variables qui apparaissent — directement ou via une chaîne de dépendances dans `usedVars` — dans la condition de chemin ou dans les définitions qu'elle référence sont retenues. Ce filtrage réduit la taille du script et améliore les performances du solveur.

Comment les variables sont-elles déclarées dans le script ?

La fonction `genVariableDeclarationPyZ3` émet, pour chaque variable pertinente, une ligne de la forme :

```
varName_i = Int('varName_i')
```

Les variables fraîches d'overflow (`k1`, `xbar1`, etc.) sont déclarées de la même façon.

Comment le débordement potentiel est-il encodé dans les définitions SSA ?

La fonction `genVariableDefinitionPyZ3` construit d'abord un graphe de dépendances entre les versions SSA et procède à un **tri topologique** pour garantir que chaque définition apparaît après celles dont elle dépend. Elle produit ensuite pour chaque variable, selon son contexte, l'une des deux formes suivantes.

Pour une **définition inconditionnelle** sans opération arithmétique sujette à débordement :

```
uid_sid_0 = (userId_0 * serviceId_0)
```

Pour une **définition avec encodage du débordement**, produite par la fonction `handleArithmeticWithOverflow` :

Listing 5.6: Encodage du débordement dans une définition SSA (Python/Z3)

```
uid_sid_0 = If(
    And(
        (userId_0 * serviceId_0) == 2**32 * k1 + xbar1,
        k1 > 0
    ),
    xbar1,              # valeur stockee si debordement (wrap-around)
    (userId_0 * serviceId_0)  # valeur exacte si pas de debordement
)
```

La largeur n (ici 32) est fournie par `getBitWidthForType` en interrogeant $\mathcal{KD}_A$ avec l'environnement spécifié dans `target.json`. C'est par ce point précis que l'environnement cible influence les contraintes : un `long` passera de $n = 64$ bits (LP64, Linux) à $n = 32$ bits (LLP64, Windows), modifiant la frontière de débordement. La variable `k1` représente le quotient de débordement et `xbar1` le résidu (valeur stockée après wrap-around).

Pour une **définition conditionnelle** (variable affectée dans une branche `if/else`), la structure est :

`x_1 = If(condition_smt, nouvelleValeur, x_0)`

Cette expression modélise la sémantique du nœud ϕ : si la condition est vraie, `x` prend la nouvelle valeur ; sinon elle conserve sa valeur précédente `x_0`.

Quelles assertions constituent le cœur du script généré ?

Les trois dernières fonctions de génération émettent des assertions `s.add(...)` sur un objet `Solver` Z3 :

Listing 5.7: Structure des assertions dans le script `checking.py` généré

```
# *** Condition de chemin (genConditionPathPyZ3) ***
s = Solver()
s.add(uid_sid_0 == 0)         # garde de la branche then a la cible

# *** Contraintes utilisateur (genConstraintsPyZ3) ***
s.add(userId_0 >= 0)
s.add(userId_0 < 150000000)
s.add(userId_0 != 0)
s.add(serviceId_0 > 0)
s.add(serviceId_0 < 65)

# *** Bornes de types depuis KnowledgeBase (genTypeRangeConstraintsPyZ3) ***
# modele LLP64 : long = 32 bits sur Windows x86
s.add(userId_0 >= -2**31);  s.add(userId_0 <= 2**31 - 1)
s.add(serviceId_0 >= -2**31); s.add(serviceId_0 <= 2**31 - 1)
s.add(uid_sid_0 >= -2**31); s.add(uid_sid_0 <= 2**31 - 1)

```

```
# *** Verification et modele ***
print(s.check())
if s.check() == sat:
    print(s.model())
```

Les bornes de type, calculées à partir de la largeur résolue via $\mathcal{KD}_A$, expriment que chaque variable appartient à l'intervalle $[-2^{w-1},\ 2^{w-1}-1]$ défini par les limites arithmétiques de son type sur l'architecture cible. Elles sont indispensables pour éviter que le solveur ne produise des contre-exemples mathématiquement valides mais physiquement impossibles sur la plateforme considérée.

5.4 Étude de cas : exécution complète de l'outil

Pour illustrer les mécanismes décrits dans les sections précédentes, nous déroulons tout le pipeline sur un exemple concret de vulnérabilité par débordement entier. Cet exemple est traité en trois temps : présentation du code source et de l'instruction cible, déroulement pas à pas des phases d'analyse, puis examen des contraintes générées et de la réponse du solveur.

5.4.1 Présentation du code source et de l'instruction cible

Nous considérons un scénario de contrôle d'accès distribué avec l'outil (fichier `samples/pseudo.c`), pour illustrer la résolution du modèle de données par $\mathcal{KD}_A$. Nous le considérons ici pour suivre l'intégralité du pipeline, de la localisation de l'instruction cible jusqu'à l'interprétation du contre-exemple produit par le solveur. La plateforme expose deux niveaux de privilèges : un **compte administrateur** (`userId = 0`) avec accès en lecture-écriture, et des **utilisateurs réguliers** (`userId` $\in [1,\ 150\,000\,000]$) limités à la lecture seule.

Listing 5.8: Programme source `samples/pseudo.c` : contrôle d'accès par produit arithmétique

```
void readAndWriteService(int a, int b) {}  // admin : lecture + ecriture
void readOnlyService(int a, int b) {}      // utilisateur regulier :
    lecture seule

void foo(char username[], char password[]) {
    int userId    = random_int(0, 150000000);
    int serviceId = random_int(1, 64);

    long int uid_sid = (long)userId * serviceId;

```

```
10      if (uid_sid == 0) {
11          readAndWriteService(uid_sid, serviceId); // ligne 11 -- branche
                admin
12      } else {
13          readOnlyService(uid_sid, serviceId);
14      }
15  }
```

La **question de vérification** : existe-t-il un utilisateur régulier (`userId` $\neq 0$) pour lequel la branche `readAndWriteService` à la **ligne 11** serait malgré tout atteinte ?

Le fichier `target.json` du Listing 5.9 désigne la ligne 11 comme cible et encode l'environnement Windows x86 — où le type `long` n'occupe que **32 bits** (modèle LLP64) contre 64 bits sur Linux (LP64).

Listing 5.9: Fichier `target.json` : hypothèses du développeur sur Windows x86

```
1  *{
2    "target": 11,
3    "constraints": [
4      { "variable": "userId",    "operator": ">=", "value": 0          }
           ,
5      { "variable": "userId",    "operator": "<",  "value": 150000000 }
           ,
6      { "variable": "userId",    "operator": "!=", "value": 0          }
           ,
7      { "variable": "serviceId", "operator": ">",  "value": 0          }
           ,
8      { "variable": "serviceId", "operator": "<",  "value": 65         }
9    ],
10   "executionEnv": { "Arch": "x86", "OS": "windows", "flags": [] }
11 }
```

5.4.2 Déroulement de l'analyse étape par étape

Étape 1 — Localisation et fonction englobante. `ConditionPathAstVisitor` identifie, à la ligne 11, un nœud `CallExpr` correspondant à l'appel `readAndWriteService(uid_sid, serviceId)`. La méthode `findParentStmt` confirme que l'instruction appartient à la fonction `foo`.

Étape 2 — Étiquetage SSA. Le parcours séquentiel de `foo` par `visitAllStmts` enregistre `userId_0` et `serviceId_0` comme variables symboliques libres — les appels à `random_int` retournent une chaîne vide dans `exprToPyZ3`, et sont donc traités comme des variables symboliques. La déclaration `long int uid_sid = (long)userId * serviceId` est le point critique : lors de son traitement par `VisitBinaryOperator`, la fonction `handleArithmeticWithOverflow` interroge $\mathcal{KD}_A$ pour le type `long` sur Windows x86, obtient $n = 32$ bits via le modèle LLP64, et produit la définition SSA avec encodage du débordement (voir Listing 5.6).

Étape 3 — Condition de chemin. La ligne 11 se trouve dans le bloc `then` de l'`IfStmt`. Lorsque `visitAllStmts` entre dans ce bloc, la condition `uid_sid_0 == 0` est empilée dans `gloablPathConditions`. C'est cette condition qui constituera la précondition de chemin du script généré.

5.4.3 Contraintes SMT générées et résultat du solveur

L'invocation des cinq fonctions de génération dans `exploreFunctionByStmt` produit le script `checking.py` du Listing 5.10, dont les commentaires indiquent quelle fonction a généré chaque section.

Listing 5.10: Script `checking.py` généré par `AETHER` — pour Listing 5.8

```
1  from z3 import *
2
3  # *** Variables Declarations ***          <-- genVariableDeclarationPyZ3
4  serviceId_0 = Int('serviceId_0')
5  uid_sid_0   = Int('uid_sid_0')
6  userId_0    = Int('userId_0')
7  k1          = Int('k1')
8  xbar1       = Int('xbar1')
9
10 # *** Variables Definitions ***          <-- genVariableDefinitionPyZ3
11 # userId_0 is a free symbolic variable (SymbInt32)
12 # serviceId_0 is a free symbolic variable (SymbInt32)
13 uid_sid_0 = If(
14     And(
15         (userId_0 * serviceId_0) == 2**32 * k1 + xbar1,
16         k1 > 0
17     ),
18     xbar1,
19     (userId_0 * serviceId_0)
20 )
21
```

```
# *** Condition path ***                    <-- genConditionPathPyZ3
s = Solver()
s.add(uid_sid_0 == 0)

# *** User constraints ***                  <-- genConstraintsPyZ3
s.add(userId_0 >= 0)
s.add(userId_0 < 150000000)
s.add(userId_0 != 0)
s.add(serviceId_0 > 0)
s.add(serviceId_0 < 65)

# *** Type range constraints ***            <-- genTypeRangeConstraintsPyZ3
s.add(serviceId_0 >= -2**31); s.add(serviceId_0 <= 2**31 - 1)
s.add(uid_sid_0   >= -2**31); s.add(uid_sid_0   <= 2**31 - 1)
s.add(userId_0    >= -2**31); s.add(userId_0    <= 2**31 - 1)

# *** Checking satisfiability and getting models ***
print(s.check())
if(s.check()==sat):
    print(s.model())
```

La sortie produite par AETHER après exécution de `python3 checking.py` :

Listing 5.11: Sortie complète d'AETHER sur le cas d'étude

```
[KB] loaded 4 data models, 18 rules from ./KnowledgeBase.json
target instruction : 11

TARGET FOUND: CallExpr at line 11

=== SMT Context ===
Target path conditions:
  uid_sid_0 == 0
Relevant variables:  serviceId_0  uid_sid_0  userId_0

[*] Script written to checking.py

 ----- Durations -----
 AST build    :  5.3 ms
 script gen   :  2.6 ms
 Z3 checking  :  788 ms

sat
[serviceId_0 = 64, xbar1 = 0, userId_0 = 134217728, k1 = 2]
```

La réponse `sat` constitue un **contre-exemple certifié** invalidant l'hypothèse de sécurité du développeur. Les valeurs `userId_0 = 134 217 728` ($= 2^{27}$) et `serviceId_0 = 64` ($= 2^6$) sont toutes deux légitimes selon les contraintes utilisateur. Leur produit exact

$2^{27} \times 2^6 = 2^{33} = 2 \times 2^{32} + 0$ déborde sur un `long` de 32 bits : le quotient est $k_1 = 2$ et le résidu est $\bar{x}_1 = 0$. La variable `uid_sid` prend donc la valeur **zéro** en machine, et le garde `uid_sid == 0` est satisfait malgré `userId` $\neq 0$.

La sensibilité architecturale est décisive : avec le modèle LP64 (Linux 64 bits, `long` = 64 bits), le même produit 2^{33} ne déborde pas dans l'espace de 2^{64}, et Z3 répondrait `unsat` — un faux négatif architectural que $\mathcal{KD}_A$ permet d'éliminer.

5.5 Discussion et limites de l'implémentation

La présentation du pipeline et de l'étude de cas a mis en évidence les capacités du prototype. Cette section examine ses contraintes actuelles, inhérentes aux choix de conception retenus, puis identifie les axes d'extension qui permettraient d'en élargir la portée.

5.5.1 Limites actuelles

L'outil est un prototype fonctionnel reposant sur plusieurs hypothèses simplificatrices.

Portée intra-procédurale. L'analyse se confine au corps de la fonction englobante. Les appels inter-procéduraux sont modélisés par des variables symboliques abstraites (`SymbInt32`, etc.), ce qui est conservatif mais peut introduire des faux positifs si la sémantique du corps appelé contraint la valeur de retour plus précisément que le type seul.

Absence de déroulage de boucles. Les structures itératives ne sont pas analysées : `visitAllStmts` arrête le parcours à la première occurrence de l'instruction cible, sans considérer les exécutions répétées de boucles. Des débordements conditionnés par un nombre d'itérations ne seront donc pas détectés dans la version actuelle.

Identification par numéro de ligne. Cette approche, retenue pour sa simplicité, présente des fragilités connues : plusieurs instructions sur une même ligne, expansions de macros ou constructions générées par le préprocesseur peuvent conduire à une localisation incorrecte.

5.5.2 Perspectives d'extension et de généralisation

Analyse inter-procédurale. L'inlining contrôlé des fonctions appelées, ou une analyse par résumé (*summary-based*), permettrait de tenir compte de la sémantique précise des fonctions appelées et d'éliminer une large part des faux positifs actuels.

Déroulage symbolique borné des boucles. Cette extension requiert de gérer l'expansion des itérations dans `visitAllStmts` et la création des versions SSA correspondantes pour chaque itération déroulée.

Identification robuste de l'instruction cible. une solution plus robuste consisterait à intégrer une interface interactive permettant à l'utilisateur de sélectionner directement l'instruction cible ou d'y associer un marqueur explicite, suivant un principe similaire à celui des outils de débogage.

Extension de $\mathcal{KD}_A$. L'enrichissement de la base de connaissances — compilateurs supplémentaires, architectures embarquées, nouveaux triplets de compilation — élargit directement la portée de l'analyse sans aucune modification du moteur, grâce à la séparation entre résolution du modèle et logique de vérification.

5.6 Conclusion

Ce chapitre a présenté l'implémentation concrète de l'outil `AETHER`, en ancrant chaque étape du pipeline dans les fonctions et structures réelles du code source. Le pipeline conduit, depuis le chargement de la configuration et la construction de l'AST, jusqu'à la génération et l'exécution du script Python/Z3, en passant par l'étiquetage SSA sensible au chemin et le calcul de la condition de chemin.

Trois éléments constituent l'ossature technique originale. En premier lieu, la **résolution dynamique des tailles de types** via $\mathcal{KD}_A$, implémentée par `resolveDataModel` et `getBitWidthForType`, fait d'AETHER un vérificateur sensible à l'architecture cible. En second lieu, la **forme SSA sensible au chemin** : en associant à chaque définition de `DefinitionInfo` le champ `globalConditionContext`, l'outil préserve la sémantique du flot de contrôle et génère des expressions `If(cond, val, prev)` qui reproduisent les nœuds ϕ sans les construire. Enfin, **`handleArithmeticWithOverflow`** traduit chaque opération arithmétique en une contrainte Z3 qui rend observable le débordement potentiel avec une précision directement conditionnée par l'environnement cible.

L'étude de cas a démontré concrètement la valeur de cette sensibilité : une vulnérabilité d'escalade de privilèges par débordement entier, invisible sous le modèle LP64, est automatiquement découverte dès lors que l'environnement Windows x86 (LLP64) est correctement spécifié. Les limites identifiées — portée intra-procédurale, absence de déroulage de boucles, identification par numéro de ligne — délimitent le périmètre du prototype et fournissent une feuille de route naturelle pour ses extensions.

6 Conclusion générale

Les systèmes embarqués occupent aujourd'hui une place centrale dans des domaines critiques où la moindre défaillance peut avoir des conséquences graves : systèmes de transport autonome, équipements médicaux, infrastructures industrielles ou encore dispositifs de défense. La fiabilité de ces systèmes repose sur deux dimensions étroitement liées — la **sûreté** et la **sécurité** — dont l'articulation constitue le fil conducteur de cette thèse.

L'observation issue de nos travaux est que la fiabilité des systèmes est généralement étudiée soit au niveau logiciel, soit au niveau matériel, mais rarement à leur interface. Or, certaines classes de bogues de sûreté et de sécurité ne sont pas visibles dans le code source seul : elles n'apparaissent qu'une fois le programme déployé dans un environnement d'exécution particulier. Un programme syntaxiquement correct peut ainsi exhiber un comportement défaillant sur une plateforme cible et rester sans défaut apparent sur une autre. Pourtant, très peu de travaux d'analyse statique intègrent explicitement cette dimension environnementale dans leur modèle de vérification.
Notre problème est donc de **caractériser et détecter les bogues de programmes liés à leur exécution sur des plateformes particulières**. Une plateforme peut être caractérisée par de nombreux paramètres : son espace mémoire, ses capacités de calcul, son horloge d'exécution, etc. Dans cette thèse, nous nous sommes concentrés principalement sur **l'espace mémoire**, et plus précisément sur les largeurs effectives des types entiers qui en découlent, lesquelles varient selon l'architecture matérielle, le système d'exploitation et le compilateur.

Cette lacune constitue le point de départ de notre recherche. Un programme C syntaxiquement correct peut se comporter différemment selon qu'il est compilé pour une architecture 32 bits ou 64 bits, sous Windows ou sous Linux, avec GCC ou MSVC. Le type `long`, par exemple, occupe 32 bits sous Windows (modèle LLP64) mais 64 bits sous Linux (modèle LP64). Une multiplication qui reste dans les bornes arithmétiques sur l'une de ces plateformes peut silencieusement déborder sur l'autre, ouvrant la voie à une vulnérabilité d'escalade de privilèges ou de corruption mémoire.

L'objectif de cette thèse est de proposer **un cadre permettant aux développeurs d'analyser les programmes et d'identifier des failles de sécurité induites par des problèmes de sûreté liés à un environnement d'exécution particulier.** Par ailleurs, **ce cadre se veut extensible, afin de permettre aux chercheurs d'y intégrer de nouvelles contributions couvrant un éventail plus large de propriétés de sûreté**, au-delà des seuls problèmes liés aux débordements.

Pour répondre à cette problématique, nos travaux s'articulent autour de quatre contributions principales:

- **Une approche formelle intégrant conjointement le logiciel et l'impact du matériel sur son exécution.** Nous avons proposé une méthode d'analyse et de vérification basée sur les SMT solveurs, qui ramène la détection de vulnérabilités à un problème de satisfiabilité (Chapitre 3). Étant donnée une instruction cible dans un programme `C`, un ensemble de contraintes de sécurité SC spécifiées par le développeur, ainsi qu'un contexte d'exécution EC, l'approche consiste à déterminer s'il existe un scénario d'exécution dans cet environnement pour lequel la condition de chemin PC menant à cette instruction est satisfaite, tout en violant les contraintes de sécurité [57].
  Ce problème est formalisé par la requête suivante :

  $$EC \vdash PC \wedge \neg SC$$

  où EC désigne le contexte d'exécution. Un résultat *sat* prouve l'existence d'une vulnérabilité et fournit un contre-exemple concret ; un résultat *unsat* constitue une preuve formelle d'absence de violation.
- **Construction d'une base de connaissances architecturales $\mathcal{KD}_A$.**
  Nous proposons une base de connaissances structurée (Chapitre 4), visant à **modéliser l'impact des différents environnements d'exécution — incluant le système d'exploitation, le compilateur, l'architecture et les options de compilation** — sur les tailles effectives des types entiers du langage `C`. Cette

base permet de capturer, de manière explicite, les variations dépendantes de la plateforme qui influencent le comportement des programmes. Elle permet ainsi **l'instanciation des assertions de sûreté** utilisées lors de l'analyse [56].

- **Developpements des algorithmes pour automatiser l'approche.** Nous avons mis en œuvre une chaîne complète d'analyse, entièrement automatisée, allant de l'analyse statique à la génération de code SMT (Chapitre 3 et Chapitre 5). Cette chaîne inclut l'extraction des formules depuis la base de connaissances, la construction des conditions de chemin (path conditions), et la formulation du problème sous forme de contraintes logiques. Elle permet de produire des résultats précis, indiquant la présence ou l'absence de violations des propriétés de sûreté ou de sécurité [77].
- **Implementation de l'outil AETHER — une chaîne d'analyse automatisée.** Nous avons implémenté l'outil `AETHER` (*Architecture-aware Execution and THEoretical Reasoning*) `https://github.com/NazimKYS/AETHER`, qui matérialise concrètement l'ensemble de l'approche (Chapitre 5). Reposant sur l'infrastructure Clang/LLVM pour la construction de l'AST et l'analyse statique, `AETHER` orchestre un pipeline en quatre phases : localisation de l'instruction cible, étiquetage SSA sensible au chemin, calcul de la condition de chemin, puis génération et exécution d'un script Python/Z3. L'outil résout les tailles de types entiers au moment de l'analyse via les fonctions `resolveDataModel` et `getBitWidthForType`, ce qui le rend sensible à l'architecture cible plutôt qu'à la machine sur laquelle il s'exécute. Le cas d'usage de référence — une vulnérabilité d'escalade de privilèges par débordement d'une multiplication `long` sous Windows x86 — illustre comment une faille invisible sous un modèle 64 bits est automatiquement détectée dès que l'environnement cible est correctement spécifié.

Les travaux présentés dans cette thèse ont permis de proposer un cadre formel d'analyse statique sensible à l'environnement d'exécution, intégrant conjointement le comportement du programme et l'impact des caractéristiques matérielles. Cette approche, concrétisée par l'outil `AETHER` et la base de connaissances $\mathcal{KD}_A$, permet de détecter des vulnérabilités dépendantes de la plateforme cible, tout en produisant des preuves formelles ou des contre-exemples exploitables.

Sur le plan méthodologique, l'approche définie couvre un ensemble de constructions du langage `C`, incluant notamment les structures conditionnelles ainsi que les boucles, ces dernières étant traitées au niveau théorique à travers des techniques **telles que le déroulage borné**. Toutefois, le support des **boucles** n'est pas encore intégré dans `AETHER`, ce qui limite l'analyse de programmes comportant des boucles complexes. De

même, l'analyse demeure, dans sa version actuelle, strictement **intra-procédurale** : les appels de fonctions sont modélisés comme des entités abstraites, assimilées à des variables symboliques dont le type correspond à celui de la valeur de retour. Cette modélisation, bien que correcte du point de vue formel, reste approximative et peut engendrer des imprécisions dans certains cas.

Par ailleurs, la prise en compte des appels imbriqués et des mécanismes de récursivité nécessite des investigations supplémentaires afin de déterminer des stratégies d'intégration adaptées dans le cadre proposé. Enfin, les situations dans lesquelles les solveurs SMT retournent un résultat ***unknown*** ne sont pas explicitement traitées dans la version actuelle, ce qui constitue une limite en termes de robustesse du processus de vérification.
Plusieurs **perspectives de recherche** peuvent être envisagées afin de prolonger et d'enrichir les travaux présentés:

- **À court terme**, une priorité consiste à renforcer l'implémentation de l'outil AETHER en intégrant le traitement effectif des **structures itératives**, conformément au modèle théorique proposé. Par ailleurs, une meilleure gestion des résultats ***unknown*** retournés par les solveurs SMT pourrait être mise en place, notamment en combinant plusieurs solveurs reposant sur des stratégies de résolution différentes, et en exploitant le premier résultat concluant (*sat* ou *unsat*) obtenu.
- **À moyen terme**, l'extension de l'analyse vers un cadre inter-procédural constitue une évolution majeure, permettant de capturer plus fidèlement les interactions entre fonctions, y compris dans des contextes impliquant des appels imbriqués ou récursifs. Une autre perspective importante concerne **l'extension de l'approche aux architectures parallèles**, notamment dans des environnements exploitant des unités de calcul telles que les **GPU**, où les modèles d'exécution diffèrent significativement des architectures **CPU** traditionnelles. Par ailleurs, **l'enrichissement de la base de connaissances** $\mathcal{KD}_A$ à d'autres dimensions de la sûreté constitue une piste prometteuse, incluant non seulement l'espace mémoire, mais également des propriétés temporelles ou liées à la vitesse des processeurs. Enfin, **l'intégration de l'outil sous forme d'extension** dans des environnements de développement tels que Visual Studio Code permettrait de faciliter son adoption dans des contextes pratiques.
- **À plus long terme**,
  plusieurs axes exploratoires peuvent être envisagés. L'extension de l'approche à l'analyse de modèles d'intelligence artificielle, tels que les réseaux de neurones ou les modèles de langage de grande taille, constitue une direction particulière-

ment prometteuse, notamment dans une optique de vérification de propriétés de sûreté et de robustesse. Plus largement, l'intégration de techniques issues de l'intelligence artificielle pourrait également contribuer à améliorer certaines étapes de l'analyse, en guidant par exemple l'exploration des chemins ou la génération de contraintes. Enfin, l'étude de nouveaux paradigmes de calcul, tels que les architectures quantiques, pourrait ouvrir des perspectives inédites pour l'évolution des méthodes de vérification formelle.

Enfin, cette thèse démontre qu'une prise en compte explicite du contexte d'exécution permet de dépasser les limites des approches classiques d'analyse statique. En réconciliant les dimensions logicielle et matérielle dans un cadre formel unifié, elle ouvre la voie à des méthodes de vérification plus précises, mieux adaptées aux systèmes critiques et capables de révéler des vulnérabilités jusque-là invisibles. Elle constitue, nous l'espérons, une contribution utile à la communauté de la vérification formelle et de la sécurité des systèmes embarqués, et un socle sur lequel des extensions futures pourront s'appuyer.

Références

[1] *Affected Processors: Transient Execution Attacks & Related Security... — intel.com*. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html. [Accessed 13-03-2025].

[2] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Boston, MA: Addison-Wesley Professional, 2007. ISBN: 0-321-48681-1.

[3] Anil Altinay et al. "BinRec: dynamic binary lifting and recompilation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[4] A W Appel and M Ginsburg. *Modern compiler implementation in C*. Cambridge: Cambridge Univ. Press, 2010.

[5] Ludovic Apvrille and Yves Roudier. "SysML-Sec: A Model Driven Approach for Designing Safe and Secure Systems". In: *3rd International Conference on Model-Driven Engineering and Software Development*. France: SCITEPRESS Digital Library, Feb. 2015. URL: https://hal.telecom-paris.fr/hal-02287013.

[6] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. "Bounded model checking of software using SMT solvers instead of SAT solvers". In: *International Journal on Software Tools for Technology Transfer* 11.1 (2009), pp. 69–83. ISSN: 1433-2787. DOI: 10.1007/s10009-008-0091-0. URL: https://doi.org/10.1007/s10009-008-0091-0.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, Nov. 2023.

[8] Andrei Arusoaie et al. "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code". In: *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2017, pp. 161–168. DOI: 10.1109/SYNASC.2017.00035.

[9] Roberto Baldoni et al. “A survey of symbolic execution techniques”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–39.

[10] Haniel Barbosa et al. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9.

[11] David Basin. *Formal Methods for Security Knowledge Area*. Ed. by Steve Schneider. Version 1.0.0. With contributions from reviewers: Rod Chapman, Stephen Chong, Mark Ryan, Andrei Sabelfeld. Cyber Security Body of Knowledge (CyBOK), 2020. URL: https://www.cybok.org/media/downloads/Formal_Methods_for_Security_v1.0.0.pdf.

[12] Dirk Beyer. “State of the Art in Software Verification and Witness Validation: SV-COMP 2024”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Cham: Springer Nature Switzerland, 2024, pp. 299–329. ISBN: 978-3-031-57256-2.

[13] Corinne Bieder and Kenneth Pettersen Gould. “Exploring the Interrelations Between Safety and Security: Research and Management Challenges”. In: *The Coupling of Safety and Security: Exploring Interrelations in Theory and Practice*. Ed. by Corinne Bieder and Kenneth Pettersen Gould. Cham: Springer International Publishing, 2020, pp. 105–113. ISBN: 978-3-030-47229-0. DOI: 10.1007/978-3-030-47229-0_11. URL: https://doi.org/10.1007/978-3-030-47229-0_11.

[14] Frank S de Boer and Marcello Bonsangue. “Symbolic execution formally explained”. In: *Formal Aspects of Computing* 33.4 (2021), pp. 617–636.

[15] Marcel Böhme et al. “Software Security Analysis in 2030 and Beyond: A Research Roadmap”. In: *ACM Trans. Softw. Eng. Methodol.* 34.5 (May 2025). ISSN: 1049-331X. DOI: 10.1145/3708533. URL: https://doi.org/10.1145/3708533.

[16] Cristina Borralleras et al. “Incomplete SMT techniques for solving non-linear formulas over the integers”. In: *ACM Transactions on Computational Logic (TOCL)* 20.4 (2019), pp. 1–36.

[17] El Habib Boudjema et al. “VYPER: Vulnerability detection in binary code”. In: vol. 3. 2. Wiley, Dec. 2019. DOI: 10.1002/spy2.100. URL: https://hal.archives-ouvertes.fr/hal-02485434.

[18] Fraser Brown, Deian Stefan, and Dawson Engler. “Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 199–

216. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/brown.

[19] Jochen Burghardt et al. “ACSL by Example”. In: *DEVICE-SOFT project publication. Fraunhofer FIRST Institute* (2010).

[20] Frank Busse, Martin Nowack, and Cristian Cadar. “Running symbolic execution forever”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 63–74. ISBN: 9781450380089. DOI: 10.1145/3395363.3397360. URL: https://doi.org/10.1145/3395363.3397360.

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[22] Miguel Castro, Manuel Costa, and Tim Harris. “Securing software by enforcing data-flow integrity”. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI ’06. Seattle, Washington: USENIX Association, 2006, pp. 147–160. ISBN: 1931971471.

[23] *CERN Computer Security*. https://security.web.cern.ch. [Accessed 10-02-2024].

[24] Julian Cohen. *Contemporary Automatic Program Analysis (Slide 4: Automatic Program Analysis)*. Presentation at Black Hat USA 2014, August 2–7, Mandalay Bay Las Vegas, NV. 2014. URL: http://blackhat.com/docs/us-14/materials/us-14-Cohen-Comtemporary-Automatic-Program-Analysis.pdf (visited on 08/23/2025).

[25] LoıHERE!HERE!c Correnson et al. “Frama-C user manual”. In: *CEA LIST* 111 (2010).

[26] PATRICK COUSOT and RADHIA COUSOT. “Abstract Interpretation Frameworks”. In: *Journal of Logic and Computation* 2.4 (Aug. 1992), pp. 511–547. ISSN: 0955-792X. DOI: 10.1093/logcom/2.4.511. eprint: https://academic.oup.com/logcom/article-pdf/2/4/511/2740133/2-4-511.pdf. URL: https://doi.org/10.1093/logcom/2.4.511.

[27] cpredef contributors. *Data Models*. [Accessed 17-09-2024]. GitHub. Mar. 2024. URL: https://github.com/cpredef/predef/blob/master/DataModels.md (visited on 09/13/2025).

[28] Rick David Craig and Stefan P Jaskiel. *Systematic software testing*. Artech House, 2002.

[29] CWE. “Common Weakness Enumeration”. In: (2021). [Accessed 10-02-2024]. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[30] Ron Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[31] Lesly Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure”. In: *ACM Transactions on Privacy and Security* 26 (2 Apr. 2023). ISSN: 24712574. DOI: 10.1145/3563037.

[32] Robin David et al. “BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis”. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 653–656. DOI: 10.1109/SANER.2016.43. URL: https://doi.org/10.1109/SANER.2016.43.

[33] Leonardo De Moura and Nikolaj Børner. “Z3: an efficient SMT solver”. In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’08/ETAPS’08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3540787992.

[34] CVE details. *Évolution des vulnérabilités par type (Overflow et Corruption de la mémoire) depuis 2015*. https://www.cvedetails.com/. [Accessed 02-02-2025]. 2025.

[35] Will Dietz et al. “Understanding Integer Overflow in C/C++”. In: *ACM Trans. Softw. Eng. Methodol.* 25.1 (Dec. 2015). ISSN: 1049-331X. DOI: 10.1145/2743019. URL: https://doi.org/10.1145/2743019.

[36] Shiyu Dong et al. “Studying the influence of standard compiler optimizations on symbolic execution”. In: *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 205–215. DOI: 10.1109/ISSRE.2015.7381814. URL: https://doi.org/10.1109/ISSRE.2015.7381814.

[37] Blane Erwin. *The Groundbreaking 2015 Jeep Hack Changed Automotive Cybersecurity — fractionalciso.com*. https://fractionalciso.com/the-groundbreaking-2015-jeep-hack-changed-automotive-cybersecurity/. [Accessed 12-03-2025].

[38] CEA LIST, Software Safety and Security Laboratory and Inria Saclay - Île-de-France, Toccata team. *Frama-C - Framework for Modular Analysis of C programs*. https://frama-c.com/. Common development with LRI-CNRS and Université Paris-Sud 11. 2008.

[39] Nils Froleyks et al. "SAT Competition 2020". In: *Artificial Intelligence* 301 (2021), p. 103572. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2021.103572. URL: https://www.sciencedirect.com/science/article/pii/S0004370221001235.

[40] Mikhail R. Gadelha et al. "ESBMC 5.0: An Industrial-Strength C Model Checker". In: *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018, pp. 888–891. DOI: 10.1145/3238147.3240481.

[41] Christophe Garion et al. "A gentle introduction to C code verification using the Frama-C platform". PhD thesis. ISAE-SUPAERO; ONERA–The French Aerospace Lab; ENAC, 2022.

[42] Weiwei Gong and Xu Zhou. "A survey of SAT solver". In: *Applied Mathematics and Computer Science*. Vol. 1836. American Institute of Physics Conference Series. June 2017, 020059, p. 020059. DOI: 10.1063/1.4981999.

[43] Anjana Gosain and Ganga Sharma. "A Survey of Dynamic Program Analysis Techniques and Tools". In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Ed. by Suresh Chandra Satapathy et al. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5.

[44] Anjana Gosain and Ganga Sharma. "Static Analysis: A Survey of Techniques and Tools". In: *Intelligent Computing and Applications*. Ed. by Durbadal Mandal et al. New Delhi: Springer India, 2015, pp. 581–591. ISBN: 978-81-322-2268-2.

[45] Jera Hensel et al. "Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution". In: *Journal of Logical and Algebraic Methods in Programming* 97 (2018), pp. 105–130. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2018.02.004. URL: https://www.sciencedirect.com/science/article/pii/S2352220817300524.

[46] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999 draft. 1999. URL: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

[47] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[48] Muhammad Abid Jamil et al. “Software Testing Techniques: A Literature Review”. In: *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*. 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.

[49] Young-Su Jang and Jin-Young Choi. “Automatic Prevention of Buffer Overflow Vulnerability Using Candidate Code Generation”. In: *IEICE Trans. Inf. Syst.* 101-D (2018), pp. 3005–3018.

[50] Dae R Jeong et al. “Razzer: Finding kernel race bugs through fuzzing”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 754–768.

[51] Fuqi Jia et al. “Improving Bit-Blasting for Nonlinear Integer Constraints”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 14–25. ISBN: 9798400702211. DOI: 10.1145/3597926.3598034. URL: https://doi.org/10.1145/3597926.3598034.

[52] Timotej Kapus, Martin Nowack, and Cristian Cadar. “Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?” In: *Tests and Proofs*. Ed. by Dirk Beyer and Chantal Keller. Cham: Springer International Publishing, 2019, pp. 41–54. ISBN: 978-3-030-31157-5.

[53] Arvinder Kaur and Ruchikaa Nayyar. “A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code”. In: *Procedia Computer Science* 171 (2020). Third International Conference on Computing and Network Communications (CoCoNet’19), pp. 2023–2029. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.04.217. URL: https://www.sciencedirect.com/science/article/pii/S1877050920312023.

[54] Seokmo Kim, R. Young Chul Kim, and Young B. Park. “Software Vulnerability Detection Methodology Combined with Static and Dynamic Analysis”. In: *Wireless Personal Communications* 89 (3 Aug. 2016), pp. 777–793. ISSN: 1572834X. DOI: 10.1007/s11277-015-3152-1.

[55] Florent Kirchner et al. “Frama-C: A software analysis perspective”. In: *Formal aspects of computing* 27.3 (2015), pp. 573–609.

[56] Salim Yahia Kissi, Rabéa Ameur-Boulifa, and Yassamin Seladji. “Security Vulnerabilities Detection Through Assertion-Based Approach”. In: *Theoretical Aspects of Software Engineering: 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8–10, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, pp. 381–387. ISBN: 978-3-031-10362-9. DOI: 10.1007/978-3-031-10363-6_25. URL: https://doi.org/10.1007/978-3-031-10363-6_25.

[57] Salim Kissi., Yassamine Seladji., and Rabéa Ameur-Boulifa. “Detection of Security Vulnerabilities Induced by Integer Errors”. In: *Proceedings of the 16th International Conference on Software Technologies - ICSOFT,* INSTICC. SciTePress, 2021, pp. 177–184. ISBN: 978-989-758-523-4. DOI: 10.5220/0010551301770184.

[58] Paul Kocher et al. “Spectre attacks: exploiting speculative execution”. In: *Commun. ACM* 63.7 (June 2020), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/3399742. URL: https://doi.org/10.1145/3399742.

[59] Nikolai Kosmatov and Julien Signoles. “A Lesson on Runtime Assertion Checking with Frama-C”. In: *4th International Conference Runtime Verification RV 2013, France, September 24-27, 2013. Proceedings*. Vol. 8174. Lecture Notes in Computer Science. Springer, 2013, pp. 386–399. DOI: 10.1007/978-3-642-40787-1_29.

[60] Daniel Kroening et al. “Loop summarization using state and transition invariants”. In: *Form. Methods Syst. Des.* 42.3 (June 2013), pp. 221–261. ISSN: 0925-9856. DOI: 10.1007/s10703-012-0176-y. URL: https://doi.org/10.1007/s10703-012-0176-y.

[61] Melina Kulenovic and Dzenana Donko. “A survey of static code analysis methods for security vulnerabilities detection”. In: *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2014, pp. 1381–1386. DOI: 10.1109/MIPRO.2014.6859783.

[62] Matthieu Lemerre. “SSA Translation Is an Abstract Interpretation”. In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571258. URL: https://doi.org/10.1145/3571258.

[63] Guangpu Li et al. “Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 162–180.

[64] Moritz Lipp et al. “Meltdown: reading kernel memory from user space”. In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC’18. Baltimore, MD, USA: USENIX Association, 2018, pp. 973–990. ISBN: 9781931971461.

[65] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. “An empirical study on the effectiveness of static C code analyzers for vulnerability detection”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. ISBN: 9781450393799. DOI: 10.1145/3533767.3534380. URL: https://doi.org/10.1145/3533767.3534380.

[66] Paul Lokuciejewski et al. “A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models”. In: *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’09. USA: IEEE Computer Society, 2009, pp. 136–146. ISBN: 9780769535760. DOI: 10.1109/CGO.2009.17. URL: https://doi.org/10.1109/CGO.2009.17.

[67] Sanoop Mallissery and Yu-Sung Wu. “Demystify the fuzzing methods: A comprehensive survey”. In: *ACM Computing Surveys* 56.3 (2023), pp. 1–38.

[68] Valentin JM Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2312–2331.

[69] Hector Marco-Gisbert and Ismael Ripoll Ripoll. “Address Space Layout Randomization Next Generation”. In: *Applied Sciences* 9.14 (2019). ISSN: 2076-3417. DOI: 10.3390/app9142928. URL: https://www.mdpi.com/2076-3417/9/14/2928.

[70] Horiba Mira. *MISRA C:2012 Amendment 2*. Tech. rep. Updates for ISO/IEC 9899:2011 Core functionality. 2020. URL: https://misra.org.uk/app/uploads/2021/06/MISRA-C-2012-AMD2.pdf.

[71] David Monniaux. “A Survey of Satisfiability Modulo Theory”. In: *Computer Algebra in Scientific Computing*. Ed. by Wolfram Gerdt Vladimir P.and Koepf and Evgenii V. Seiler Werner M.and Vorozhtsov. Cham: Springer International Publishing, 2016, pp. 401–425. ISBN: 978-3-319-45641-6.

[72] M. Saqib Nawaz et al. “A Survey on Theorem Provers in Formal Methods”. In: *ArXiv* abs/1912.03028 (2019). URL: https://api.semanticscholar.org/CorpusID:208858384.

[73] Ya Pan et al. “A Systematic Literature Review of Android Malware Detection Using Static Analysis”. In: *IEEE Access* 8 (2020), pp. 116363–116379. DOI: 10.1109/ACCESS.2020.3002842.

[74] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SYMCC: don’t interpret, compile!” In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC’20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[75] Michele Portolan. “Conception d’un système embarqué sûr et sécurisé”. In: 2006. URL: https://api.semanticscholar.org/CorpusID:190534994.

[76] Herbert Prähofer et al. “Opportunities and challenges of static code analysis of IEC 61131-3 programs”. In: *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. 2012, pp. 1–8. DOI: 10.1109/ETFA.2012.6489535.

[77] Rabéa Ameur-Boulif Salim Yahia Kissi and Yassamine Seladji. “Identifying Security Vulnerabilities in Source Code with Safety Verification”. In: *International Journal of Critical Computer-Based Systems* (2025). Accepted for publication, pp. 1–26. DOI: 10.1504/IJCCBS.2025.10075749.

[78] Piyumika Samarasekara, Ridmi Hettiarachchi, et al. “A Comparative Analysis of Static and Dynamic Code Analysis Techniques”. In: *Authorea Preprints* (2023).

[79] *SEI CERT C Coding Standard*. Tech. rep. [Accessed 10-02-2024]. Software Engineering Institute, Carnegie Mellon University, 2021. URL: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[80] Yan Shoshitaishvili et al. “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[81] João P. Marques Silva and Karem A. Sakallah. “GRASP—a new search algorithm for satisfiability”. In: ICCAD ’96. San Jose, California, USA: IEEE Computer Society, 1997, pp. 220–227. ISBN: 0818675977.

[82] Laurent Simon, David Chisnall, and Ross Anderson. “What You Get is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 1–15. DOI: 10.1109/EuroSP.2018.00009.

[83] Nick Stephens et al. “Driller: Augmenting fuzzing through selective symbolic execution.” In: *NDSS*. Vol. 16. 2016. 2016, pp. 1–16.

[84] Thomas Sutter et al. “Dynamic Security Analysis on Android: A Systematic Literature Review”. In: *IEEE Access* 12 (2024), pp. 57261–57287. DOI: 10.1109/ACCESS.2024.3390612.

[85] Andrew S. Tanenbaum and Herbert J. Bos. *Modern Operating Systems*. 4th. Pearson, 2015.

[86] Texas Instruments. *TMS320C28x CPU and Instruction Set, Reference Guide*. 2015. URL: https://www.ti.com/lit/ug/spru430f/spru430f.pdf?ts=1707819377802.

[87] Texas Instruments. *TMS320C28x Optimizing C/C++ Compiler v22.6.0.LTS, User’s Guide*. Tech. rep. Texas Instruments, 2023. URL: https://www.ti.com/lit/ug/spru514z/spru514z.pdf?ts=1707582792919.

[88] Francesc Mateo Tudela et al. “On Combining Static, Dynamic and Interactive Analysis Security Testing Tools to Improve OWASP Top Ten Security Vulnerability Detection in Web Applications”. In: *Applied Sciences* (2020). URL: https://api.semanticscholar.org/CorpusID:234536018.

[89] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. “Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction”. In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. 2017, pp. 45–52. DOI: 10.1109/ICTAI.2017.00019.

[90] Junjie Wang et al. “Software Testing With Large Language Models: Survey, Landscape, and Vision”. In: *IEEE Trans. Softw. Eng.* 50.4 (Apr. 2024), pp. 911–936. ISSN: 0098-5589. DOI: 10.1109/TSE.2024.3368208. URL: https://doi.org/10.1109/TSE.2024.3368208.

[91] Xi Wang et al. “Improving Integer Security for Systems with KINT”. In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 163–177. ISBN: 978-1-931971-96-6. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/wang.

[92] Xie Xiaofei. “Static loop analysis and its applications”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 1130–1132. ISBN: 9781450342186. DOI: 10.1145/2950290.2983972. URL: https://doi.org/10.1145/2950290.2983972.

[93] Jianhao Xu et al. “Silent bugs matter: a study of compiler-introduced security bugs”. In: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC ’23. Anaheim, CA, USA: USENIX Association, 2023. ISBN: 978-1-939133-37-3.

[94] Mingwei Zhang and R. Sekar. “Control flow integrity for COTS binaries”. In: *Proceedings of the 22nd USENIX Conference on Security*. SEC’13. Washington, D.C.: USENIX Association, 2013, pp. 337–352. ISBN: 9781931971034.

[95] Zeineb Zhioua, Stuart Short, and Yves Roudier. “Static Code Analysis for Software Security Verification: Problems and Approaches”. In: *2014 IEEE 38th International Computer Software and Applications Conference Workshops*. 2014, pp. 102–109. DOI: 10.1109/COMPSACW.2014.22.

Webographie

[1] *Affected Processors: Transient Execution Attacks & Related Security... — intel.com*. https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html. [Accessed 13-03-2025].

[23] *CERN Computer Security*. https://security.web.cern.ch. [Accessed 10-02-2024].

[24] Julian Cohen. *Contemporary Automatic Program Analysis (Slide 4: Automatic Program Analysis)*. Presentation at Black Hat USA 2014, August 2–7, Mandalay Bay Las Vegas, NV. 2014. URL: http://blackhat.com/docs/us-14/materials/us-14-Cohen-Comtemporary-Automatic-Program-Analysis.pdf (visited on 08/23/2025).

[27] cpredef contributors. *Data Models*. [Accessed 17-09-2024]. GitHub. Mar. 2024. URL: https://github.com/cpredef/predef/blob/master/DataModels.md (visited on 09/13/2025).

[29] CWE. "Common Weakness Enumeration". In: (2021). [Accessed 10-02-2024]. URL: https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[34] CVE details. *Évolution des vulnérabilités par type (Overflow et Corruption de la mémoire) depuis 2015*. https://www.cvedetails.com/. [Accessed 02-02-2025]. 2025.

[37] Blane Erwin. *The Groundbreaking 2015 Jeep Hack Changed Automotive Cybersecurity — fractionalciso.com*. https://fractionalciso.com/the-groundbreaking-2015-jeep-hack-changed-automotive-cybersecurity/. [Accessed 12-03-2025].

[70] Horiba Mira. *MISRA C:2012 Amendment 2*. Tech. rep. Updates for ISO/IEC 9899:2011 Core functionality. 2020. URL: https://misra.org.uk/app/uploads/2021/06/MISRA-C-2012-AMD2.pdf.

[79] *SEI CERT C Coding Standard*. Tech. rep. [Accessed 10-02-2024]. Software Engineering Institute, Carnegie Mellon University, 2021. URL: https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[86] Texas Instruments. *TMS320C28x CPU and Instruction Set, Reference Guide*. 2015. URL: https://www.ti.com/lit/ug/spru430f/spru430f.pdf?ts=1707819377802.