HAL
open science

Compilation vérifiée et sécurisée contre les canaux cachés temporels

Rémi Hutin

To cite this version:

Rémi Hutin. Compilation vérifiée et sécurisée contre les canaux cachés temporels. Cryptographie et sécurité [cs.CR]. École normale supérieure de Rennes, 2021. Français. NNT : 2021ENSR0029 . tel-03616445

HAL Id: tel-03616445

https://theses.hal.science/tel-03616445v1

Submitted on 22 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'ÉCOLE NORMALE SUPÉRIEURE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Rémi HUTIN

Verified Secure Compilation against Timing Side-Channels

Thèse présentée et soutenue à l'Ecole normale supérieure de Rennes, le 1er décembre 2021
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Mme Tamara REZK	Directrice de Recherche – Inria Sophia Antipolis
M. Alejandro RUSSO	Professeur – Chalmers University of Technology

Composition du Jury :

Président :	M. David BAELDE	Professeur des universités – ENS Rennes
Examinateurs :	Mme Tamara REZK	Directrice de Recherche – Inria Sophia Antipolis
	M. Alejandro RUSSO	Professeur – Chalmers University of Technology
	M. Marco VASSENA	Chercheur – CISPA Stanford Center for Cybersecurity
Directrice de thèse :	Mme Sandrine BLAZY	Professeur des universités – Université de Rennes 1
Co-encadrant. de thèse :	M. David PICHARDIE	Chercheur – Facebook Research

Résumé en français

Dans notre société, les ordinateurs sont des outils de communication essentiels et sont devenus, au fil des années, fondamentaux dans de nombreux domaines, aussi bien dans la vie quotidienne que dans l'industrie, la finance, les transports, etc. Pourtant, comme ces systèmes sont développés par des humains, ils sont sujets à des erreurs de conception, généralement appelées *bugs*. Selon le contexte, un bug peut être considéré comme une gêne mineure ou comme un problème majeur. En effet, lorsque des vies humaines, des infrastructures coûteuses ou des données privées sont en danger, les bugs peuvent avoir des conséquences dramatiques, que nous illustrons par quelques exemples. En 1996, un bug dans le logiciel du lanceur de la fusée Ariane 5 a produit un débordement d'entier dans le système informatique, qui a finalement conduit à la destruction de la fusée [Ben01]. En 2014, un rapport a mis en évidence des erreurs de programmation dans les voitures Toyota, qui sont soupçonnées d'être responsables de dizaines de décès [Koo14]. En 2002, un autre rapport s'est intéressé au coût global des bugs logiciels dans l'industrie. Ils ont été estimés à près de 60 milliards de dollars par an pour l'économie américaine [Pla02]. Enfin, les bugs peuvent également entraîner des menaces pour la sécurité, en rendant les programmes vulnérables à des attaques. Récemment, la pandémie de COVID-19 a provoqué une augmentation sans précédent des cyber-attaques, ciblant à la fois les citoyens travaillant à domicile et les infrastructures nationales de soins de santé [Aya16].

Comme les conséquences des erreurs dans les logiciels critiques peuvent être dramatiques, cela motive l'utilisation des *méthodes formelles*. Les méthodes formelles consistent en un ensemble de techniques mathématiques, dont la *sémantique formelle*, permettant de raisonner rigoureusement sur l'exécution d'un programme dans un langage donné. Le but de ces méthodes est d'obtenir des garanties formelles sur un programme, en particulier qu'il soit *sûr*, c'est-à-dire que son exécution ne donne jamais lieu à un bug.

Afin de raisonner sur un programme, on utilise la *sémantique formelle* pour définir précisément le comportement d'un programme. Pour la plupart des langages de programmation, la sémantique du langage est définie en utilisant le langage naturel, qui peut parfois être ambigu. Au contraire, une sémantique formelle est un objet mathématique qui décrit rigoureusement l'exécution d'un programme, et ne laisse donc aucune place à ces ambiguités.

Les méthodes formelles sont généralement appliquées au niveau du langage source, c'est-à-dire sur un programme lisible par l'homme, écrit dans un langage tel que le C. Or, l'objectif de ces méthodes est d'obtenir des garanties sur le programme réellement exécuté, c'est-à-dire le code machine produit après sa compilation. Deux options apparaissent

dans cette situation. Premièrement, nous pouvons effectuer des analyses statiques sur le programme compilé. Cependant, comme de nombreuses abstractions (les fonctions, les instructions de contrôle de flux comme les boucles, les types, etc.) sont perdues après la compilation, le raisonnement sur le code machine est beaucoup plus difficile que sur un programme source.

L'autre possibilité consiste à établir des propriétés sur un programme au niveau source, puis à s'appuyer sur la *compilation formellement vérifiée* pour s'assurer que les garanties sont préservées par la compilation. Nous nous concentrons sur cette deuxième option dans cette thèse.

Compilation formellement vérifiée. Le but d'un compilateur est de transformer un programme écrit dans un langage *source* S en un langage *cible* T. Par exemple, un compilateur C typique traduit un programme C en code assembleur. Les compilateurs sont aussi généralement des programmes volumineux et complexes, qui mettent en œuvre plusieurs traductions et optimisations. Comme tout autre programme, ils sont sujets à des bugs, menant à la production d'un code incorrect par le compilateur.

Même les compilateurs populaires et largement testés, tels que LLVM ou GCC, ne peuvent être considérés comme exempts de bugs. Par exemple, en 2011, une étude empirique a mis en évidence plusieurs bugs dans ces compilateurs [Yan+11]. Ces bugs incluent soit la production d'un code incorrect, soit un crash du compilateur lors de la compilation d'un code source valide. Ces bugs peuvent être considérés comme négligeables par rapport aux bugs potentiels présents dans le programme source. Cependant, la possibilité que des bugs soient introduits pendant la compilation ne peut être ignorée dans le cas de logiciels critiques, pour lesquels la sûreté est essentielle.

En outre, de nombreux outils d'analyse statique existent pour raisonner sur les propriétés de sûreté indécidables d'un programme. Par exemple, l'absence d'erreur d'exécution (pas d'accès hors limites, pas de division par zéro, etc.) peut être établie pour un logiciel en utilisant ces outils. Cependant, comme la plupart de ces outils ne raisonnent typiquement que sur le code source, il n'y a aucune assurance que ces garanties soient préservées par le compilateur.

L'objectif de la compilation formellement vérifiée est de résoudre ce problème. Un compilateur formellement vérifié est un compilateur qui est accompagné d'une preuve de correction. Intuitivement, la correction du compilateur indique que le programme compilé a le même comportement que le programme source, en ce qui concerne la sémantique formelle des langages source et cible. Par conséquent, toute propriété de sûreté énoncée sur le programme source est garantie d'être préservée sur le code cible produit par le compilateur.

Dans la littérature, plusieurs projets ont abordé avec succès le problème de la compila-

tion formellement vérifiée. Ces projets incluent CakeML [Kum+14], qui compile depuis un sous-ensemble de ML vers du code assembleur, Vellvm [Zha+12], un outil permettant de raisonner formellement sur la représentation intermédiaire et les optimisations de LLVM, ou encorele compilateur CompCert [Ler09; Ler+12]. Dans cette thèse, certaines de nos contributions reposent sur le compilateur CompCert.

Un fossé entre la sûreté et la sécurité. Nous avons discuté du besoin de sûreté dans le cas de logiciels critiques, où les bugs peuvent avoir des conséquences dévastatrices. Cependant, dans certains cas, la sûreté n'est pas suffisante. Même un code correct et sûr peut être exploité par des attaquants malveillants, généralement pour apprendre des informations censées rester secrètes. Le domaine de la *cryptographie* est particulièrement vulnérable aux failles de sécurité car le code cryptographique manipule des données sensibles telles que des clés privées, et ces données doivent rester secrètes à tout prix.

Cependant, l'impact du compilateur sur la sécurité d'un programme est une question pertinente. En effet, même si on qu'un programme source est sécurisé, la compilation peut produire code non sécurisé. On dit dans ce cas que la compilation ne préserve pas la sécurité du programme. Ce problème vient du fait que les menaces de sécurité peuvent provenir d'éléments qui ne sont pas capturés par la sémantique formelle du langage source considéré par le compilateur, et ne peuvent donc pas être spécifiés comme une propriété de sûreté. Contrairement au problème de la préservation de la sûreté, qui est une conséquence directe de la correction d'un compilateur, la préservation de la sécurité ne peut pas être directement déduite. Cet écart, qui est appelé l'écart correction-sécurité, est largement discuté dans [DPS15].

Le domaine de la *compilation sécurisée* vise à étudier les compilateurs préservant les politiques de sécurité. Avec un tel outil, il est possible d'utiliser l'analyse statique pour établir une propriété de sécurité au niveau d'un programme source, puis de se fier au compilateur préservant la sécurité pour préserver la politique de sécurité tout au long de la compilation. Dans cette thèse, nous nous concentrons sur l'étude de la préservation des contre-mesures contre les attaques dites par canaux cachés temporels.

Attaques par canaux cachés temporels. Les attaques par *canaux cachés* sont une classe de menaces de sécurité, résultant du fait qu'un logiciel est exécuté sur un ordinateur, qui interagit nécessairement avec son environ nement physique. L'observation de ces interactions peut permettre de découvrir des informations sensibles censées rester secrètes. Ces interactions se présentent sous de nombreuses formes, mais les plus courantes sont la puissance consommée pendant l'exécution d'un programme, le temps d'exécution d'un programme, l'émanation sonore ou électromagnétique produite lors d'une exécution, etc.

Le domaine de la cryptographie est particulièrement vulnérable aux attaques par

canaux cachés temporels, c'est-à-dire les attaques exploitant le temps d'exécution d'un programme. De nombreux exemples d'attaques temporelles réelles montrent comment de très petites variations du temps d'exécution peuvent être utilisées pour récupérer partiellement des clés secrètes de primitives cryptographiques. Ces attaques sont relativement simples à réaliser, et peuvent également être exécutées à distance, sur un réseau, tant que les variations sont assez importantes pour être remarquées [Ber05].

Les variations du temps d'exécution peuvent provenir de différents types de constructions de programmes. Toute branche (instructions if, boucles, etc.) est susceptible d'introduire une variation temporelle qui dépend des conditions testées. De même, tout accès à la mémoire est susceptible d'introduire une variation temporelle, en raison des mécanismes de cache existant sur la plupart des architectures informatiques modernes [Ber05; YGH17].

Afin de contrer les attaques par canaux cachés temporels, les experts en cryptographie ont développé une politique de sécurité populaire, appelée *cryptographic constant-time*, ou plus simplement *constant-time*. Il s'agit d'une discipline de programmation qui impose de fortes restrictions sur le code des primitives cryptographiques. Notons que le nom de cette politique peut être trompeur, car cette politique ne considère aucune notion de temps d'exécution réel. Au lieu de cela, la politique stipule que le flux de contrôle du programme, et la liste des d'accès à la mémoire effectués pendant une exécution doivent être indépendants des données secrètes manipulées par le programme. En termes plus simples, si nous considérons une variable secrète appelée `secret`, tout programme contenant un branchement conditionnel dépendant de cette variable (`if (secret)`), ou un accès mémoire dépendant de cette variable (`array[secret]`) serait rejeté par la politique constant-time. La politique constant-time a été utilisée avec succès pour implémenter de nombreuses bibliothèques cryptographiques populaires, telles que Curve25519 [Ber06; Lan15], TEA [WN94], NaCl [BLS12], mbedTLS [ARM16], ou OpenSSL [Ope19].

Une autre politique également utilisée dans des implémentations cryptographiques est la politique *constant-resource* [Wu+18; Aga00; Ngo+17]. Cette politique est plus relachée que la politique constant-time, car elle impose moins de contraintes sur les programmes pour qu'ils soient considérés comme sécurisés. On considère ici un modèle de consommation de ressources pendant une exécution, qui peut modéliser le temps d'exécution par exemple. Un programme sécurisé peut contenir des branchements dépendants de variables secrètes, tant que les deux branches sont équilibrées vis-à-vis du modèle de consommation de ressources.

Contenu de la thèse. Dans cette thèse, nous présentons principalement les deux contibutions suivantes.

Dans un premier temps, nous nous intéressons à la politique constant-time, et à sa

préservation lors de la compilation au sein du compilateur CompCert. Nos travaux se concentrent uniquement sur les modifications que nous avons apportées à ce compilateur afin qu'il préserve cette politique de sécurité.

Dans un second temps, nous nous intéressons à la politique constant-resource. Au meilleur de nos connaissances, aucuns travaux n'avaient eu pour but de s'intéresser à la préservation de cette politique lors d'une transformation de programme. C'est le défi que nous relevons dans cette thèse. Plus précisement, nous proposons une politique plus flexible et des techniques de preuves permettant de démontrer la préservation de cette politique lors d'une transformation de programme. Enfin, nous appliquons ces méthodes sur des optimisations de programmes classiques.

Tous les résultats présentés dans cette thèse sont prouvés avec l'aide de l'assistant de preuve Coq.

Table of Contents

1 Introduction **17**

2 Enforcing Non-Interference with Typing **31**

2.1 Definition of a Language . . . 33
2.1.1 Syntax of $\mathscr{L}$ . . . 33
2.1.2 Big-step Semantics for Expressions of $\mathscr{L}$ . . . 34
2.1.3 Small-step Semantics with Continuations . . . 35
2.1.4 Definition of Non-Interference . . . 37

2.2 Secure Flow Type-System . . . 38
2.2.1 Type-system for Expressions . . . 39
2.2.2 Type-system for Statements . . . 40

2.3 Soundness of the Type-System . . . 41
2.3.1 Generalization to Continuation . . . 42
2.3.2 Indistinguishability Lemmas . . . 42
2.3.3 The Unobservability Relation . . . 44
2.3.4 Reasoning on Prefixes of Executions . . . 46
2.3.5 Proof of Type Soundness . . . 47

2.4 Type Preserving Compilation . . . 49
2.4.1 A Problematic Transformation: Array Concatenation . . . 51
2.4.2 Limitations of the Type-system . . . 52

2.5 Extension to Observational Non-Interference . . . 55
2.5.1 Instrumenting a Semantics with Leakages . . . 55
2.5.2 Defining the Constant-Time Policy as an Instance of ONI . . . 57
2.5.3 Enforcing CCT with a Type-System . . . 59
2.5.4 Preservation of ONI . . . 61

2.6 Conclusion . . . 61

3 A Constant-Time Preserving CompCert Compiler **63**

3.1 Examples of Standard Compilers Breaking the Constant-time Policy 64
3.1.1 Optimization Breaking Constant-time 64
3.1.2 Floating Point Value Conversion 65
3.1.3 Conclusion 66
3.2 A Constant-time Selection Operation 67
3.3 Detailed Background on the CompCert Compiler 68
3.3.1 Architecture of CompCert 68
3.3.2 x86 Operations in CompCert 71
3.4 CCT-preservation Breach in CompCert 75
3.4.1 Non-preservation of the Constant-time Policy by the Instruction Selection Pass 75
3.5 Modification of the CompCert Compiler 76
3.5.1 Modification of the Set of Operations of the Backend Compiler . . . 78
3.5.2 Modification of the Assembly Generation Pass 79
3.5.3 Modification of the Instruction Selection Pass 81
3.6 Integration to CompCert 3.6 82
3.7 Experimental Evaluation 82
3.8 Conclusion 85

4 Constant-Resource Policy 87

4.1 An Introduction to CR-security 89
4.1.1 Example: Common Subexpression Elimination 89
4.1.2 Motivation of the CR$^{\#}$ policy 92
4.2 Comparison between the CCT and CR Policies 93
4.2.1 Language Definition 93
4.2.2 Semantic Properties of $\mathcal{L}$ 96
4.2.3 Definition of the CCT and CR Policies 96
4.3 Our Approach using a Syntactic Annotation 98
4.3.1 CR$^{\#}$, a more Flexible Policy 98
4.3.2 Implementation of Control-flow Preserving Transformations 103
4.3.3 Using Leakage Preservation to prove CR$^{\#}$ preservation 104
4.3.4 A Cornerstone non Leakage-Preserving Transformation 107
4.3.5 Enforcing CR$^{\#}$ with a Type-System 111
4.4 Related Work 113

4.5 Conclusion 114

5 Conclusion **115**

5.1 Summary 115

5.2 Perspectives 117

5.2.1 Experimental Validation 117

5.2.2 Extending the Atomic Annotation to Memory Accesses 118

5.2.3 Spectre Vulnerabilities 119

Author's contributions **121**

Bibliography **123**

List of Figures

1.1 Example of program with secret-dependent loop and padding. 21
1.2 Two examples of non constant-time programs 22

2.1 Definition workflow of 3 instances of ONI, using 3 semantics. 56
2.2 Definition workflow of 3 instances of ONI, using 1 semantics and projection functions. 57

3.1 A C function multiplying a boolean and an integer. 65
3.2 Assembly code generated by Clang. 65
3.3 Conversions between float and unsigned integer values. 66
3.4 Assembly code generated by GCC `-O1`. 66
3.5 Implementations of constant-time selection at source level. 67
3.6 A low-level implementation of `r ← ctselect(cond, r1, r2)`. 68
3.7 The CompCert compilation chain. 69
3.9 Semantics of x86 operations in CompCert. 74
3.10 Semantics of x86 operations in CompCert. 74
3.11 Examples (in `C`-like syntax) of conditional branches introduced by CompCert. 77
3.12 New selection operation. 78
3.13 Semantics of x86 operations in CompCert. 80
3.14 Constant-time implementations (in `C`-like syntax) of the examples of Figure 3.11. 83
3.15 Relative execution times of our benchmark. 84

4.1 Examples of CR-secure programs 88
4.2 Example of branching programs 90

List of Theorems

2.1 Definition (Syntax of $\mathscr{L}$) 33
2.2 Definition (Set of values $\mathcal{V}$) 34
2.3 Definition (Evaluation of expression $\langle e, \sigma\rangle \Downarrow v$) 34
2.4 Definition (Step relation for statements $\langle \sigma_1, k_1\rangle \longmapsto \langle \sigma_2, k_2\rangle$) 36
2.5 Definition (Indistinguishability) 38
2.6 Definition (Non-interference) 38
2.7 Definition (Secure Flow type-system for expressions $\Gamma \vdash e : \tau$) 39
2.8 Definition (Secure Flow type-system for statements $\Gamma \vdash s : \tau$) 40
2.1 Theorem (Type Soundness) 41
2.9 Definition (Secure Flow type-system for continuations $\Gamma \vdash k : \tau$) 42
2.10 Definition (Non-interference with continuation) 42
2.2 Lemma (Observable expression) 43
2.3 Lemma (Locally respects) 43
2.11 Definition (Unobservable continuation) 44
2.4 Lemma (`unobservable` preservation) 45
2.5 Lemma (Indistinguishability of `unobservable` step) 45
2.6 Lemma (`unobservable` branching) 46
2.7 Lemma (Bottleneck) 47
2.8 Theorem (Type Soundness) 47
2.12 Definition (Typability-preserving transformation) 50
2.13 Definition (Array concatenation transformation) 51
2.14 Definition (Concatenation of environments) 52
2.9 Lemma (Transformation correct) 53
2.10 Theorem (The concatenation transformation is not type preserving) 53
2.15 Definition (Observational Non-Interference (ONI)) 55
2.16 Definition (Observational Non-Interference (ONI)) 56
2.17 Definition (Set of event $\mathcal{E}$) 58
2.18 Definition (Evaluation of expression $\langle e, \sigma\rangle \Downarrow (v, \ell)$) 58
2.19 Definition (Evaluation of statement $\langle s, \sigma\rangle \Downarrow (\sigma', \ell)$) 58
2.20 Definition (Constant-time type-system for expressions $\Gamma \vdash e : \tau$) 60
2.21 Definition (Constant-time type-system for statements $\Gamma \vdash s : \tau$) 60

2.11 Theorem (Soundness of the Constant-time type-system) 60
2.22 Definition (Preservation of ONI) 61

4.1 Definition (CR-security) 92
4.2 Definition (Extended syntax of $\mathscr{L}$) 94
4.3 Definition (Set of event $\mathcal{E}$) 94
4.4 Definition (Evaluation of expression $\langle e, \sigma\rangle \Downarrow (v, \ell)$) 94
4.5 Definition (Evaluation of statement $\langle s, \sigma\rangle \Downarrow (\sigma', \ell)$) 95
4.1 Lemma (Determinism) 96
4.2 Lemma (Non-cancellation) 96
4.3 Lemma (CCT leakage implies CR leakage) 97
4.4 Theorem (CCT implies CR) 98
4.6 Definition (Extended syntax of $\mathscr{L}$) 99
4.7 Definition (Set of event $\mathcal{E}$) 99
4.8 Definition (Semantics of an *atomic* annotation) 100
4.9 Definition (CRplus-security) 100
4.10 Definition (CR$^{\#}$ preservation) 102
4.11 Definition (Leakage preservation) 105
4.5 Lemma ($\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ leakage preserving) 105
4.6 Theorem (Leakage preservation implies ONI preservation) 105
4.7 Theorem (Elementary transformations are CR$^{\#}$-preserving) 105
4.12 Definition (Transformation $\mathcal{N}$) 106
4.8 Lemma ($\mathcal{N}$ is leakage preserving) 107
4.9 Theorem ($\mathcal{N}$ is CR$^{\#}$-preserving) 107
4.13 Definition (Transformation $\mathcal{D}$) 107
4.14 Definition (Leakage Preservation with Offset (LPO)) 108
4.15 Definition (Termination preservation) 109
4.10 Theorem (LPO implies ONI preservation) 109
4.11 Lemma ($\mathcal{D}$ is termination preserving) 110
4.12 Lemma ($\mathcal{D}$ is LPO) 110
4.13 Theorem ($\mathcal{D}$ is CR$^{\#}$-preserving) 111
4.16 Definition (CR$^{\#}$ type-system for expressions $\Gamma \vdash e : \tau, q$) 111
4.17 Definition (CR$^{\#}$ type-system for statements $\Gamma, s \vdash s^{\#} : \tau, q$) 112
4.14 Theorem (Soundness of the CR$^{\#}$ type-system) 113

Acronyms

CCT Cryptographic Constant-Time. 55, 57, 59–61, 96–98, 101, 111

CR Constant-Resource. 26, 29, 30, 61, 89–93, 96–98, 102, 103, 113, 115–118

CSE Common Subexpression Elimination. 89–91, 103, 104

LPO Leakage Preservaion with Offset. 108–110

NI Non-Interference. 23–26, 29, 31–33, 38, 41, 42, 44, 49, 50, 53, 55, 61, 115

ONI Observational Non-Interference. 14, 26, 31, 33, 55–57, 59, 61, 62, 96, 97, 100, 109, 115–117

Chapter 1

Introduction

In our society, computers are essential communication tools and have become fundamental over the years in many different fields, including daily life as well as industry, banking, transportation, etc. Yet, as these systems are developed by humans, they are prone to errors in their conception, usually referred to as *bugs*. Depending on the context, a bug may be considered as a minor annoyance or as a major issue. Indeed, when humans lives, costly infrastructures or private data are at risk, bugs may have dramatic consequences, that we illustrate with a few examples. In 1996, a bug in the software of the Ariane 5 rocket launcher produced an integer overflow in the computer system, which ultimately led to the destruction of the rocket [Ben01]. In 2014, a report highlighted programming errors in Toyota cars, that are suspected to be responsible for tens of deaths [Koo14]. In 2002, another report focused on the overall cost of software bugs in the industry. They were estimated to cost almost 60 billions dollars per year to the U.S. economy [Pla02]. Last, bugs may also lead to security threats, by allowing an attacker to perform attacks exploiting vulnerabilities in a program. Recently, the COVID-19 pandemic has created an unprecedented raise in cyber-attacks, targeting both working at home citizens and national healthcare infrastructures [Aya16].

As the aftermath of errors in critical software can be dramatic, the motivation to use *formal methods* rises. Formal methods consist of a set of mathematical techniques, including *formal semantics*, allowing to rigorously reason on the execution of a program in a given language. The goal of these methods is to obtain formal guarantees on a program, such as ensuring that a program is *safe*, i.e., that its execution never results in a bug.

In order to reason on a program, *formal semantics* are used to precisely define the behavior of a program. For most real-life programming languages, the semantics of the language is defined using natural language, which may sometimes be ambiguous. Instead, a formal semantics is a mathematical object that rigorously describes the execution of a program.

Formal methods can be enforced by *formal verification*, which is the action of mechanically proving a *property* of a software, with respect to a *specification*. *Proof assistants*, such as Coq [19], Isabelle/HOL [NPW02] or ACL2 [KMM13], are tools that can check the validity of a mathematical reasoning. Using a proof assistant to verify a proof gives a high level of confidence, as only the correctness of the proof assistant needs to be trusted.

Formal methods also include the use of *static analyzers*, which are tools whose goal is to verify whether or not a given property is satisfied by a program, for any set of input. In the general case, this is an undecidable problem, according to Rice's theorem [Ric53]. Several static analyzers have been successfully developed and even used in industrial contexts. We highlight here some examples. Astrée [DS07] is a static analyzer, whose goal is to show the absence of run-time errors in C programs. It has been used to this end on several instances of Airbus embedded software. Verasco [Jou+15] is another instance of static analyzer aiming at proving the absence of run-time error in a program. Its most distinguishing trait is the fact that it is implemented and proved using the Coq proof assistant.

These methods are typically applied at a source-level language, i.e., on a human-readable program, written in a language such as C. Yet, the goal of these methods is to obtain guarantees on the program actually being executed, i.e., the machine code produced after its compilation. Two options appear in this situation. First, we can perform static analyses on the compiled program. However, as many abstractions (e.g., functions, control flow constructs such as loops, types, etc) are lost after the compilation, reasoning on machine code is much harder than on a source program.

The other possibility consists in establishing properties on a program at source level, and then relying on *formally verified compilation* to ensure that the guarantees are preserved by the compilation. We focus on this second option in this thesis.

Formally Verified Compilation. The goal of a compiler is to transform a program written in a *source* language S into a *target* language T. As an example, a typical C compiler translates a C program into assembly code. Compilers are also usually large and complex programs, implementing several translations and optimizations. As any other programs, they are prone to bugs, leading to a compiler producing incorrect code.

Even popular and extensively tested compilers, such as LLVM or GCC, cannot be considered as bug free. For example, in 2011, an empirical study highlighted several bugs in these compilers [Yan+11]. These bugs include either the production of incorrect code or a crash of the compiler while compiling a valid source code. These bugs may be considered as negligible when compared to potential bugs present in the source program. However, the possibility of bugs being introduced during the compilation cannot be ignored in the case of critical software, for which the safety is essential.

In addition, many static analysis tools exists to reason on undecidable safety properties of a program. For example, the absence of run-time error (no out-of-bound access, no division by zero, etc) may be established for a software by using these tools. However, as most of these tools typically reason on source code only, there is no insurance that these guarantees are preserved by the compiler.

The goal of formally verified compilation is to solve this issue. A formally verified compiler is a compiler that comes with a proof of correctness. Intuitively, the correctness of the compiler states that the compiled program has the same behavior as the source program, with respect to the formal semantics of the source and target languages. Therefore, any safety property stated on the source program is guaranteed to be preserved on the target code produced by the compiler.

In the literature, several projects have successfully tackled the problem of formally verified compilation. These projects include CakeML [Kum+14], compiling a subset of ML to assembly, Vellvm [Zha+12], a framework allowing to formally reason on LLVM's intermediate representation and optimizations, or the CompCert compiler [Ler09; Ler+12]. In this thesis, some of our contributions rely on the CompCert compiler, that we detail further.

The CompCert Compiler. The CompCert compiler is moderately optimizing compiler, developed for critical embedded software, compiling source programs written in (a large subset of) C language, down to assembly code. Supported target architectures include x86, ARM, PowerPC and RISC-V.

The CompCert compiler has been mechanically verified using the Coq proof assistant [19]. A correctness theorem has been proved, that captures the idea that if the compilation is successful, then the source program and its compiled counterpart have the same behavior for any set of input. It is stated as follows:

Theorem 1: CompCert's semantic preservation [Ler+12].

For all source programs S and compiler-generated code C, if the compiler, applied to the source S, produces the code C, without reporting a compile-time error, then the observable behavior of C is one of the possible observable behaviors of S.

In the theorem above, an observable behavior is a product of the execution of the program, and is defined through the formal semantics of the language considered. A direct corollary of **Theorem 1** shows that any safety property verified by the source program is verified by the produced program as well. It is stated as follows:

Theorem 2: CompCert's safety property preservation [Ler+12].

Let Σ be a set of acceptable behaviors, characterizing a desired safety or liveness property of the program. Assume that a source program S satisfies Σ: all possible observable behaviors of S are in Σ. Further assume that the compiler, applied to the source S, produces the code C. Then, the compiled code C satisfies Σ: the observable behavior of C is in Σ.

The underlying idea of this theorem is that for any specification, given by a user, such as the absence of errors during the execution of a program, is guaranteed to be preserved during the compilation. In other words, if the source program is proved to be safe, then the CompCert compiler carries over this property to the compiled program.

In chapter 3, we will present our first contribution, which is based on the CompCert compiler. This chapter contains a more detailed background on the implementation part of CompCert. Namely, we will detail implementation choices, such as the intermediate representations and passes of the compiler.

A Gap between Safety and Security. We discussed the need for safety in the case of critical software, where bugs may have devastating consequences. However, in some cases, safety is not sufficient. Even correct and safe code may be exploited by malicious attackers, typically to learn information supposed to remain secret. The field of *cryptography* is particularly vulnerable to security breaches, as cryptographic code manipulates sensitive data such as private keys, and this data must remain secret.

However, the impact of compiler on the security of a program is a relevant question. Indeed, a source program may be proved to be *secure* with respect to a given attacker, yet, the compilation may not preserve the security, and produce *unsecure* code. This problem comes from the fact that security threats may arise from elements that are not captured by the formal semantics of the source language considered by the compiler, and thus can not be specified as a safety property. Unlike the problem of safety-preservation (**Theorem 2**), which is a direct consequence of the correctness of a formally-verified compiler (**Theorem 1**), the preservation of security can not directly be deduced. This gap, called the correctness-security gap, is largely discussed in [DPS15], and we now illustrate it with an example.

We give an example of non-preservation of security through compilation. We consider an attacker trying to deduce secret information by measuring the execution-time of a program. This is a case *timing side-channel* attack [Ber05], where the execution-time reveals information on secret values. Intuitively, the notion of security we focus on can be expressed as: "execution-time must be independent of secret data." Respecting this property ensures that any measurement of the execution-time of a program does not allow to deduce any sensitive information. However, as execution-time is typically not specified in the semantics of a language, such property is not guaranteed to be preserved, even by a formally-verified compiler.

For example, consider the following code snippet in Figure 1.1, where we consider a secret value `n`, bounded between 0 and 32, that an attacker tries to discover. We also consider a function `update`, and for the sake of simplicity, we assume this function to have a constant execution-time that we call t. We last consider the function `dummyUpdate`, a

```
repeat(n)      { update(); }
repeat(32 - n) { dummyUpdate(); }
```

Figure 1.1 – Example of program with secret-dependent loop and padding.

function specifically designed to have the same constant execution-time t as `update`, while doing nothing.

The underlying idea of this program is that useless code (i.e., the second line of the example) has been introduced in order to keep the global execution-time constant (here, equal to $32 \times t$). Introducing such useless code is called *padding*. The general idea is to improve the security by introducing padding, which aims at *balancing* the branches of the program, so that every path of execution performs computations with similar execution-time. Note that this is a simplified example, and that reasoning on actual execution time is almost impossible. Indeed, especially on modern architectures, behaviors such as pipelining or branch-prediction make modeling the execution-time of a program a very complex task. Such architectural considerations are out of the scope of this thesis. However, the introduction of such padding can be seen as a first step towards a more secure program, and is used in real-life cryptographic implementations [Ath+18].

Yet, some issues may appear when compiling this program. An optimizing compiler would be likely to detect the second line of the program (i.e., the second loop containing padding) as dead-code, and thus be likely to simply remove it. Indeed, this code has no impact on the observable behavior of the program. Removing this assignment would increase the performance of the program, while preserving its semantics, which is what an optimizing compiler is designed to do. However, the security of the program is not preserved by this optimization. Indeed, the execution-time of the optimized program would be $\mathtt{n} \times t$, and thus this new program violates the security property, as the execution-time depends on the secret value `n`. A timing side-channel attack would then be easier to perform.

The simple example above motivates the need for *secure compilation*. Secure compilation is an emerging field, that aims at reducing the correctness-security gap. Its goal is to study compilers that take security models into account, and design compilers that either enforce or preserve security properties. In this thesis, we focus on the study of the preservation of countermeasures against timing side-channel attacks. When combined with static analysis tools able to verify that a source program is secure with respect to timing side-channel attacks [BPT19], such compilers then allow to ensure that this security property is propagated down to the compiled program.

```
if (secret) {
  /* ... */
} else {
  /* ... */
}
```

(a) Secret dependent branch

```
array[secret] = /* ... */
```

(b) Secret dependent memory access

Figure 1.2 – Two examples of non constant-time programs

Timing Side-channel Attacks. Side-channel attacks [Koc96] is a class of security threats, arising from the fact that software is executed on a computer, that necessarily interacts with the physical world. Observing these interactions may allow to discover sensitive information supposed to remain secret. Such interactions come in many forms, but the most common ones include the power consumed during the execution of a program [KJJ99; Koc+11], the execution-time of a program [Koc96], sound or even electromagnetic emanation produced during an execution, etc.

The field of cryptography is particularly vulnerable to this kind of attacks. Many examples of real-life timing attacks show how very small variations of execution-time can be used to partially recover secret keys from cryptographic primitives. These attacks are relatively simple to perform, and may also be performed remotely, over a network, as long as variations are big enough to be noticed [Koc96].

Variations in execution-time may arise from different kinds of program constructs. As we saw earlier, any branch (if statements, loops, ...) is likely to introduce a timing variation depending on the value of the condition. Similarly, any memory access is likely to introduce a timing variation, because of cache mechanisms existing on most computer architectures [Ber05; YGH17].

In order to counter timing side-channel attacks, experts in cryptography have developed a popular security policy, called *cryptographic constant-time*, or simply *constant-time* [Alm+16]. This is a programming discipline that imposes strong restrictions on the implementation of cryptographic primitives. Note that the name of this policy may be misleading, as this policy do not consider any notion of actual execution-time. Instead, the policy states that the control flow of the program, and the list of memory accesses performed during an execution must be independent from the secret data manipulated by the program. In simpler terms, if we consider a secret variable called `secret`, any program containing one of the code snippet of Figure 1.2 would not be considered to be constant-time. The constant-time policy has been successfully used to implement many popular cryptographic libraries, such as Curve25519 [Ber06; Lan15], TEA [WN94], NaCl [BLS12], mbedTLS [ARM16], or OpenSSL [Ope19]. In this thesis, our first contribution focuses on

the preservation of the constant-time policy by the CompCert compiler.

Spectre attacks [Koc+19] are a recent kind of side-channel attacks. They exploit the *speculative execution* performed by modern processors, that can speculate on the result of the calculation. For example, if a branching depends on the result of a memory read being performed, the processor may speculate on the value and proceed the execution in the speculated branch. If the speculation was correct, the execution continues normally, otherwise, the speculative execution is discarded and proceeds in the other branch. Such discarded executions are invisible from an architectural point of view (e.g., content of registers), however, they can leave a trace in the *microarchitectural state*, typically in the cache. Thereafter, a usual cache side-channel attack may leak secret data that have been wrongly stored in the cache due to a mispeculated execution.

Spectre attacks were discovered in 2018, and have been since extensively studied. Several variants have been studied, along with tools allowing to detect vulnerabilities [DBR21], or to automatically remove vulnerabilities [Vas+21]. These works rely on security policies allowing to protect software against these attacks. In particular, the *speculative constant-time* [Cau+20] policy is an adaptation of the constant-time policy that acts as a countermeasure against Spectre attacks. Spectre attacks are out of the scope of this thesis, however, we will detail works related to these attacks as a perspective for future work in our conclusion.

A main part of our contribution builds upon the constant-time policy, that we intuitively presented above. It can be formally defined as a *non-interference* policy, which we now detail.

The Non-Interference Policy. Non-Interference (NI) [GM82; GM84] is a security policy used to model a system in which users may have different *security levels* (or *security classes*). In such a system, a *low* security level user must never be allowed to access *high* security level data. The Non-Interference (NI) policy formally captures this objective. A program may manipulate data with different security levels, and a program is said to be NI-secure if its execution does not allow a user to learn information he is no supposed to have access to.

In order to define the NI policy, we first introduce the notion of security classes [Den76]. To keep the definitions simple, we use the simplest relevant classification, that consists of two distinct security classes, that we call L and H.

— Security class L, which stands for *low* security, contains all the variables of the program that can be accessed or modified without any restriction. We also say that the variables of L are *public* or have *low sensitivity*.
— Security class H, which stands for *high*, corresponds to variables whose access is restricted. The variables of H are said to be *private*, *secret* or *sensitive*.

Users of a system are then also assigned a security class as their security level. A user with security level H can access and modify any variable is the system, while a user with security level L should only be able to access and modify low sensitivity (i.e., in L) variables. The goal of non-interference is to ensure that a program does not leak the sensitive variables to public users.

Definition of Non-Interference. In order to define the NI policy, we consider a program p that manipulates variables contained in a memory. Executing program p from initial memory a leads to output memory b. Every input and output of the program are specific locations in the memory. For a memory m, we denote as m_P the memory m restricted to variables of class P. m_L represents the public variables of m, and m_H represents the secret variables of m. NI is then defined as follows:

Definition 1: Non-interference.

A program p satisfies the non-interference policy if, for any memories a and a' such that $a_L = a'_L$, executing p on a and a' respectively leads to memories b and b', such that $b_L = b'_L$. Such a program is said to be NI-secure, and written NI(p).

Definition 1 considers two executions starting from memories that have identical public variables. In other words, if these memories differ, they only differ on their secret variables. Therefore, from the perspective of a low-security user, these two executions should have identical effects on public variables, otherwise this user could learn information on secret values. This is why we expect the resulting memories to have identical public variables; the resulting secret values however may differ: this is irrelevant as the low user may not access them anyway.

Examples of Interferent Programs. We illustrate the NI policy by stating two example programs that do not satisfy the policy.

Example 1: Explicit Flow.

We consider two variables `public` $\in L$ and `secret` $\in H$. The following program does not satisfy the NI policy:

```
public = secret;
```

We elaborate why this program is interferent. We consider two input memories a and a', that share identical public variables (i.e., `public` has the same value in both a and a'). After the execution, the value of `public` is replaced with the value of `secret`,

on which we do not make any assumption. Therefore, the resulting memories are not ensured to share identical public variables.

However, all the following assignments satisfy the NI policy:

```
public = public;
secret = secret;
secret = public;
```

Indeed, assignments between variables of the same security class are always allowed, and assigning a public value in a secret variable does not present any security threat.

The example above highlights that assigning a value to a variable which has a lower security class may break the NI policy. Such an assignment is called an *explicit flow*. It is natural to forbid such an operation, as storing a secret value in a public variable makes the value available for any user. Forbidding explicit flows is a first step towards NI, but is not sufficient, as explained in this second example.

Example 2: Implicit Flow.

We consider two variables `public` $\in L$ and `secret` $\in H$. The following program must be forbidden.

```
if (secret) {
  public = 1;
} else {
  public = 0;
}
```

Indeed, a public user could deduce the truth value of `secret` by observing `public`. Indeed, `secret` evaluates to true when `public = 1`, and `secret` evaluates to false when `public = 0`. More precisely, we consider two input memories a and a', that share identical public variables. After the execution, the values of `public` are conditionally replaced with value 0 or 1, depending on the truth value a the `secret` variable, on which we do not make any assumption. Therefore, the resulting memories are not ensured to share identical public variables.

This second example highlights that modifying a public value in a branch whose condition depends on a secret may break the NI policy. Indeed, any user could then observe the resulting public value to deduce the truth values of the secret conditions. This is called an *implicit flow*, and forbidding implicit flows is another step toward NI.

Forbidding both explicit and implicit flows is a sufficient (but not necessary) condition to enforce the NI policy. These flows can typically be detected by using a type-system. In chapter 2, we will present an extended background on the NI policy. We will present a type-system allowing to detect flows, and conduct a didactic proof its soundness.

Observational Non-Interference. The NI policy can be used to detect flaws in a system allowing different levels of security for users, by ensuring that high-security data never flows to lower security data. However, this does not capture the side-channel behavior of a system.

The *side-channel behavior* of a program includes the information that may be measured by an attacker, including execution-time, power consumption, etc. It should then be considered as public data, and the attacker may deduce sensitive information from these measurements.

The Observational Non-Interference (ONI) [BGL18] policy captures this idea. Formally, the semantics of the language is instrumented to encompass a notion of *leakage* emitted by an execution. The leakage is typically a list of events that capture the side-channel behavior of the execution. We assume that an execution of a program p relates an initial memory a to a leakage ℓ.

Definition 2: Observational Non-Interference (ONI).

A program p satisfies the Observational Non-Interference (ONI) policy if, for any memories a and a' such that $a_L = a'_L$, executing p on a and a' respectively leaks ℓ and ℓ', such that $\ell = \ell'$.

Similarly to the non-interference policy, ONI considers two executions starting from memories that have similar identical public variables. For any pair of executions, the leakages produced must be identical. Indeed, any difference in the leakage would lead to a flow from sensitive data towards leaked information.

Interestingly, ONI may be used to model several popular side-channels countermeasures. This includes the constant-time policy, and another policy known in the literature as *time-balancing* or *constant-resource* [Wu+18; Aga00; Ngo+17]. This policy, which we will exclusively refer to as Constant-Resource (CR) throughout this thesis, can be seen as a relaxation of constant-time; indeed, it allows more programs, that may contain secret-dependent branches that are balanced with respect to a given timing model. In chapter 3, our work will focus on the constant-time policy, and we will develop our second contribution on the CR policy in chapter 4. The problem of preservation of the CR policy is challenging, as existing proof methodology for other well-studied ONI policies, such as constant-time, can not be used in this case.

Enforcement of Security during Compilation. Our work focuses on the preservation of side-channel security during compilation, as our setting assumes the source programs to be secure, which is typically ensured by a prior program analysis. Other works also consider the problem of constant-time preservation, that we now present. Jasmin [Alm+17; Alm+19] is a programming framework, allowing the programmer to write programs in the Jasmin programming language. The Jasmin programming language allows the programmer to control low-level details (assembly instructions, registers, etc.), but also provides high-level abstractions (e.g., variables, functions, etc.). The language is then particularly suitable to implement cryptographic primitives. The Jasmin compiler then compiles programs down to efficient assembly code, that uses vectorized instructions [Alm+19], that is resistant to timing side-channel attacks. The Jasmin compiler is formally verified for semantic correctness, but not for preservation of security. Still, the author argue that the Jasmin compiler preserves the constant-time policy.

In [Cau+19], the authors present FaCT, a domain specific language addressing the challenge of writing human-readable constant-time cryptographic code. The language provides high-level constructs, that are compiled by the FaCT compiler down to constant-time LLVM bytecode. It is designed to make cryptographic libraries easier to implement. Indeed, a FaCT developer can focus on the correctness of the implementation, and then rely on the compiler to apply usual recipes (such as bitwise operations) yielding a constant-time compiled program. The FaCT compiler relies on a static information-flow type-system. The type-system allows to annotate variables as secret or public, then reject unsafe programs. However, they only rely on empirical evaluation, using dudect [RBV17], to ensure that the generated code has a constant-time behaviour. Dudect is an experimental tool designed to evaluate whether or not a program runs with a constant execution-time, using empirical measurements and classic statistical tests.

In [Mur+16], the authors present a flow-sensitive dependent type-system for shared-memory programs, which enforces a strong policy: timing-sensitive non-interference for concurrent programs. Then in [SM19], the authors study the preservation of this policy during compilation. The authors formally verify the preservation of their policy through compilation from a small concurrent imperative language to a RISC architecture. Their approach relies on the use of an information-flow type-system in the compilation chain. Their work is mechanically verified using the Isabelle proof assistant.

In [BDJ19], the authors consider the preservation of Information-Flow during compilation. The property they study does not deal with timing side-channels, but rather considers an attacker capable of observing an arbitrary amount of information during an execution. Their policy ensures that a compiled program does not leak more information than the associated source program, for instance by optimizing away code erasing secret values in the memory. They also present proof principles designed to prove that a trans-

formation preserves their security policy. In recent work [Coh+21], the authors consider a similar problem with a different approach. They introduce an opaque annotation, and a security policy enforcing that any observation occurring in an opaque area must be preserved through compilation, thus allowing to prevent unwanted dead-code elimination. They study the preservation of their policy from C code to machine code. Their opaque annotations disable aggressive optimizations, while the compiler does not need to know the security levels of the variables.

In [Vas+21], the authors focus on speculative attacks, and present Blade, an automatic tool able to repair a program vulnerable to this kind of attacks. They use a leakage modeling the effects of speculative execution. Their approach is based on the insertion of a minimal amount of annotations called *protect* in the source code. These annotations may then be implemented as memory fence instructions at assembly level.

Recently, in [Bar+21a], the authors tackle the problem of integrating the verification of the speculative constant-time policy into the Jasmin framework. They also modify the Jasmin language to include a new *fence* instruction, that is compiled down to an assembly *memory fence* instruction. Memory fences after conditionals and before load instructions is a standard way to protect against speculative attacks, as fences disable the speculative mechanisms [Wan+19; Bar+21a]. The authors explain that the *stack sharing* pass of the Jasmin compiler is likely to break the speculative constant-time policy. They then check that a program is secure after this pass, and argue that the remaining compiler passes preserve speculative constant-time.

In another recent work [PG21], the authors present a compilation framework allowing to reason about classes of speculative execution attacks. This framework targets the usual imperative While language equipped with speculative semantics. The speculative semantics consists in always executing a fixed number of steps in the wrong branch of every branch instruction, before rolling back, and thus do not model any microarchitectural behavior. The authors then present speculative non-interference (SNI), that exactly captures the information leaked only by speculative execution, and address the problem of the preservation of SNI during compilation. The paper also define speculative safety (SS), a sound taint-based over-approximation of SNI. Finally, the authors present a methodology to prove or disprove their preservation criteria, and apply it to study common countermeasures implemented in compilers against Spectre v1.

Contributions and Organization of the Thesis. In this thesis, we focus on formally verified secure-compilation that preserves countermeasures against timing side-channel attacks. More precisely, we tackle mainly two questions.

1. How can we modify an existing compiler, so that it preserves the constant-time policy?

2. How to ensure the preservation of the constant-resource policy?

We provide our answers to these challenges in this document. The first question raises issues from a compilation perspective. Specifically, we answer it in the case of the CompCert compiler, that we modified in order to make it constant-time preserving.

We tackle the second question by focusing on the preservation of the constant-resource policy, which had never been tackled in the literature to the best of our knowledge. Studying the preservation of this policy has raised several challenges, that led us to define a more generic security policy. In this work, we focus on a small language and reason on common optimization techniques, rather than the CompCert compiler.

We present our answers to the questions above as follows.

— First, chapter 2 is designed as a didactic background on the Non-Interference (NI) policy. In this chapter, we show how type-systems can be used to enforce NI policies, and present a didactic proof of the soundness of such type-system, in the case of a language equipped with a small-step semantics with continuation. Using these results, we discuss the limitations of this approach when interacting with compilers to better justify the methodology we follow in the rest of this thesis.

— In chapter 3, we directly tackle the problem of secure compilation preserving countermeasures against timing side-channel attacks. We focus on the cryptographic constant-time policy, and study how to make the CompCert compiler constant-time preserving. This chapter is based on the work published in the 47th Symposium on Principles of Programming Languages (POPL) in [Bar+19]. The contribution of this work was to present a machine checked proof that a modified version of CompCert preserves the constant-time policy. This work presented two distinct contributions: (a) a modification of the compiler, that fixes problematic parts that explicitly break the policy and (b) a proof methodology, applied to all the intermediate transformations, allowing to prove that the whole modified compiler preserves the policy. Our personal contribution is the first of these two items. In chapter 3, we present in detail this work. Specifically, we first present a detailed background on the CompCert compiler, detailing its architecture (i.e., intermediate representations, transformations), then we highlight several parts that we modified in the compiler, in order to make it constant-time preserving. The modifications presented in this chapter have since been integrated in the current version of CompCert.

— Last, in chapter 4, we tackle the problem of the preservation of the Constant-Resource (CR) policy. We study this policy in the case of a small imperative language, and study the preservation of the policy through usual optimizations. Our contribution includes the introduction of a more generic security policy, that we call CR$^{\#}$. The CR$^{\#}$ relies on syntactic annotations, called *atomic* annotations, used to keep track of secret values in a program without the need of a taint-tracking

type-system. This new policy can be seen as a flexible mix between the constant-time and the CR policies. We then study standard optimizations operating on a small imperative language. We describe how to implement these program transformations in a CR$^{\#}$-preserving way. Our approach relies the introduction of a minimal amount of padding instructions. This chapter is a longer version of the work published in the 34th Computer Security Foundations Symposium (CSF) in [Bar+21b].

The results presented in this thesis have all been formally specified, implemented and mechanically verified using the Coq proof assistant [19], but this document is intended to be readable with only very little knowledge of this tool. The whole development is provided as supplementary material and can be found online [1]. The electronic version of this document contains links to the development for every definition, theorem or proof presented in this thesis. These links are indicated with Coq logos: .

1. `https://remihut.in/thesis/coq/toc.html`

Chapter 2

Enforcing Non-Interference with Typing

In chapter 1, we introduced the notion of Non-Interference (NI). We recall that this policy relies on a classification of the security level of the variables of the program. In our example, we reason on a two levels classification: L refers to the public variables, and H refers to the secret variables. A program is then said to be NI if considering two executions starting from memories that have identical public variables, the resulting memories also have identical public variables.

In this chapter, we present a more complete study of the NI policy. This chapter is intended to be didactic, and inspires a lot on works by Volpano, Irvine and Smith [VIS96], and Barthe, Pichardie and Rezk [BPR07]. We follow a similar methodology, but apply it to a language equipped with a different kind of semantics: specifically, we use a small-step semantics with continuations.

More precisely, in this chapter, we first introduce a language, with its detailed semantics, in section 2.1. In particular, we justify our choice of semantics in this section. Next, in section 2.2, we present a type-system, called the *secure flow type-system*, whose goal is to prevent both direct and indirect flows in a program. We then present a detailed proof in section 2.3 that any well-typed program is secure with respect to the NI policy. Afterwards, in section 2.4, we also highlight of the limitations of type-system we introduce when used inside the compilation chain of a realistic compiler. Ultimately, the goal of this chapter is to better motivate the approach we follow in chapter 3 and chapter 4, in which we study the preservation during compilation of information-flow policies with a methodology that does not rely on a taint-tracking mechanism (such as the secure flow type-system). Last, we conclude that chapter by introducing the Observational Non-Interference (ONI) policy, a generalization of the NI policy, that can be used to model policies protecting against side-channels attacks. The *constant-time* and *constant-resource* are examples of such policies, and can be formally defined as instances of ONI. We will study these policies in respectively chapter 3 and chapter 4.

Type-Systems. A type-system is a usual addition to a programming language, allowing to associate a type (boolean, integer, array, pointer, ...) to every variable in the program.

The goal of the type-system is to restrict the operations performed by the program, by forbidding some operations that do not make sense in the language (e.g., dividing an integer by an array). The type-system then ensures that a well-typed program does not perform such operation.

A type-system is usually written as a relation between a typing environment Γ, a program p and a type τ, written as $\Gamma \vdash p : \tau$, and which means that program p has type τ in environment Γ. The type-system may then be defined as inference rules. We give an example below:

Example 3: Simple type-system.

We consider a simple type-system for integer expressions, defined by the following inference rules:

$$\textsc{int}\ \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \textsc{var}\ \frac{\Gamma(x) = \mathsf{int}}{\Gamma \vdash x : \mathsf{int}} \qquad \textsc{add}\ \frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \ \texttt{+}\ e_2 : \mathsf{int}}$$

Rule INT means that any integer literal n has type int. Rule VAR means that if a variable x has type int is the environment Γ, then the expression x is well-typed of type int. Rule ADD allows to deduce the type of an addition of expression: if both operands have type int, then the expression has type int.

We can use this type-system to deduce the type of expression $a \ \texttt{+}\ 5$, assuming that $\Gamma(a) = \mathsf{int}$. We have the following deductions:

$$\textsc{add}\ \frac{\textsc{var}\ \dfrac{\Gamma(a) = \mathsf{int}}{\Gamma \vdash a : \mathsf{int}} \qquad \textsc{int}\ \dfrac{}{\Gamma \vdash n : \mathsf{int}}}{\Gamma \vdash a \ \texttt{+}\ 5 : \mathsf{int}}$$

This type-system allows to conclude that $a \ \texttt{+}\ 5$ is well-typed of type int. However, if a had a different type in Γ, say $\Gamma(a) = \mathsf{array}$, this expression would have been rejected by the type-system.

Type-systems are generic tools, that can be used as an automatic procedure to either accept or reject a program. They are a standard way to enforce the NI policy, which is what we will focus on in this chapter. In other words, we will see how to build a type system such that any well-typed program is NI:

$$\Gamma \vdash p : \tau \implies \mathrm{NI}(p)$$

Such a type-system is designed to reject any program that performs either explicit or implicit flows, and is thus call a *Secure Flow* type system.

Plan of the chapter. During this chapter, we will first define a language $\mathscr{L}$ in section 2.1, and define a *Secure Flow* type system for this language in section 2.2. Then, in section 2.3, we will conduct a proof of the soundness of this type-system, i.e., we will prove that any well-typed program is non-interferent. During section 2.4, we will study the case of preservation of the non-interference policy during compilation. We address this problem by studying the preservation of the Secure Flow type-system during the compilation, and present the limitations of this approach. Last, we will extend the NI policy in section 2.5, and introduce the Observational Non-Interference (ONI) policy.

2.1 Definition of a Language

In this section, we define a language $\mathscr{L}$ by giving its syntax and its semantics. This language will be used and extended throughout this document. This precise definition will later allow us to reason about the execution of a program.

2.1.1 Syntax of $\mathscr{L}$

We consider an imperative language, that can manipulate integer variables and integer arrays of constant size, and that contains usual `if` and `while` statements. In this language, arrays are statically allocated, and their size is never modified during an execution. The syntax is given in the following definition:

Definition 2.1: Syntax of $\mathscr{L}$.

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle int \rangle \mid \langle ident \rangle \mid \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{]} \mid \langle exp \rangle \diamond \langle exp \rangle \mid \\
\langle stmt \rangle & ::= & \texttt{skip} \\
& \mid & \langle ident \rangle \texttt{ := } \langle exp \rangle \\
& \mid & \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{] := } \langle exp \rangle \\
& \mid & \langle stmt \rangle \texttt{ ; } \langle stmt \rangle \\
& \mid & \texttt{if(} \langle exp \rangle \texttt{) \{ } \langle stmt \rangle \texttt{ \} else \{ } \langle stmt \rangle \texttt{ \}} \\
& \mid & \texttt{while(} \langle exp \rangle \texttt{) \{ } \langle stmt \rangle \texttt{ \}}
\end{array}
$$

Language $\mathscr{L}$ consists of expressions $\langle exp \rangle$ and statements $\langle stmt \rangle$. An expression is either an integer constant n, a variable identifier x, an access to an array element $x\texttt{[}e\texttt{]}$ or an operation applied to a pair of expressions (e.g., arithmetic operations, comparisons, ...). A statement is either a `skip` (which does nothing), an assignment s `:=` e with e

an expression, an assignment to an array element $s\texttt{[}e_1\texttt{]}$ `:=` e_2 with e_1, e_2 expressions, a sequence of statements $s_1\texttt{;}\ s_2$, and usual control flow with `if` and `while` statements.

2.1.2 Big-step Semantics for Expressions of $\mathscr{L}$

We define a formal semantics for language $\mathscr{L}$. We first present a semantics for the expressions of the language. As the evaluation of an expression always terminates in our model, we choose to define their semantics using a big-step style semantics.

Our semantics relies a on notion of values and environments. The set of values $\mathcal{V}$ describes the values to which the variables of our language may evaluate to. In our language, a value v may either be an integer $n \in \mathbb{Z}$ or an array of integers of constant size. The set of value is defined as follows:

Definition 2.2: Set of values $\mathcal{V}$.

$$\langle value \rangle \ ::= \ n \mid [n_1; \ldots; n_m]$$

Next an environment σ is a partial mapping from variable identifiers to the set of values $\mathcal{V}$. We denote $\sigma[x]$ the value associated to the identifier x in the environment σ. If identifier x is not defined in σ, then $\sigma[x]$ evaluates to the default value 0. We denote $\sigma[x \leftarrow v]$ the environment that is identical to σ, except for the identifier x which is now mapped to the value v.

The semantics of expressions is defined as a relation $\langle e, \sigma \rangle \Downarrow v$. It reads as expression e, evaluated in environment σ, produces value v. It is defined as follows:

Definition 2.3: Evaluation of expression $\langle e, \sigma \rangle \Downarrow v$.

$$\text{VAR}\ \frac{\sigma[x] = v}{\langle x, \sigma \rangle \Downarrow v} \qquad \text{ARRAY}\ \frac{\sigma[x] = [v_1; ...; v_n; ...v_m] \qquad \langle e, \sigma \rangle \Downarrow n \qquad 1 \leq n \leq m}{\langle x\texttt{[}e\texttt{]}, \sigma \rangle \Downarrow v_n}$$

$$\text{CONST}\ \frac{}{\langle n, \sigma \rangle \Downarrow n} \qquad \text{OP_BIN}\ \frac{\langle e_1, \sigma \rangle \Downarrow v_1 \qquad \langle e_2, \sigma \rangle \Downarrow v_2}{\langle e_1 \diamond e_2, \sigma \rangle \Downarrow v_1 \diamond v_2}$$

There are four rules in **Definition 2.3**. Evaluating an integer constant n produces the value n itself, no matter the environment the expression is being evaluated in (rule CONST).

Evaluating a variable identifier x in environment σ produces value v where $v = \sigma[x]$ (rule VAR). Evaluating an array access identifier $x\,[e]$ in environment σ requires to evaluate e in σ to value n, and requires x to be an array of length at least n in σ. $x\,[e]$ then evaluates to v_n, the nth element of array x (rule ARRAY). Last, evaluating a binary operation $\diamond$ between two expressions e_1 and e_2 in environment σ first requires to evaluate e_1 and e_2 in σ to their respective values v_1 and v_2. Expression $e_1 \diamond e_2$ then produces the value $v_1 \diamond v_2$ (rule OP_BIN). Note that for the sake of simplicity, we use the same symbol $\diamond$ both for the syntax of the binary operation and its semantics.

2.1.3 Small-step Semantics with Continuations

We now focus on statements of our language, for which we define a semantics in a small-step style using continuations. We choose this style of semantics for the following two reasons:

— First, small-step semantics with continuations are easier to reason about than usual small-step semantics. They allow to avoid the need to reason inductively when dealing with loops and sequences. This can be observed in the proofs that will be conducted in this chapter.

— Second, this type of semantics is largely used in the CompCert compiler. Our goal in this work is to adapt existing proof methodologies from [VIS96; BPR07], to better fit to the restrictive framework of the CompCert compiler. To the best of our knowledge, the proof of the soundness of a secure flow type-system with this kind of semantics had never been tackled in the literature.

Our semantics is expressed as a transition system between pairs of environments and continuation.

We define a continuation k as a stack of statements. The top element in the continuation is the statement being currently executed, and the other elements of the stack will only be executed once the current statement is done. We use notation $s \cdot k$ to denote the continuation with statement s on top, with k the rest of the continuation. We also denote $k_1 + k_2$ as the concatenation of continuations k_1 and k_2, and ϵ the empty continuation. Last, we denote $\|k\|$ the length of the continuation k.

Executing the continuation $s_1 \cdot s_2 \cdot s_3 \cdot \epsilon$ consists in executing s_1, then s_2, then s_3, before terminating. During an execution, new elements can be stacked in the continuation.

A step of execution is defined with the following step relation:

$$\langle \sigma_1,\, k_1 \rangle \longmapsto \langle \sigma_2,\, k_2 \rangle$$

Note that the current instruction does not explicitly appear in the step relation, but remember that it is hidden in the top element of k_1. The execution terminates when

the empty continuation ϵ is reached, which means that nothing is left to execute. The complete definition of the step relation is presented below:

Definition 2.4: Step relation for statements $\langle \sigma_1,\, k_1\rangle \longmapsto \langle \sigma_2,\, k_2\rangle$.

SKIP

$$\frac{}{\langle \sigma,\, \texttt{skip} \cdot k\rangle \longmapsto \langle \sigma,\, k\rangle}$$

ASSIGN

$$\frac{\langle \sigma,\, e\rangle \Downarrow v \qquad v \text{ is an integer}}{\langle \sigma,\, (x\ \texttt{:=}\ e) \cdot k\rangle \longmapsto \langle \sigma[x \leftarrow v],\, k\rangle}$$

ASSIGN_ARRAY

$$\frac{\sigma[x] = [v_1; ...; v_n; ...v_m] \qquad \langle \sigma,\, e_1\rangle \Downarrow n \qquad \langle \sigma,\, e_2\rangle \Downarrow v' \qquad 1 \leq n \leq m \qquad v' \in \mathbb{Z}}{\langle \sigma,\, (x\,\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2) \cdot k\rangle \longmapsto \langle \sigma[x \leftarrow [v_1; ...; v_{n-1}; v'; v_{n+1}; ...v_m]],\, k\rangle}$$

SEQ

$$\frac{}{\langle \sigma,\, (s_1\texttt{;}\ s_2) \cdot k\rangle \longmapsto \langle \sigma,\, s_1 \cdot s_2 \cdot k\rangle}$$

IF_TRUE

$$\frac{\langle \sigma,\, e\rangle \Downarrow v \qquad v \neq 0}{\langle \sigma,\, (\texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}) \cdot k\rangle \longmapsto \langle \sigma,\, s_1 \cdot k\rangle}$$

IF_FALSE

$$\frac{\langle \sigma,\, e\rangle \Downarrow v \qquad v = 0}{\langle \sigma,\, (\texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}) \cdot k\rangle \longmapsto \langle \sigma,\, s_2 \cdot k\rangle}$$

WHILE

$$\frac{}{\langle \sigma,\, (\texttt{while}(e)\ \{s\}) \cdot k\rangle \longmapsto \langle \sigma,\, \texttt{if}(e)\ \{s\texttt{;}\ \texttt{while}(e)\ \{s\}\}\ \texttt{else}\ \{\texttt{skip}\} \cdot k\rangle}$$

By convention, executing a program p with initial environment σ with our semantics is done by executing from the pair $\langle \sigma,\, p \cdot \epsilon\rangle$, i.e., the continuation made of only one element: the program to be executed. The execution is over when an empty continuation ϵ is reached.

We now detail the semantics. All the rules are defined depending on the top element of the continuation, i.e., the current instruction. Executing a `skip` instruction does nothing, the execution the proceeds to the next element of the continuation (rule SKIP). Executing an assignment x `:=` e first requires to evaluated expression e to a value v. The step leads to environment $\sigma[x \leftarrow v]$, in which the rest of the continuation k will later be executed (rule ASSIGN). Executing an assignment to an array element x`[`e_1`]` `:=` e_2 first requires both e_1 and e_2 to evaluate to integer values, respectively n and v'. The step leads to the environment in which the nth element of x is replaced by v' (rule ASSIGN_ARRAY). Executing a sequence s_1`;` s_2 only consists in stacking s_2 then s_1 on the continuation. This means that the execution will proceed by executing s_1, and will later execute s_2 when execution of s_1 is done (rule SEQ). Executing a `if(`e`) {`s_1`} else {`s_2`}` statement first requires to evaluate the condition e. Depending on the result, the appropriate branch statement s_1 or s_2 is stacked on the continuation, to be executed later (rules IF_TRUE and IF_FALSE). Executing a `while(`e`) {`s`}` loop consists in stacking the whole statement `if(`e`) {`s`; while(`e`) {`s`}} else {skip}`. In other words, our semantics unrolls one iteration of the WHILE loop.

Finally, we define a multi-step relation, denoted with $\langle \sigma_1, k_1 \rangle \longmapsto^* \langle \sigma_2, k_2 \rangle$, meaning that executing form $\langle \sigma_1, k_1 \rangle$ will lead to $\langle \sigma_2, k_2 \rangle$ in any number of steps (including 0). $\longmapsto^*$ is the reflexive, transitive closure of $\longmapsto$. A complete execution of a program p is then characterized by $\langle \sigma_1, p \cdot \epsilon \rangle \longmapsto^* \langle \sigma_2, \epsilon \rangle$ Similarly, $\langle \sigma_1, k_1 \rangle \longmapsto^n \langle \sigma_2, k_2 \rangle$ means that $\langle \sigma_1, k_1 \rangle$ leads to $\langle \sigma_2, k_2 \rangle$ in exactly n steps.

2.1.4 Definition of Non-Interference

We now formally define the notion of non-interference for our language $\mathscr{L}$. This security policy is defined w.r.t. to a security lattice $(\mathcal{L}, \leq)$ of security levels, as defined in [Den76]. The most common security lattice is $\{L, H\}$ with $L \leq H$, where L and H respectively represent low and high security levels. We will use this lattice to illustrate our examples, even though our work is defined in the general case of an arbitrary security lattice, with $\leq$ a partial order on $\mathscr{L}$.

We then introduce a security typing environment Γ that matches every variable identifier to a security level. For example, if variable x verifies $\Gamma[x] = H$, then x is a secret variable. The notion of non-interference is defined relative to a security level τ_{obs}, that corresponds to the security level of an attacker. Intuitively, the attacker is allowed to observe variables whose security level is lower or equal to τ_{obs} (and wants to deduce data he is not allowed to observe). We fix once and for all this security level τ_{obs}.

We first define a notion of a indistinguishability between two environments. Intuitively, two environments are indistinguishable with respect to the security level τ_{obs} if any observable variable x (i.e., $\Gamma[x] \leq \tau_{obs}$) has the same value in both environments. Formally:

Definition 2.5: Indistinguishability.

Two environments σ_1 and σ_2 are indistinguishable, denoted as $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$, if:

$$\sigma_1 \simeq_{\tau_{obs}} \sigma_2 \triangleq \forall x, \Gamma(x) \leq \tau_{obs} \implies \sigma_1[x] = \sigma_2[x]$$

Non-interference is then defined as follows. Intuitively, a program p is non-interferent if for any pair of indistinguishable initial environments σ_1 and σ_2, executing p from both environments will lead to indistinguishable final environments. Formally:

Definition 2.6: Non-interference.

Let p be a program and τ_{obs} a security level. If for any initial environments σ_1 and σ_2 such that:

- $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$
- $\langle \sigma_1, p \cdot \epsilon \rangle \longmapsto^* \langle \sigma_1', \epsilon \rangle$
- $\langle \sigma_2, p \cdot \epsilon \rangle \longmapsto^* \langle \sigma_2', \epsilon \rangle$

we have that $\sigma_1' \simeq_{\tau_{obs}} \sigma_2'$ (i.e., the final states are indistinguishable), we then say that p is non-interferent. We denote it as $\mathrm{NI}(p)$.

Note that this definition is a direct instanciation of **Definition 1**, adapted to our language $\mathscr{L}$. The definition we choose to express the NI policy is *termination-insensitive*. This means that we only considers terminating executions of programs (indeed, we expect both executions to reach the empty continuation i.e., the end of the execution in our language), and ignore diverging executions (e.g., infinite loops). *Termination-sensitive* definition of NI have been studied in the literature [DP10; KWH11].

This conclude this section of definitions of the language $\mathscr{L}$. In the next section, we will focus on the *Secure Flow* type-system, whose goal is to enforce NI.

2.2 Secure Flow Type-System

We introduce the *Secure Flow* type-system that can enforce the NI policy. The goal of this type-system is to reject any program performing either an explicit or an implicit flow. Therefore, any well-typed program with respect to this type system should be NI. The type-system we present here is *flow-insensitive*, which means that a variable x has a fixed security level $\Gamma(x)$ throughout an execution, given by the typing environment Γ. *Flow-sensitive* definitions exists as well, and allow the security of variables to be modified during an execution. Flow-insensitivity is easier to define and reason about, but is also more restrictive, as NI-secure programs may be rejected by a flow-insensitive type-system,

but accepted by a flow-sensitive one. Yet, this restriction can be relaxed with program transformation ensuring that variables are assigned only once during an execution, and thus that their security level does not need to be modified. This is detailed in [HS06].

2.2.1 Type-system for Expressions

The following definition presents the type-system for expressions.

Definition 2.7: Secure Flow type-system for expressions $\Gamma \vdash e : \tau$.

$$\text{CONST}\ \frac{}{\Gamma \vdash n : \tau} \qquad \text{VAR}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{ARRAY}\ \frac{\Gamma[a] = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash a[e] : \tau}$$

$$\text{OP_BIN}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \diamond e_2) : \tau} \qquad \text{SUBTYPE_E}\ \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

A constant value n is well-typed and can be assigned any type τ (rule CONST). A variable identifier x is well-typed if and only if it is defined in the typing environment Γ; its type is then $\Gamma[x]$ (rule VAR). An operation $e_1 \diamond e_2$ is well-typed of type τ if and only if both e_1 and e_2 are well-typed with the same type τ (rule OP). This rule is restrictive as it only allows to combine expressions of the same type. Intuitively, we want l+h to be well typed, and we want it to be of type H. This is why we introduce the last rule SUBTYPE_E. This rule states that any well-typed expression can be casted into a larger type (w.r.t. $\leq$). For example, any L expression can be casted to H.

Below is an example of the typing derivation of the expression l+h.

Example 4: Typing of l+h.

To illustrate the type-system, we introduce variable identifiers l and h, which are of respective type L and H.

$$\text{OP_BIN}\ \frac{\text{SUBTYPE_E}\ \dfrac{\text{VAR}\ \dfrac{\Gamma(l) = L}{\Gamma \vdash l : L} \qquad L \leq H}{\Gamma \vdash l : H} \qquad \text{VAR}\ \dfrac{\Gamma(h) = H}{\Gamma \vdash h : H}}{\Gamma \vdash (l+h) : H}$$

Typing derivation of expression l+h, with l and h of respective types L and H.

2.2.2 Type-system for Statements

The following definition presents the type-system for statements.

Definition 2.8: Secure Flow type-system for statements $\Gamma \vdash s : \tau$.

$$\text{SKIP}\quad \frac{}{\Gamma \vdash \texttt{skip} : \tau} \qquad \text{ASN}\quad \frac{\Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (x \ \texttt{:=}\ e) : \tau} \qquad \text{ARRAY_ASN}\quad \frac{\Gamma[a] = \tau \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (a\texttt{[}e_1\texttt{]}\ \texttt{:=}\ e_2) : \tau}$$

$$\text{SEQ}\quad \frac{\Gamma \vdash s_1 : \tau \qquad \Gamma \vdash s_2 : \tau}{\Gamma \vdash (s_1; s_2) : \tau} \qquad \text{IF}\quad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s_1 : \tau \qquad \Gamma \vdash s_2 : \tau}{\Gamma \vdash (\texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) : \tau}$$

$$\text{WHILE}\quad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash s : \tau}{\Gamma \vdash (\texttt{while}\, e\, \texttt{do}\, s) : \tau} \qquad \text{SUBTYPE_S}\quad \frac{\Gamma \vdash s : \tau_2 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash s : \tau_1}$$

First, a `skip` instruction can be assigned any arbitrary type τ (rule SKIP). An assignment instruction x `:=` e is well-typed of type τ if and only both variables x and expression e have the same type τ (rule ASN). Note that thanks to the subtyping rule for expressions, this intuitively means that the type of e is lower than the type of x (w.r.t. $\leq$). This rule then prevents explicit flows, as a statement l `:=` h is rejected by the type-system. The rules SEQ, IF and WHILE all require every sub-expression and sub-statement to be well-typed of type τ, in order to type the resulting statement. In the case of rule IF (reasoning is similar for rule WHILE), it prevents implicit flows. Indeed, a statement `if(`h`) {`l `:= 2}` is rejected by the type-system, as h has type H and l `:= 2` has type L. However, the rules here are too restrictive, as statement `if(`l`) {`h `:= 2}` is similarly rejected by the type-system, whereas it does not present any information flow. This is why we introduce the last rule SUBTYPE_S. This rule states that any well-typed statement can be casted into a smaller type (w.r.t. $\leq$).

Below is an example of the typing derivation of the statement `if(`l`) {`h `:= 2}`.

Example 5: Typing of `if(`l`) {`h `:= 2}`.

To illustrate the type-system, we introduce variable identifiers l and h, which are of

respective type L and H.

$$\text{IF}\ \dfrac{\text{VAR}\ \dfrac{\Gamma(l) = L}{\Gamma \vdash l : L} \qquad \text{SUBTYPE_S}\ \dfrac{\text{ASN}\ \dfrac{\Gamma(h) = H \qquad \Gamma \vdash 2 : H}{\Gamma \vdash (h\ \texttt{:=}\ 2) : H} \qquad L \leq H}{\Gamma \vdash (h\ \texttt{:=}\ 2) : L}}{\Gamma \vdash (\texttt{if}(l)\ \{h\ \texttt{:=}\ 2\}) : L}$$

Typing derivation of statement `if(`l`) {`h `:=` 2`}`, with l and h of respective types L and H. The `else` branch is omitted for brevity.

This conclude this section introduction the Secure Flow type-system. In the next section, we will conduct a complete and didactic proof that the type-system enforces the NI policy.

2.3 Soundness of the Type-System

The Secure Flow type-system introduced in section 2.2 is used to reject any program performing implicit or explicit flows. Our goal is to prove that this restriction on programs is sufficient to enforce the NI policy. In other words, we want to prove the following theorem:

Theorem 2.1: Type Soundness.
Any program well-typed program p is non-interferent.

In this section, we conduct a detailed and didactic proof of this theorem, and thus introduce several intermediate lemmas, and introduce some proof artifacts. More specifically, in subsection 2.3.1 we first extend the Secure Flow type-system to continuations, and state a generalized type soundness theorem, dealing with continuations. Next, we prove several lemmas about the indistinguishability relation in subsection 2.3.2. Next, we introduce another relation called `unobservable`, and prove interesting results about this relation in subsection 2.3.3. In subsection 2.3.4, we introduce a useful lemma allowing us to reason on a prefix of an execution of a program. Last, in subsection 2.3.5, we use all these results to conduct the proof of the generalized type soundness theorem.

The proof methodology presented in this section is adapted from [BPR07]. Specifically, we tried to reuse as many intermediate lemmas as possible, while keeping similar names (e.g., *locally respects*, *high branching*, *high step*, etc).

2.3.1 Generalization to Continuation

First, in order to conduct our proof, we need to introduce some generalized definition. We need to be able to reason on complete executions of an arbitrary continuation. We then extend the definition of the non-interference property and of the type-system to continuations (instead of statements only). We say that a continuation k is well-typed if each of its elements are well-typed. We denote it as $\Gamma \vdash k$. We define it formally as follows.

Definition 2.9: Secure Flow type-system for continuations $\Gamma \vdash k : \tau$.

$$\text{EPSILON}\ \frac{}{\Gamma \vdash \epsilon : \tau} \qquad \text{PUSH}\ \frac{\Gamma \vdash s : \tau \qquad \Gamma \vdash k : \tau}{\Gamma \vdash s \cdot k : \tau} \qquad \text{SUBTYPE_K}\ \frac{\Gamma \vdash k : \tau_2 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash k : \tau_1}$$

We can remark that this definition is consistent with the typing rule of the sequence of instructions. Note also that the subtyping rule is identical to the subtyping of statements.

Second, we naturally extend the non-inference policy to continuation as follows:

Definition 2.10: Non-interference with continuation.

Let k be a continuation and τ_{obs} a security level. If for any initial environments σ_1 and σ_2 such that:

- $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$
- $\langle \sigma_1, k \rangle \longmapsto^* \langle \sigma_1', \epsilon \rangle$
- $\langle \sigma_2, k \rangle \longmapsto^* \langle \sigma_2', \epsilon \rangle$

we have that $\sigma_1' \simeq_{\tau_{obs}} \sigma_2'$ (i.e., the final states are indistinguishable), we then say that k is non-interferent. We denote it as $\mathrm{NI}(k)$.

We use these definitions to reason on the complete execution of a continuation. Note that definition **Definition 2.6** is an instance of **Definition 2.10**, with $k = p \cdot \epsilon$. Our goal is now to prove that any well typed continuation k is NI, with respect to the definitions above.

2.3.2 Indistinguishability Lemmas

We state here two lemmas about the indistinguishability relation and the security type-system. Our first lemma states that if an expression e is well typed of type τ_{obs}, then the value it executes to only depends on variables of type $\leq \tau_{obs}$. We express this lemma as follows:

Lemma 2.2: Observable expression.

Suppose:

(I) $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$
(E_1) $\langle \sigma_1, e \rangle \Downarrow v_1$
(E_2) $\langle \sigma_2, e \rangle \Downarrow v_2$
(T) $\Gamma \vdash e : \tau_{obs}$

Then: $v_1 = v_2$

Intuitively, a direct consequence of the type-system is that for any well typed expression of type τ, all its sub-expressions may also be typed with type τ at most. **Lemma 2.2** is a direct consequence of this fact. Below is a more complete proof:

Proof of Lemma 2.2.

We reason by induction on E_1.

— Case CONST: e is necessarily a constant n, thus $v_1 = v_2 = n$.
— Case VAR: e is necessarily a variable identifier x. Therefore, we have $v_1 = \sigma_1[x]$ and $v_2 = \sigma_2[x]$. From hypothesis T, we deduce that $\Gamma[x] = \tau_{obs}$. By definition of indistinguishability (**Definition 2.5**), we then have $\sigma_1[x] = \sigma_2[x]$.
— Case BIN_OP: e is necessarily an operation $e_1 \diamond e_2$. We conclude by applying the induction hypothesis on e_1 and e_2.

□

Our second lemma deals with the step relation starting from an arbitrary continuation k. It states that if k is well typed, taking a step from indistinguishable environments σ_1 and σ_2 will lead to two other indistinguishable environments σ'_1 and σ'_2. In other words, taking a step on a well typed continuation preserves the indistinguishability relation. Note that we do not conclude on the resulting continuations, that are arbitrary. The lemma is stated as follows, and is followed by a formal proof.

Lemma 2.3: Locally respects.

Suppose:

(I) $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$
(S_1) $\langle \sigma_1, k \rangle \longmapsto \langle \sigma'_1, k'_1 \rangle$
(S_2) $\langle \sigma_2, k \rangle \longmapsto \langle \sigma'_2, k'_2 \rangle$
(T) $\Gamma \vdash k$

Then: $\sigma'_1 \simeq_{\tau_{obs}} \sigma'_2$

Proof of Lemma 2.3.

As we assume we can take a step from continuation k, it is necessarily non-empty. Let s be the top element of k. We reason by cases on the step relation. For every case except ASN, the step relation does not modify the output environment, and we then have $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$, hence $\sigma_1' \simeq_{\tau_{obs}} \sigma_2'$.

For case ASN. s is necessarily an assignment x `:=` e. We need to prove $\sigma_1' \simeq_{\tau_{obs}} \sigma_2'$. As σ_1' and σ_2' trivially have the same domain, we need to show that if $\Gamma[x] \leq \tau_{obs}, \sigma_1'[x] = \sigma_2'[x]$ (as every other variables are not altered). We then assume that $\Gamma[x] \leq \tau_{obs}$.

As we can take a step in both S_1 and S_2, e evaluates in σ_1 and σ_2, say respectively to v_1 and v_2. We need to show that $v_1 = v_2$. We know that s is typed (hypothesis T), and thus that $\Gamma \vdash e : \tau$ with $\tau \leq \Gamma[x]$ (consequence of the subtyping). We conclude using the observable expression **Lemma 2.2**.

□

This concludes our list of lemmas about the indistinguishability relation.

2.3.3 The Unobservability Relation

We previously stated a typing relation for continuations, that expects each of its elements to be well typed themselves. This relation represents the initial knowledge that we expect to have about a continuation k, i.e., we will show that this is the only assumption that we need to prove that k is NI. However, this typing relation does not carry a lot of information, as it discards all the types of its element. We introduce here a stronger relation called `unobservable`. Intuitively, a continuation k is said to be unobservable w.r.t. a type τ_{obs} if all its elements are well typed of type strictly greater than τ_{obs}. Intuitively, an unobservable continuation characterizes a high security part of the program, which means that an observer will not be able to deduce any information from this execution. Formally, we define it as follows:

Definition 2.11: Unobservable continuation.

$$\texttt{unobservable}(\Gamma, \tau_{obs}, k) \triangleq \forall s \in k, \forall \tau \not\leq \tau_{obs}, \Gamma \vdash s : \tau$$

We will use this definition to characterize high-security parts of the continuation. Specifically, we will see that branchings depending on secret values (i.e., greater than τ_{obs}) may only happen if the continuation is unobservable. The `unobservable` relation has good properties w.r.t. the step relation. The following lemma states that any step from an unobservable continuation will lead to another unobservable continuation.

Lemma 2.4: `unobservable` preservation.

Suppose:

(S) $\langle \sigma_1,\, k_1 \rangle \longmapsto \langle \sigma_2,\, k_2 \rangle$
(U) $\texttt{unobservable}(\Gamma, \tau_{obs}, k_1)$

Then: $\texttt{unobservable}(\Gamma, \tau_{obs}, k_2)$

Proof of Lemma 2.4.

We reason by case analysis on the step relation (S).

— cases SKIP and SKIP are trivial as they remove the top element of the continuation.

— cases SEQ, IF and WHILE insert elements in the continuation k', we need to prove these elements to be unobservable. Again by case analysis, we can show that as all these elements come from unobservable continuation k, they are themselves unobservable.

□

The next lemma states that if we take a step from an unobservable continuation, the input and output environments are indistinguishable (w.r.t. the same τ_{obs}).

Lemma 2.5: Indistinguishability of `unobservable` step.

Suppose:

(S) $\langle \sigma_1,\, k_1 \rangle \longmapsto \langle \sigma_2,\, k_2 \rangle$
(U) $\texttt{unobservable}(\Gamma, \tau_{obs}, k_1)$

Then: $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$

Proof of Lemma 2.5.

We reason by case analysis on the step relation (S). For every case except ASN, the step relation does not modify the output environment. We thus have $\sigma_1 = \sigma_2$.

For case ASN. s is necessarily an assignment x `:=` e. We need to show that if $\Gamma[x] \leq \tau_{obs}, \sigma_1[x] = \sigma_2[x]$ (as every other variables are not altered). We then assume that $\Gamma[x] \leq \tau_{obs}$.

We know that s is unobservable (hypothesis U), and thus that $\Gamma \vdash e : \tau$ with $\tau \not\leq \tau_{obs}$. However, as a consequence of the subtyping, we have $\tau \leq \Gamma[x]$. Therefore $\tau \leq \tau_{obs}$, which is a contradiction.

Additionally, we easily show that σ_1 and σ_2 have the same domain to conclude the proof. □

Our most important lemma about the unobservable relation is the following. We assume to have two different steps from a single statement s leading to two different continuations $k_1 \neq k_2$ (think of a branching leading to the `then` branch on one hand, and to

the `else` branch on the other hand). If the starting environments are indistinguishable and s is well typed, then k_1 is unobservable (and so is k_2 by symmetry).

Intuitively, the lemma can be interpreted as follows: if an `if` statement executed from indistinguishable environments leads respectively to the `then` branch and the `else` branch, then both branches are high-security (i.e., unobservable) parts of the program.

Lemma 2.6: `unobservable` branching.

Suppose:

$$
\begin{array}{ll}
(S_1) & \langle \sigma_1,\ s \cdot \epsilon \rangle \longmapsto \langle \sigma_1',\ k_1 \rangle \\
(S_2) & \langle \sigma_2,\ s \cdot \epsilon \rangle \longmapsto \langle \sigma_2',\ k_2 \rangle \\
(T) & \Gamma \vdash s : \tau \\
(D) & k_1 \neq k_2 \\
(I) & \sigma_1 \simeq_{\tau_{obs}} \sigma_2
\end{array}
$$

Then: $\texttt{unobservable}(\Gamma, \tau_{obs}, k_1)$

Proof of Lemma 2.6.

We reason by case analysis on the step relations S_1 and S_2. The only case that yields different continuations k_1 and k_2 is the case IF, the other cases are therefore trivially proved.

We focus on case IF, s is then necessarily an `if` statement: $s = \texttt{if}(e)\ \{s_1\}\ \texttt{else}\ \{s_2\}$. As s is well-typed (hypothesis T), e, s_1 and s_2 are all well-typed with type τ' such that $\tau \leq \tau'$.

We reason by cases.

— If $\tau' \leq \tau_{obs}$, we can use the observable expression lemma to deduce that e evaluates to the same value in both σ_1 and σ_2. As this would imply that $k_1 = k_2$, this is a contradiction.

— If $\tau' \not\leq \tau_{obs}$, k_1 and k_2 are unobservable by definition.

This concludes our list of lemmas about the unobservability relation.

□

2.3.4 Reasoning on Prefixes of Executions

Last, we state a lemma that only deals with the semantics of our program, and none of the security definitions that we introduced before. This lemma assume that we take n step from a continuation $k_1 + k_2$ (i.e., an arbitrary continuation in which we identify a prefix that we call k_1) leading to a continuation k whose length is less or equal of the length of k_1. This lemma shows that k_1 will step to ϵ in at most n steps. In other words, k_1 must be fully executed before execution of k_2 begins, and k_2 has no impact on the execution of k_1.

In other words, when executing a continuation, any prefix has to be fully executed before executing the rest of the continuation.

Lemma 2.7: Bottleneck.

Suppose:

(a) $\langle \sigma_1,\, k_1 + k_2 \rangle \longmapsto^n \langle \sigma_2,\, k \rangle$

(b) $\|k_1\| \leq \|k\|$

Then we can find $n' \in \mathbb{N}$ such that $n' \leq n$ and a state σ', such that:

$$\langle \sigma_1,\, k_1 \rangle \longmapsto^{n'} \langle \sigma',\, \epsilon \rangle.$$

This lemma shows this if we have an execution of $\langle \sigma_1,\, k_1 + k_2 \rangle \longmapsto^n \langle \sigma_2,\, k \rangle$, with $\|k_1\| \leq \|k\|$, then there is a bottleneck state $\langle \sigma',\, k_2 \rangle$ that will be encountered during the execution, verifying:

$$\langle \sigma_1,\, k_1 + k_2 \rangle \longmapsto^{n'} \langle \sigma',\, k_2 \rangle \longmapsto^{(n-n')} \langle \sigma_2,\, k \rangle$$

2.3.5 Proof of Type Soundness

We now have all the tools needed to prove the type soundness theorem, that we state as follows:

Theorem 2.8: Type Soundness.

Suppose:

(T) $\Gamma \vdash k$

(I) $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$

(E_1) $\langle \sigma_1,\, k \rangle \longmapsto^{n_1} \langle \sigma'_1,\, \epsilon \rangle$

(E_2) $\langle \sigma_2,\, k \rangle \longmapsto^{n_2} \langle \sigma'_2,\, \epsilon \rangle$

Then: $\sigma'_1 \simeq_{\tau_{obs}} \sigma'_2$

Proof of Theorem 2.8.

We reason by induction on $n = \max(n_1, n_2)$. If $n = 0$, then $n_1 = n_2 = 0$. We directly deduce that $\sigma'_1 = \sigma_1$ and $\sigma'_2 = \sigma_2$, allowing us to conclude. Otherwise, both n_1 and n_2 have to be strictly positive. This is a consequence of the fact that both executions start in environments that have the same domain of definition (I) and both executions terminate.

We then know that both executions take a step. We can find k_1, k_2, ρ_1, ρ_2 such that:

$$\begin{aligned} \langle \sigma_1,\, k\rangle &\longmapsto \langle \rho_1,\, k_1\rangle \\ \langle \sigma_2,\, k\rangle &\longmapsto \langle \rho_2,\, k_2\rangle \end{aligned} \tag{2.1}$$

Using **Lemma 2.3** (locally respects), we directly deduce that $\rho_1 \simeq_{\tau_{obs}} \rho_2$, i.e., ρ_1 and ρ_2 are indistinguishable.

If we suppose that $k_1 = k_2$, we then have:

$$\begin{aligned} \langle \rho_1,\, k_1\rangle &\longmapsto^{(n_1-1)} \langle \sigma_1',\, \epsilon\rangle \\ \langle \rho_2,\, k_1\rangle &\longmapsto^{(n_2-1)} \langle \sigma_2',\, \epsilon\rangle \end{aligned} \tag{2.2}$$

We can then directly conclude using the induction hypothesis, as both these executions are strictly shorter that n.

Otherwise, let us now assume that $k_1 \neq k_2$. First, note that k is necessarily non-empty (we could not take a step otherwise). We then introduce $k = s \cdot k'$, s being the top element of the continuation. By case analysis on the step relation, k' is necessarily a suffix of both k_1 and k_2. We note:

$$\begin{aligned} k_1 &= k_1' \cdot k' \\ k_2 &= k_2' \cdot k' \end{aligned} \tag{2.3}$$

where k_1' and k_2' are the respective prefixes of k_1 and k_2 (they may be empty).

Furthermore, taking a step from continuation $s \cdot k'$ will not modify k' (again by case analysis of the semantics). We then have the following executions:

$$\begin{aligned} \langle \sigma_1,\, s \cdot \epsilon\rangle &\longmapsto \langle \rho_1,\, k_1'\rangle \\ \langle \sigma_2,\, s \cdot \epsilon\rangle &\longmapsto \langle \rho_2,\, k_2'\rangle \end{aligned} \tag{2.4}$$

Using the **Lemma 2.6** (`unobservable` branching), we can now conclude that both k_1' and k_2' are unobservable.

Next, using the *bottleneck* lemma, we can find two environments ρ_1' and ρ_2' such that:

$$\begin{aligned} \langle \rho_1,\, k_1'\rangle &\longmapsto^{m_1} \langle \rho_1',\, \epsilon\rangle \\ \langle \rho_2,\, k_2'\rangle &\longmapsto^{m_2} \langle \rho_2',\, \epsilon\rangle \end{aligned} \tag{2.5}$$

with $m_1 \leq n_1$ and $m_2 \leq n_2$.

Using **Lemma 2.4** (`unobservable` preservation) and **Lemma 2.5** (indistinguishability of `unobservable` step), we can deduce any intermediary continuation appearing during both executions (2.5) are unobservable. We can also deduce that the intermediary environments are pairwise indistinguishable. Thus: $\rho_1' \simeq_{\tau_{obs}} \rho_2'$.

Finally, we can append the k' continuation (this does not impact the executions as they both terminated in a fixed number of steps) to show that:

$$\begin{aligned} \langle \rho_1,\, k'_1 \cdot k' \rangle &\longmapsto^{m_1} \langle \rho'_1,\, k' \rangle \\ \langle \rho_2,\, k'_2 \cdot k' \rangle &\longmapsto^{m_2} \langle \rho'_2,\, k' \rangle \end{aligned} \tag{2.6}$$

We conclude by applying the induction hypothesis on states $\langle \rho'_1,\, k' \rangle$ and $\langle \rho'_2,\, k' \rangle$.

□

This proof concludes this section, in which we proved the soundness of the Secure Flow type-system, in the case of a small-step semantics with continuations.

2.4 Type Preserving Compilation

In the previous sections of this chapter, we introduced the NI policy, and then presented a secure flow type-system. We then proved that this type-system enforces the NI policy, i.e., that any well-typed program is non-interferent:

$$\forall p, \quad \Gamma \vdash p : \tau \implies \mathrm{NI}(p)$$

This is a strong result, but it is not satisfying when considering the compilation of a program. Indeed, even if a source program is proved to be non-interferent, there is *a priori* no guarantee that this policy will be preserved by a compiler. This is a property that needs to be proved about a compiler. Formally, a transformation T preserves the NI policy if:

$$\forall p, \quad \mathrm{NI}(p) \implies \mathrm{NI}(\mathcal{T}(p))$$

However, as the NI policy is only enforced by the secure flow type-system, we can restrict this preservation property to well-typed programs only. In other words, we want to prove that any well-typed program p, which is then non-interferent, is compiled to $T(p)$, also non-interferent. A way to prove this result is to prove that the typing is preserved by the transformation:

$$\forall p, \quad \Gamma \vdash p : \tau \implies \Gamma \vdash \mathcal{T}(p) : \tau$$

Indeed, the result above allows to conclude as $T(p)$ being well-typed implies that $T(p)$ is non-interferent. This approach then consists in turning the compiler into a type-preserving compiler. A compiler being usually split into several passes, each of them needs to be proved to preserve the type system. In other words, the whole compilation chain needs to preserve the type system.

However, the criteria above is very restrictive and can be relaxed. First, we do not need the transformed program to have the same type as the source program. Indeed, we only need to prove that the transformed program is typed by any type τ' to imply it

is non-interferent. In other word, we only need to verify that the *typability* is preserved. Second, expecting the transformed program to be typable in the same environment Γ is a strong expectation. Indeed, a transformation could be allowed to modify the security level of a variable. Indeed, it does not pose any security issue to transform a source public variable into a private variable, and some transformations may have to perform such modifications, as we will see during this section. However, the security level of a variable should never be reduced by a transformation, to avoid the introduction of information flow during the transformation. Formally, the transformed typing environment Γ' must verify $\forall x, \Gamma[x] \leq \Gamma'[x]$. We then define our criteria for a typability-preserving transformation as follows:

Definition 2.12: Typability-preserving transformation.

A transformation $\mathcal{T}$ is said to be typability-preserving if for any program p such that:

$$\Gamma \vdash p : \tau$$

we can find a type τ' and a typing environment Γ' verifying $\forall x, \Gamma[x] \leq \Gamma'[x]$, such that:

$$\Gamma' \vdash \mathcal{T}(p) : \tau'$$

On the one hand, designing a typability-preserving compiler is interesting, as it allows the compiler to be aware of the precise security level of each variable it manipulates at every level of the compilation chain. Therefore, the compiler may take advantage of this information to compile differently a piece of code, depending on the type-system. For example, if an optimization breaks the NI policy (by introducing explicit or implicit flows), the compiler may still decide to optimize if the security levels of the variables allow it. If the compiler was not aware of the security levels, it could not give such special treatment, and would most likely have to restrict its optimization.

On the other hand, designing a Secure Flow preserving compiler is a very challenging task. At source level, on a simple memory model, the Secure Flow type-system is naturally defined as we did previously in this chapter. However, for a low level language and memory model, this is much more challenging. A low level memory level is usually a simple array of bytes, that contains all the variables manipulated by the program. This then requires to design a more complex security type-system, as variables are not identified by a unique name but by an address in a memory. Such a type-system then needs to keep track of the variables through addresses in the memory and subtle reasoning on the indices of arrays, which requires complex analyses such as alias analysis.

The goal of this section is to highlight the limitations of a security type system such as the one previously introduced in this chapter. We show how this type-system is not suitable

for low level languages and memory model. To this end, we introduce a transformation emulating a low-level program transformation. The salient feature of this transformation is that it concatenates the memory into one big array of integers, similarly to low-level memory models. We then show that this transformation can not be proved to be typability-preserving.

2.4.1 A Problematic Transformation: Array Concatenation

We introduce a transformation $\overset{a_2}{\underset{a_1}{\Rrightarrow}}$, where a_1 and a_2 are two array identifiers. We define it as a relations between programs. $p \overset{a_2}{\underset{a_1}{\Rrightarrow}} p'$ means that p is transformed into p'. The goal of this transformation is to concatenate arrays a_1 and a_2 into a_1. More specifically, if a_1 and a_2 are arrays of respective length n_1 and n_2, a_1 is transformed into an array of length $n_1 + n_2$, where the n_1 first elements remain untouched, and the n_2 following are the elements from a_2. Meanwhile, after the transformation, a_2 is never referred to and can be safely removed from the environment. In order to infer the length of arrays a_1 and a_2, we also use a type-system, different from the Secure Flow type-system. As this is a simple standard type-system, we do not detail its definition. We simply write $\Delta(a) = n$ when array a has length n in the source program.

The transformation $\overset{a_2}{\underset{a_1}{\Rrightarrow}}$ is defined as follows:

Definition 2.13: Array concatenation transformation.

$$\frac{\Delta(a_1) = n_1 \qquad e \overset{a_2}{\underset{a_1}{\Rrightarrow}} e'}{a_2\,[e] \overset{a_2}{\underset{a_1}{\Rrightarrow}} a_1\,[e' + n_1]} \qquad \frac{a \neq a_2 \qquad e \overset{a_2}{\underset{a_1}{\Rrightarrow}} e'}{a\,[e] \overset{a_2}{\underset{a_1}{\Rrightarrow}} a\,[e']}$$

$$\frac{\Delta(a_1) = n_1 \qquad e_1 \overset{a_2}{\underset{a_1}{\Rrightarrow}} e_1' \qquad e_2 \overset{a_2}{\underset{a_1}{\Rrightarrow}} e_2'}{(a_2\,[e_1] \;:=\; e_2) \overset{a_2}{\underset{a_1}{\Rrightarrow}} (a_1\,[e_1' + n_1] \;:=\; e_2')} \qquad \frac{a \neq a_2 \qquad e_1 \overset{a_2}{\underset{a_1}{\Rrightarrow}} e_1' \qquad e_2 \overset{a_2}{\underset{a_1}{\Rrightarrow}} e_2'}{(a\,[e_1] \;:=\; e_2) \overset{a_2}{\underset{a_1}{\Rrightarrow}} (a\,[e_1'] \;:=\; e_2')}$$

We omit the other cases, that simply recursively apply the transformation to sub-expressions and sub-statements.

We give below an example of this transformation, applied to a simple program:

Example 6: Example of array concatenation transformation.

We consider the following program, where `t1` and `t2` are arrays, both of length 10. `x`, `y`, are `z` are integer variables.

```
x = t1[2];
y = t2[3];
t2[4] = z;
```

Applying the transformation $\overset{t_2}{\underset{t_1}{\Longrightarrow}}$ yields the following resulting program. `t1` is now the result of the concatenation of `t1` and `t2`, and has a length of 20.

```
x = t1[2];
y = t1[13];
t1[14] = z;
```

Even though we consider here a high-level language, this transformation is designed to imitate what happens at low level in a compiler. Indeed, this transformation combines two arrays into one larger array. By successively applying this transformation to every array in the initial memory, the resulting memory will contain one single array, which is close to what a low-level memory model may look like in a realistic compiler.

As the transformation modifies the memory layout, it also needs to modify the input environment (σ) for the transformation to be semantically correct. We define it as follows:

Definition 2.14: Concatenation of environments.

$$\frac{\Delta(a_1) = n \qquad \sigma[a_1] = [v_1; \ldots; v_n] \qquad \sigma[a_2] = [u_1; \ldots; u_m] \qquad \sigma'[a_1] = [v_1; \ldots; v_n; u_1; \ldots; u_m] \qquad \forall a \notin \{a_1, a_2\}, \sigma[a] = \sigma'[a]}{\sigma \overset{a_2}{\underset{a_1}{\Longrightarrow}} \sigma'}$$

2.4.2 Limitations of the Type-system

We now state a few results about the transformation $\overset{a_2}{\underset{a_1}{\Longrightarrow}}$. The first lemma is the semantic correction of the transformation.

Lemma 2.9: Transformation correct.

Let p be a program and p' its transformed program such that $p \overset{a_2}{\underset{a_1}{\Longrightarrow}} p'$. We consider two input environments σ_1 and σ'_1, such that $\sigma_1 \overset{a_2}{\underset{a_1}{\Longrightarrow}} \sigma'_1$. Assuming $\langle p, \sigma_1 \rangle \Downarrow \sigma_2$, then we can find σ'_2 such that $\langle p', \sigma'_1 \rangle \Downarrow \sigma'_2$ and $\sigma_2 \overset{a_2}{\underset{a_1}{\Longrightarrow}} \sigma'_2$.

This is a standard result that we do not detail this proof here as we rather focus on the typability-preservation of this transformation. In fact, we show that this transformation does not preserve the typability-preservation criteria defined in **Definition 2.12**, hence the following theorem:

Theorem 2.10: The concatenation transformation is not type preserving.

The underlying idea is that there exists source programs that well-typed, but whose transformed program cannot be typed. The problems appear when the transformation has to concatenate arrays of different security levels. This is highlighted in the following proof.

Before detailing the proof, we first extend our language with a usual `print` instruction. We do not formally detail its semantics as is it standard - defining it typically done by extending the semantics with a list of observable events, and every call to the `print` instruction append a value to the list.

The definition of NI could be naturally modified, to ensure that the list of observable events is independent from secret data. The interesting feature comes from the fact that the `print` instruction must then force the security level of a variable to be public, as printing as high security variable would trivially break the NI policy. This would then lead to modify the secure flow type-system accordingly, in order to prevent private variable to be printed, by introducing the following rule:

$$\Gamma \vdash \texttt{print}(x) : \tau_{obs}$$

We use this newly introduced instruction to find a counterexample in the following proof, in which the security level is forced to be public throughout the transformation. We now detail the proof of **Theorem 2.10**:

Proof of Theorem 2.10.

We assume the transformation $\overset{a_2}{\underset{a_1}{\Longrightarrow}}$ to be typability-preserving. We show that this leads to a contradiction.

We introduce several elements:

— let x be a variable identifier, that we assume to be different from both a_1 and a_2.

— let L and H be two security level. We assume to have $L \leq \tau_{obs}$ and $\tau_{obs} \not\leq H$ (intuitively, L is public, H is private).

— let p be the following program:

$$x \texttt{ := } a_2\texttt{[1]; print(}x\texttt{)}$$

— let Γ be the following typing environment:

$$\Gamma = [a_1 \leftarrow H][a_2 \leftarrow L][x \leftarrow L]$$

i.e., a_1 is private, and a_2 and x are public in Γ.

First we can verify that p is well typed in Γ. Indeed, the assignment does not perform an explicit flow, and the printed variable x is public.

Second, p is transformed by $\overset{a_2}{\underset{a_1}{\Longrightarrow}}$ into the following program that we call p':

$$x \texttt{ := } a_1\texttt{[1 + n]; print(}x\texttt{)}$$

with $n = \Delta(a_1)$.

By assumption, as $\overset{a_2}{\underset{a_1}{\Longrightarrow}}$ is typability-preserving, we can find τ' and Γ' such that:

$$\Gamma' \vdash p' : \tau' \quad \text{and} \quad \forall x, \Gamma[x] \leq \Gamma'[x]$$

We make the following conclusions:

— The type of a_1 cannot the lowered during the transformation, thus $H = \Gamma[a_1] \leq \Gamma'[a_1]$.

— Variable x cannot remain public, to avoid an explicit flow. Therefore, we must have $\Gamma'[a_1] \leq \Gamma'[x]$.

— As variable x is printed, its type must be lower than τ_{obs}. Thus: $\Gamma'[x] \leq \tau_{obs}$.

As, by definition we also have $\tau_{obs} \not\leq H$, we deduce:

$$H \leq \Gamma'[a_1] \leq \Gamma'[x] \leq \tau_{obs} \not\leq H$$

which is a contradiction and then concludes the proof.

□

In order to make the concatenation transformation typability-preserving, we would need to introduce a more precise, but more complex type-system. This type-system would require to keep precise track of each individual value, instead of only assigning a global

security level to an array. This type-system would then require to reason on the indices of the memory locations. As a memory locations may be referred to more than one pointer, it requires to perform an alias analysis which limits the precision of the type-system. This is a very complex task, and as far as we know, no real-life compiler on a realistic low-level language manages to perform this goal.

We use this section as a motivation for our methodology in this document. In the following chapter, we will focus on security policies, that are similar to NI, and which may be enforced with similar type-systems. We will more specifically study the preservation of these polices during a transformation of program. However, we follow a different approach than the sketch presented in this section. We will focus on methodologies that do not rely on the preservation of a type-system by a compiler, in order to avoid the problems presented here. The policies we will focus on are instances of a generic security policy, called Observational Non-Interference (ONI). The next and last section of this chapter will focus on its definition.

2.5 Extension to Observational Non-Interference

2.5.1 Instrumenting a Semantics with Leakages

In order to formally define the ONI policy, we first need to introduce the notion of leakage. This is usually done by instrumenting the semantics of the language, i.e., by defining a semantics that outputs the leakage of an execution. We do not fully define it yet, but we introduce a big-step semantics for our language $\mathscr{L}$, written as:

$$\langle p, \sigma \rangle \Downarrow (\sigma', \ell)$$

It reads as follows: program p executes from input environment σ down to environment σ'. The execution produces a leakage ℓ. With this definition, the ONI policy can be defined as follows:

Definition 2.15: Observational Non-Interference (ONI).

Let p be a program and τ_{obs} a security level. We assume to have initial environments σ_1 and σ_2 such that $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$, and the executions $\langle p, \sigma_1 \rangle \Downarrow (\sigma'_1, \ell_1)$ and $\langle p, \sigma_2 \rangle \Downarrow (\sigma'_2, \ell_2)$. If we have that $\ell_1 = \ell_2$, we then say that p is ONI.

This definition of ONI is dependent on the way we define the leakage in the semantics of the language. With different instances of leakage, the ONI policy can be instantiated into different security policies. As an example, the Cryptographic Constant-Time (CCT) policy can be defined as an instance of ONI. In the remaining of this document, we will

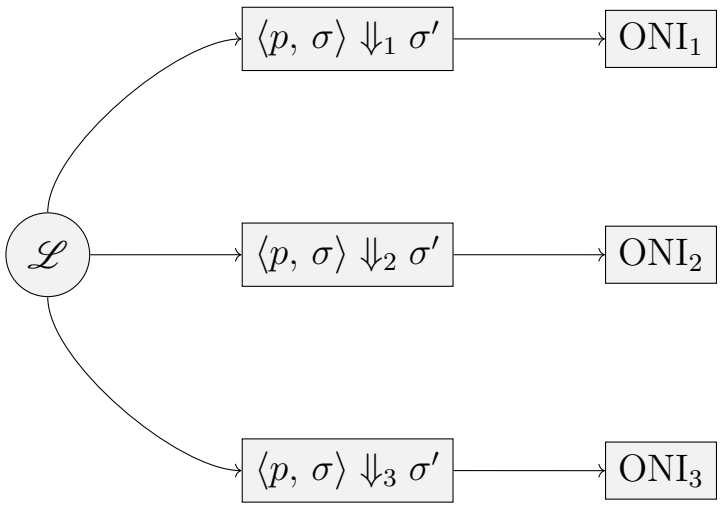


Figure 2.1 – Definition workflow of 3 instances of ONI, using 3 semantics.

work on three different instances of ONI, and specifically focus on the preservation of these policies during a program transformation. We will also precisely compare these policies.

Defining three instances of ONI requires to define three semantics for $\mathscr{L}$ that are very similar. This is summed up in the diagram presented in Figure 2.1. In this figure, we illustrate how we could introduce three semantics for our language $\mathscr{L}$, denoted as $\Downarrow_1$, $\Downarrow_2$, and $\Downarrow_3$, and these semantics to define three different instances of ONI. This is not convenient, as it requires to introduce several definitions that only differ on a few elements. But it becomes really clumsy if we want to compare theses policies, and we would have to reason on executions that use different semantics for a same program.

For these reasons, we choose a slightly different approach. We define a unique semantics for our language, that contain all the leakage we need to define the three policies we are interested in. We call this leakage the *raw leakage* of an execution. Then, to define an ONI policy, we define a projection function π, that takes a leakage as input and filters it to only keep the leakage relevant for this instance of ONI. The projected leakage is what we use to define ONI. Formally, we use the following definition:

Definition 2.16: Observational Non-Interference (ONI).

Let π be a projection function. Let p be a program and τ_{obs} a security level. We assume to have initial environments σ_1 and σ_2 such that $\sigma_1 \simeq_{\tau_{obs}} \sigma_2$, and the executions $\langle p, \sigma_1 \rangle \Downarrow (\sigma'_1, \ell_1)$ and $\langle p, \sigma_2 \rangle \Downarrow (\sigma'_2, \ell_2)$. If we have that $\pi(\ell_1) = \pi(\ell_2)$, we then say that p is ONI with respect to π. We denote it as $\mathrm{ONI}_\pi(p, \tau_{obs})$.

Using this definition, defining several instances of ONI becomes much easier, as we only need to define the corresponding projection functions. It is also easier to compare

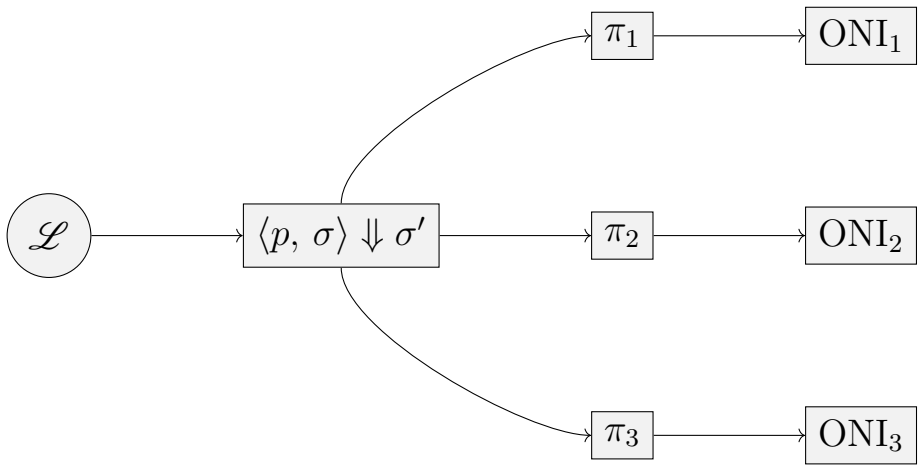


Figure 2.2 – Definition workflow of 3 instances of ONI, using 1 semantics and projection functions.

these policies, as they are all based on the same semantics, that emits the raw leakage. This is summed up in Figure 2.2. In this figure, we illustrate one single semantics, denoted as $\Downarrow$ is used to define three instances of ONI, using three projection function π_1, π_2 and π_3. Using this approach, defining a new instance of ONI only comes down to simply defining a new projection function π.

In our work, we follow this second approach. The instrumented semantics of $\mathscr{L}$ will be incrementally augmented throughout this document, by introducing new kind of raw leakages when needed. The first version of this semantics will be defined in the following section, in which we define the CCT policy as an instance of ONI.

2.5.2 Defining the Constant-Time Policy as an Instance of ONI

The CCT policy focuses on the control-flow of an execution, and on the memory accesses performed during this execution. More precisely, to be considered CCT, a program must never perform a branching that depend on a secret value, nor any memory access whose location depends on a secret value. To define CCT as an instance of ONI, we can naturally define the emitted leakage as the control-flow and the sequence of memory accesses performed by the execution. This definition ensures that observing the leakage (i.e., the control-flow and the memory accesses) does not allow to learn any information on the secret of the program.

We define a leakage as a list events. Whenever a program executes a branching or a memory access, a corresponding event is emitted in the leakage. We then need two kinds of events, to records both the control-flow and the sequence of memory accesses. The set of events $\mathcal{E}$ is defined as follows:

Definition 2.17: Set of event $\mathcal{E}$.

$$\langle event \rangle \ ::= \ e_B(b) \mid e_M(id, n)$$

$e_B(b)$ is an event emitted at a branch, where b is a boolean value. b is the truth value to which to condition has evaluated to. $e_M(id, n)$ is an event emitted at a memory access, where id is an array identifier, and n is the index of the value accessed in this array.

We now define a leakage as a list of events, using notation $e \cdot \ell$ to add event e to leakage ℓ, and notation $\ell_1 \cdot \ell_2$ to denote the concatenation of leakages. ϵ is an empty leakage.

Example 7: Example of leakage.

For example, if an execution emits leakage $e_B(\text{true}) \cdot e_M(x, 2) \cdot e_b(\text{false})$, this means that the execution first encountered a branching, whose condition evaluated to true. Then, the execution encountered a memory access to the index 2 of array x. Then, another branching whose condition evaluated to false.

We now provide a complete big-step semantics, in the following definitions, both for expressions and statements.

Definition 2.18: Evaluation of expression $\langle e, \sigma \rangle \Downarrow (v, \ell)$.

VAR
$$\frac{\sigma[x] = v}{\langle x, \sigma \rangle \Downarrow (v, \epsilon)}$$

ARRAY
$$\frac{\sigma[x] = [v_1; ...; v_n; ...v_m] \qquad \langle e, \sigma \rangle \Downarrow (n, \ell) \qquad 1 \leq n \leq m}{\langle x\,\texttt{[}e\texttt{]}, \sigma \rangle \Downarrow (v_n, (\ell \cdot e_M(x, n)))}$$

CONST
$$\frac{}{\langle n, \sigma \rangle \Downarrow (n, \epsilon)}$$

OP_BIN
$$\frac{\langle e_1, \sigma \rangle \Downarrow (v_1, \ell_1) \qquad \langle e_2, \sigma \rangle \Downarrow (v_2, \ell_2)}{\langle e_1 \diamond e_2, \sigma \rangle \Downarrow (v_1 \diamond v_2, (\ell_1 \cdot \ell_2))}$$

Definition 2.19: Evaluation of statement $\langle s, \sigma \rangle \Downarrow (\sigma', \ell)$.

SKIP
$$\frac{}{\langle \texttt{skip}, \sigma \rangle \Downarrow (\sigma, \epsilon)}$$

ASSIGN
$$\frac{\langle e, \sigma \rangle \Downarrow (v, \ell)}{\langle id\texttt{:=}e, \sigma \rangle \Downarrow (\sigma[id \leftarrow v], \ell)}$$

ASSIGN_ARRAY

$$\frac{\sigma[x] = [v_1; ...; v_n; ...v_m] \qquad \langle e_1,\, \sigma\rangle \Downarrow (n, \ell_1) \qquad \langle e_2,\, \sigma\rangle \Downarrow (v', \ell_2) \qquad 1 \leq n \leq m \qquad v' \in \mathbb{Z}}{\langle a\,\texttt{[}e_1\texttt{]}\,\texttt{:=}e_2,\, \sigma\rangle \Downarrow (\sigma[a \leftarrow [v_1; \ldots; v_{n-1}; v'; v_{n+1}; \ldots; v_m]], (\ell_1 \cdot \ell_2 \cdot e_M(x, n)))}$$

SEQ

$$\frac{\langle p_1,\, \sigma\rangle \Downarrow (\sigma', \ell_1) \qquad \langle p_2,\, \sigma'\rangle \Downarrow (\sigma'', \ell_2)}{\langle (p_1\texttt{;}\ \ p_2),\, \sigma\rangle \Downarrow (\sigma'', (\ell_1 \cdot \ell_2))}$$

IF_TRUE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n \neq 0 \qquad \langle p_1,\, \sigma\rangle \Downarrow (\sigma', \ell_1)}{\langle \texttt{if(}e\texttt{)}\ \texttt{\{}p_1\texttt{\}}\ \texttt{else}\ \texttt{\{}p_2\texttt{\}},\, \sigma\rangle \Downarrow (\sigma', (\ell \cdot e_B(\text{true}) \cdot \ell_1))}$$

IF_FALSE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n = 0 \qquad \langle p_2,\, \sigma\rangle \Downarrow (\sigma', \ell_2)}{\langle \texttt{if(}e\texttt{)}\ \texttt{\{}p_1\texttt{\}}\ \texttt{else}\ \texttt{\{}p_2\texttt{\}},\, \sigma\rangle \Downarrow (\sigma', (\ell \cdot e_B(\text{false}) \cdot \ell_2))}$$

WHILE_TRUE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n \neq 0 \qquad \langle p,\, \sigma\rangle \Downarrow (\sigma', \ell') \qquad \langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma'\rangle \Downarrow (\sigma'', \ell'')}{\langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma\rangle \Downarrow (\sigma'', (\ell \cdot e_B(\text{true}) \cdot \ell' \cdot \ell''))}$$

WHILE_FALSE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n = 0}{\langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma\rangle \Downarrow (\sigma, (\ell \cdot e_B(\text{false})))}$$

The key feature of these semantics is the emission of leakage. We can indeed notice that we emit event $e_M(x, n)$ when the index n of array x is accessed, both read (rule ARRAY) and write (rule ASSIGN_ARRAY). In rules IF_TRUE and WHILE_TRUE, we emit event $e_B(\text{true})$, and in rules IF_FALSE and WHILE_FALSE, we emit event $e_B(\text{false})$. In all the rules, the leakage emitted by sub-expressions and sub-statements are concatenated.

Last, CCT as ONI_π, with π the identity function. In other words, the projection function π keeps the whole leakage for now. Projection functions we see more use in chapter 4, where we will define two other instances of ONI.

2.5.3 Enforcing CCT with a Type-System

The Secure Flow type-system introduced in section 2.2 can easily be adapted to enforce the CCT policy. Indeed, we want to forbid both memory accesses and branchings that dependent on a secret value. We describe how to achieve this goal. We first define a new type-system, called the Constant-time type-system, which is a direct adaptation to the

one presented in section 2.2. This new type-system is defined with respect to a security level τ_{obs}. It is defined as follows, both for expressions and statements:

Definition 2.20: Constant-time type-system for expressions $\Gamma \vdash e : \tau$.

$$\text{CONST}\ \frac{}{\Gamma \vdash n : \tau} \qquad \text{VAR}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{ARRAY}\ \frac{\Gamma[a] = \tau_{obs} \qquad \Gamma \vdash e : \tau_{obs}}{\Gamma \vdash a\,[e] : \tau_{obs}}$$

$$\text{OP_BIN}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 \diamond e_2) : \tau} \qquad \text{SUBTYPE_E}\ \frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2}$$

Definition 2.21: Constant-time type-system for statements $\Gamma \vdash s : \tau$.

$$\text{ASN}\ \frac{\Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash (x\ \texttt{:=}\ e) : \tau} \qquad \text{ARRAY_ASN}\ \frac{\Gamma[a] = \tau_{obs} \qquad \Gamma \vdash e_1 : \tau_{obs} \qquad \Gamma \vdash e_2 : \tau_{obs}}{\Gamma \vdash (a\,[e_1]\ \texttt{:=}\ e_2) : \tau_{obs}}$$

$$\text{SEQ}\ \frac{\Gamma \vdash s_1 : \tau \qquad \Gamma \vdash s_2 : \tau}{\Gamma \vdash (s_1; s_2) : \tau} \qquad \text{IF}\ \frac{\Gamma \vdash e : \tau_{obs} \qquad \Gamma \vdash s_1 : \tau_{obs} \qquad \Gamma \vdash s_2 : \tau_{obs}}{\Gamma \vdash (\texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) : \tau_{obs}}$$

$$\text{SKIP}\ \frac{}{\Gamma \vdash \texttt{skip} : \tau} \qquad \text{WHILE}\ \frac{\Gamma \vdash e : \tau_{obs} \qquad \Gamma \vdash s : \tau_{obs}}{\Gamma \vdash (\texttt{while}\, e\, \texttt{do}\, s) : \tau_{obs}} \qquad \text{SUBTYPE_S}\ \frac{\Gamma \vdash s : \tau_2 \qquad \tau_1 \leq \tau_2}{\Gamma \vdash s : \tau_1}$$

Note that the Constant-time type-system is very close to the Secure Flow type-system. The only difference is that memory accesses and branchings are only allowed if they have type τ_{obs}, thus forbidding secret dependent branchings and memory accesses.

We can now state the soundness of this type system, capturing the idea that any well-typed program is CCT-secure.

Theorem 2.11: Soundness of the Constant-time type-system.

Let p be a program. If p is well-typed, i.e., $\Gamma \vdash p : \tau$, then p is CCT-secure with respect to the security level τ_{obs}, i.e., $\mathrm{CCT}(p, \tau_{obs})$.

We provide a mechanized proof of this theorem, although we do not detail it in this document.

2.5.4 Preservation of ONI

We conclude this section by defining the preservation of the ONI policy by a transformation. A transformation $\mathcal{T}$ is ONI-preserving when given a ONI-secure program P, then $\mathcal{T}(P)$ is ONI-secure as well. The underlying hypothesis is that P starts from two indistinguishable states and so does $\mathcal{T}(P)$. However, these two pairs of state are not related. In the same way, there is no relation between the two leakages observed during the two executions of P and $\mathcal{T}(P)$. This definition expresses the preservation of an ONI policy, but it is too general to be proved by a simple induction. For that reason, we define in chapter 4 less general preservation properties that fit to our program transformations and are easier to prove.

Definition 2.22: Preservation of ONI.

Let π be a leakage projection. A transformation $\mathcal{T}$ is ONI_π-preserving when, for any program p:

$$\mathrm{ONI}_\pi(p) \implies \mathrm{ONI}_\pi(\mathcal{T}(p))$$

2.6 Conclusion

In this chapter, we first presented the non-interference policy, then a secure flow type system. We then didactically proved this type system to enforce the non-interference policy, in the case of a language with a small-step semantics with continuations.

Next, we presented the problem of the preservation of this policy during the compilation of a program, with a type-preserving approach. We presented the limitations of this approach. In the following chapters of this document, we will focus on the preservation of policies similar to non-interference, by using approaches that do *not* rely on a type preserving compiler.

Last, we presented the Observational Non-Interference (ONI) policy, as well as the Cryptographic Constant-Time (CCT) policy, that we defined as an instance of ONI. The next chapter will be devoted to the study of the preservation of CCT by the CompCert compiler. In chapter 4, we will focus on a different instance of ONI, called Constant-Resource (CR)

Throughout this chapter, we showed that type-systems are powerful tools that can be used to enforce security policies such as NI and ONI at source level. However, we also

saw that such type-systems are hard to preserve during a program transformation. In the following chapters, we will focus on the preservation of ONI policies, using methodologies that do not rely on the preservation of a type-system.

Chapter 3

A Constant-Time Preserving CompCert Compiler

Formally verified compilers, such as the CompCert compiler, guarantee the preservation of the observable behavior of programs during the compilation. The absence of correctness bugs introduced by the compiler is then guaranteed, but this does not encompass security bugs that may still be introduced during the compilation. For instance, the constant-time policy may easily be broken by a compiler. We recall that a program is said to be constant-time if its control-flow and memory accesses do not depend on the secret values of the program. Therefore, as the compiler is not aware of the taints of the program (i.e., which values are secret or not), any introduction of `if` statements or memory accesses during the compilation may break the constant-time policy. Several common compilation passes and optimizations, such as optimizations on multiplications, lazy operations, or register allocation are thus likely to break this policy. As a matter of fact, most standard C compilers break this policy, as we will see in Section 3.1.

Yet, a constant-time preserving compiler is an appealing goal to achieve. Indeed, even though the constant-time policy may seem conceptually simple, according to its informal definition provided above, writing correct and secure constant-time code is a very challenging task. To help practitioners, many tools have been developed to check that cryptographic libraries adhere to the constant-time discipline. As the compiler can not be trusted to preserve the policy, these tools target low-level machine code, which presents several drawbacks. First, the analysis is less precise than at source level. Second, the result of the analysis is usually difficult to understand by the programmer. A constant-time preserving compiler would then allow to delegate this checking step to the source level, eliminating these drawbacks, by then relying on the compiler to preserve the policy down to the low-level code.

Several constant-time preserving compilers exist in the literature, such as FaCT and Jasmin. However, they all target custom domain specific languages. Our goal here is to target a realistic C compiler: the CompCert compiler. This challenge raises two questions. First, how can we modify the CompCert compiler in order to make it constant-time preserving. Second, how can we formally prove that this modified CompCert preserves this policy. In this chapter, we will exclusively focus on the former, while details on the

later question are orthogonal to our contribution, and can be found in [Bar+19].

Specifically, we will first present examples of standard C compilers that break the constant-time policy in Section 3.1. Next, we will present the selection operation, a program construct commonly used by practitioners to write constant-time code in Section 3.2. We will then present some specific background on the CompCert compiler, detailing its architecture in Section 3.3. The last sections will be devoted to the changes we made to the compiler.

Note: Throughout this chapter, we will study two distinct notions that share a very similar name. First, we will focus on several passes of the compilers, including one named *Instruction Selection*. Second, we will focus on a specific instruction, called the *selection operation*. In order to avoid ambiguity, we adopt the following notations:

— Passes (including Instruction Selection) will be referred to using a bold style: **Instruction Selection**.
— Instructions (including the selection operation) will be referred to using a typewriter style and their implementation name: `select`.
— Intermediate languages will be referred to using a sans-serif style: CompCert C.

3.1 Examples of Standard Compilers Breaking the Constant-time Policy

During the compilation of a program, the different transformations occurring may modify the intermediate computations, as long as the overall observable behavior of the program is preserved. A program may be modified either to improve the performance of the program, but may also sometimes require to be modified in order to bypass the target language limitations, such as instructions present in the source language that do not exists in the target language. We present an example for both cases, that breaks the constant-time policy during compilation.

3.1.1 Optimization Breaking Constant-time

Our first example is an optimization performed on arithmetic operations. Consider the operation `b * x` where `b` is a boolean value, and `x` is an integer value. The result of this operation is semantically equivalent to `x` if `b` is true, and `0` otherwise.

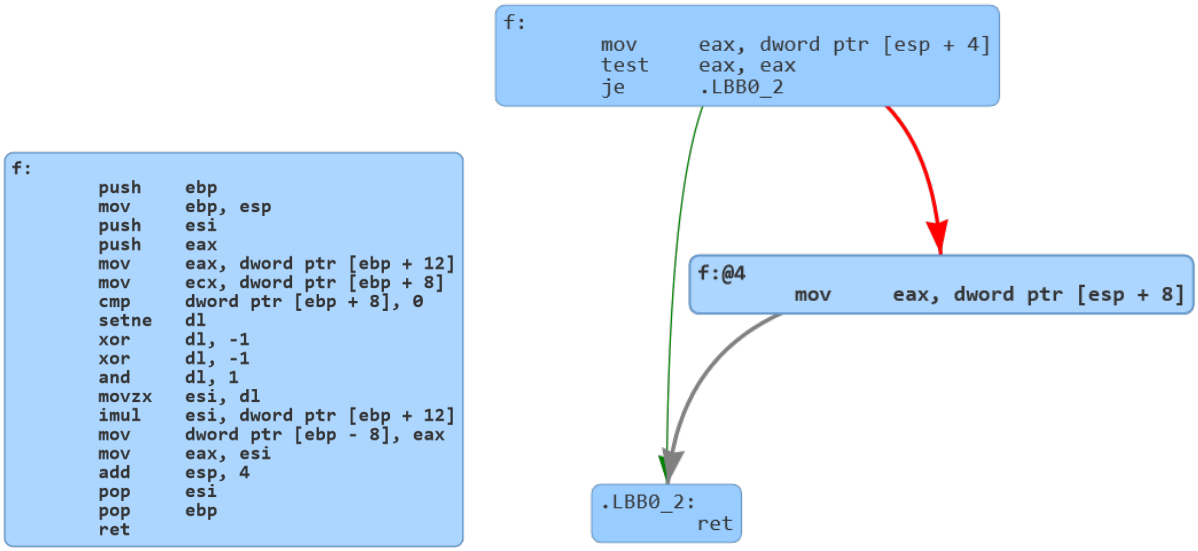
Consider the following C function:

Note that we force the value `b` to be a boolean value by applying the negation operator `!` twice. We compile it using the Clang compiler on the latest version (version 11), targeting a 32 bits x86 architecture. Figure 3.2 presents the generated assembly code. We first compile it without any compiler optimization. The produced assembly code is presented

```c
int f(int b, int x) {
    return (!!b) * x;
}
```

Figure 3.1 – A C function multiplying a boolean and an integer.

in (Figure 3.2a). Its interesting feature is that is does not contain any introduced jump instruction, and thus preserves the constant-time policy on this example. We then compile it using the level 1 optimization. The produced code (Figure 3.2b) is equivalent in C source code to the instruction `if(b) { return x; } else { return 0; }`. This time, the code produced contains a jump instruction, which means that the compiler introduced a branch during the compilation. Therefore, Clang explicitly broke the constant-time policy. This kind of behavior must be prevented in a constant-time preserving compiler.



(a) Without optimization (`-O0`)

(b) With level 1 optimization (`-O1`)

Figure 3.2 – Assembly code generated by Clang.

3.1.2 Floating Point Value Conversion

Our second example targets a limitation of the x86 architecture about conversion between floating-point values and integer values. The C language provides several cast operations between these types of values, which are handled at machine level by specific instructions. However, some of these conversion instructions do not exist at machine level

and thus require a workaround that may introduce a branch during the compilation. We observe this behavior on the latest version (version 11) of the GCC compiler.

```
unsigned f1(float x) {
    return x;
}
```

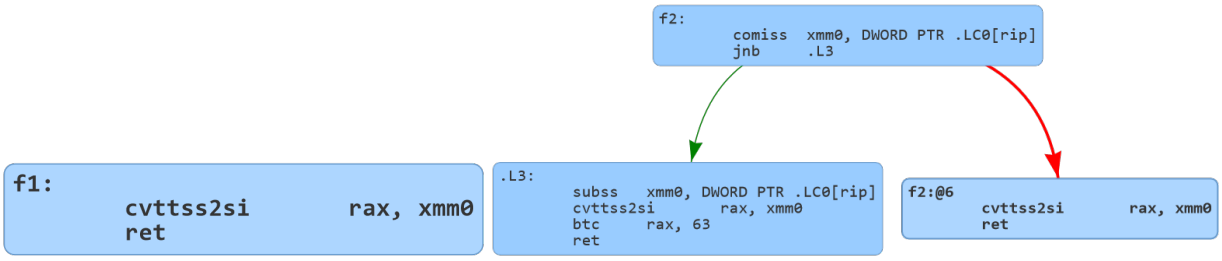
(a) Conversion from float to unsigned int.

```
unsigned long f2(float x) {
  return x;
}
```

(b) Conversion from float to unsigned long int.

Figure 3.3 – Conversions between float and unsigned integer values.

Consider the C functions presented in Figure 3.3. These functions are similar and perform a simple cast from a floating point value to respectively an unsigned integer and an unsigned long integer. Still, compiling these functions with GCC, using the level 1 optimization yields very different machine code. The generated assembly code is presented in Figure 3.4. We can notice that the conversion in function `f1` is directly compiled into one specific x86 instruction. However, the compilation of `f2` introduced a branch, and thus broke the constant-time policy. The branch has been inserted by the compiler as there is no native x86 instruction to convert a float to a value of type unsigned long int. Instead, it is implemented using the float to unsigned int instruction and a branch.



(a) Conversion from float to unsigned int.

(b) Conversion from float to unsigned long int.

Figure 3.4 – Assembly code generated by GCC `-O1`.

3.1.3 Conclusion

We showed two examples of very simple programs illustrating that the latest version of standard C compiler (Clang and GCC) may break the constant-time policy. A constant-time preserving must avoid these behaviors in order to strictly preserve the constant-time policy. The examples presented in this section will be used in the rest of this chapter.

3.2 A Constant-time Selection Operation

In a constant-time program, the control-flow must not depend on the secret value of the program. This means that a program as simple as `if(c) x = 1; else x = 2;` is forbidden, when `c` is a secret. Still, when writing cryptographic code, it is useful to be able to semantically perform a choice between two values, depending on a secret value, in a constant-time way. This is called a *selection operation*, and will denote it `ctselect(cond, e1, e2)`. The operation `x = ctselect(cond, x1, x2)` is semantically equivalent to `if(c) x = x1; else x = x2;`, but has to be implemented in a constant-time way.

At source level, there exists several ways to implement it in a branchless way, by using clever arithmetic operation or bitwise manipulation. Figure 3.5 presents two of these possible implementations. Such program constructs can be found in cryptographic code, however, they both present two drawbacks. First, they are less efficient than their natural implementations using an `if` statement. And most importantly, they are not guaranteed to be compiled in a constant-time way by the compiler. As presented in Section 3.1.1, the compiler may optimize these implementations by replacing the arithmetic or bitwise operations with a branch.

```
unsigned ctselect1(int cond, int x1, int x2) {
    return cond * x1 + (1 - cond) * x2;
}
```

(a) Constant-time selection with arithmetic operation.

```
unsigned ctselect1(int cond, int x1, int x2) {
    return x1 ^ ((x2 ^ x1) & (-(unsigned)cond));
}
```

(b) Constant-time selection with bitwise operation.

Figure 3.5 – Implementations of constant-time selection at source level.

A possible solution to fix this issue is to disable the optimizations responsible for this behavior, but it is actually not necessary. Indeed, not only such a selection operation is useful as source level to perform a branchless selection between values, but it may also be useful inside the compilation chain of a constant-time preserving compiler. As illustrated in Section 3.1.2, a transformation may have to introduce a branch during the compilation. Replacing these introduced branches with selection operations is thus a relevant solution.

This is the solution we adopted in order to make the CompCert compiler constant-time preserving. We introduce a selection operation `ctselect` at source level, and at

intermediary language of the compiler. This operation is shared by all the languages and the passes, and is thus unaffected by the existing optimizations of the compiler. This operation is also available to be used in any pass of the compiler. This will be detailed a lot during the following sections.

This raises one last question. At the end of the compilation chain, the selection operation must be implemented into machine code. Similarly to the source implementations of Figure 3.5, the selection operation can be implemented at low-level using arithmetic or bitwise operations. However, the x86 architecture provides a specific instruction that can serve this purpose: the `cmov` instruction. This instruction is commonly used by practitioners to write constant-time code. It has the following semantics: executing `cmov c r1 r2` will move the value of the register `r2` into the register `r1` if the condition `c` is verified. Otherwise, nothing happens. This looks like a branch, but is empirically executed in a time that does not depend on the condition. This `cmov` instruction is then commonly assumed to have a constant execution time and can be used in constant-time code. Figure 3.6 give a possible low-level implementation of the `ctselect` operation using two `cmov` instructions. We will detail the way we did it in the CompCert compiler in the following sections.

```
cmov cond r r1
cmov (!cond) r r2
```

Figure 3.6 – A low-level implementation of `r ← ctselect(cond, r1, r2)`.

3.3 Detailed Background on the CompCert Compiler

The CompCert compiler compiles C source code down to assembly. It is designed as a sequence of 20 consecutive compilation passes, and uses 8 intermediate languages. This section aims at giving on overview of these transformations and languages. Specifically, we will present details on the version 3.4 of CompCert, on which our work is based on. We will later refer to this overview when presenting our modifications on the CompCert compiler.

3.3.1 Architecture of CompCert

Figure 3.7 presents all the passes and languages of CompCert, including CompCert C (source) and ASM (target). Boxes represent the languages, and labeled arrows represent the transformations. The compiler is divided into two parts, the frontend compiler and the backend compiler.

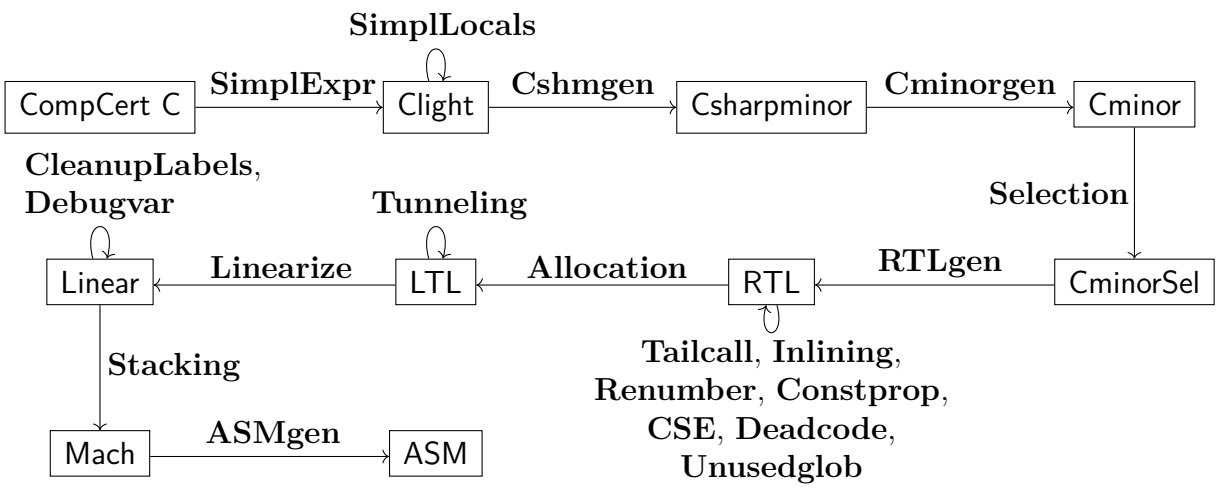


Figure 3.7 – The CompCert compilation chain.

Frontend Compiler

The frontend compiler is independent from the target architecture. It compiles programs from a source language called *CompCert C* down to *Cminor*. All the intermediate languages are structured with statements and expressions constructs. The source language is the C99 language with only a few exceptions that are detailed in the documentation of CompCert [Ler+12].

The first pass, called **SimplExpr** compiles CompCert C to Clight [BL09]. This language is a large subset of C, which mean that any valid Clight program is a valid C program. It contains only pure expressions: unlike the C language, expression can no longer contain side-effects. Therefore, in this language, function calls and assignments cannot occur in expressions but only in statements. The goal of this design is to ensure the determinism of the evaluation the language. Ternary conditions (expression such as `cond ? a : b` in the source language) are also compiled into `if` statements in Clight.

The next two languages, Csharpminor and Cminor, and their associated transformations **Cshmgen** and **Cminor** aim for several goals [BDL06]. First, the control-flow constructs of the source language (`if`, `switch`, `while`, `for`, `do...while`) are compiled to more simple low-level constructs. Second, operator overloading is resolved with type inference: type-dependent operators, such as arithmetic or conversion operators, are made explicit. Last, local variables are stack-allocated, and accesses to these variables are replaced with corresponding `load/store` operations.

Finally, we highlight here the fact that the final language, Cminor, defines a notion of operation. It encompasses all the operations that exist in the C language, such as arithmetic, boolean, bitwise, comparison or conversion operations.

Backend Compiler

The remaining of the compiler, called the backend part, is dependent on the chosen target architecture. In our work, we only focus on the x86 architecture, but we believe in can easily be adapted to other target architectures. Programs are compiled down to ASM, a language describing the abstract syntax tree of the chosen target assembly code.

The first pass, called **Selection**, connects the frontend and the backend of the compiler. This is a standard compilation pass, that aims at mapping the source operations of the Cminor language (which are the ones from the C language) to the operations supported by the target architecture. In CompCert, this pass is performed on high-level structured languages. Indeed, just like Cminor, CminorSel is structured with expressions, statements and functions. Both languages are similar, except for this notion of operation. For most operations, the translation from source to target operations is straightforward, as most of C operations have their x86 counterpart. For example, the source bitwise and operator `&` can be translated to the `and` x86 instruction. However, some source operations cannot be directly translated and thus require a workaround to be implemented. This will be detailed and discussed later.

Note that the architecture-dependent operations used in Cminor are defined and shared for every language of the backend compiler. The next pass, **RTLgen** compiles CminorSel programs to the RTL language. RTL, which stands for *register transfer language*, is a language in which programs are represented as 3-address code. This is the first language of the compilation chain which is not structured with expressions and statements, but which instead represents program as a control-flow graph.

Next, because of the simple structure of the RTL language most of CompCert's optimizations are performed at RTL level. These optimizations include for example **Constant Propagation**, **Common Subexpression Elimination**, **Deadcode Elimination** and **Function Inlining**.

The next pass is the register **Allocation** pass, which compiles a RTL program to LTL program. RTL and LTL have a similar structure, but LTL has a limited amount of register available, while RTL can manipulate infinitely many registers.

Next, the **Linearize** pass compiles program from LTL to the Linear language. In this language, the program does not longer have a control-flow graph, but is instead a list of instructions that contains conditional jumps.

Finally, during then **ASMgen** pass, the **Mach** code is transformed into assembly code. This pass truly depends on the target architecture.

3.3.2 x86 Operations in CompCert

In CompCert, the backend of the compiler is the only architecture-dependent part. Yet, most of the backend compiler is shared between the different supported architectures, and only a minimal part of the backend actually depends on the architecture. Specifically, CompCert introduces a notion of architecture-dependent operation, which roughly represents what the processor may execute in one instruction. This notion of operation is shared between all the languages of the backend. Architecture-dependent parts of the compiler mainly include:

— the definition of this set of operations,
— the generation of these operations during the **Instruction Selection** pass,
— the **Register Allocation** pass, which depends on the number of registers available on the hardware,
— the generation of assembly code from these operations during the **ASMgen** pass.

We detail here the definitions of the architecture-dependent set of operations for the x86 architecture. For brevity, we only show some illustrative cases of these operations, and omit the others.

Syntax of Operations

The syntax of the operations does not include the non-constant operands of the operation. Indeed, their definition is designed to be generic enough to be used in structured languages (such as CminorSel) or machine code (such as Mach). Therefore, these operations are meant to be applied to expressions in the former (i.e., an operation uses a list of expressions as operands), and to registers in the latter (i.e., an operation uses a list of registers as operands).

The syntax of the x86 backend operation is presented in Figure 3.8a. It consists of a list of instructions supported by the x86 architecture. In the comments of the definition, `[rd]` is the result of the operation and `[r1]`, `[r2]`, etc, are the operands.

This list includes the move instruction `Omove`, instructions setting a register to a constant value, usual 32-bit and 64-bit arithmetic and bitwise instructions, floating-point arithmetic instructions, conversion instructions and comparisons. Most of these instructions have several variations, used to support the different types of integers supported by the hardware (8-bit, 16-bit, 32-bit, 64-bit, both signed and unsigned). Some of these instructions support an immediate value as a parameter, such as the `Omulimm` instruction. These instructions are then parameterized by an integer value. Last, the comparison operation `Ocmp` is parameterized by a `condition`, whose definition is given in Figure 3.8b, and is used to represent a comparison between two registers. These conditions exist in many variation to support every size of integer, both signed and unsigned, and floating-point

value comparisons. We again only show a few of them here.

Semantics of Operations

The semantics of the operations is defined as a function that evaluates an operation to a value. A value is a type that is defined in CompCert and which is every intermediate language of the compiler. It can represent a machine integer value, a machine floating-point value, a pointer, or an undefined value, used to represent an arbitrary content.

The semantics is defined through the function `eval_operation`, and present its definition in Figure 3.9. It maps an operation `op`, a list of values `vl` (the arguments to be applied to the operation). The function also takes as parameter a global environment `genv`, that defines symbols and functions in the program, a memory `mem`, and the stack pointer `sp`. The function computes the associated resulting value, to be stored in the target register. The evaluation of an instruction may fail (e.g., a division by 0), which is captured by returning the value None.

We give the semantics of a few of the operations previously presented. To evaluate a `Omove`, the semantics excepts exactly one value (and would fail otherwise), and returns this value. Evaluating an integer constant consists in returning a value containing the constant itself. Note that all these definition are fairly simple, and rely on evaluation of values, which is shared between of the languages of CompCert. Evaluating a division `Odiv` relies on the function `Val.divs`, which may return None if `v2` is zero.

Assembly Generation

During all the following passes of the backend of CompCert, the operations are transmitted from a language to the next without any modification, until the assembly generation pass. During this pass, one specific function translates generic operations into assembly code. This function, call `transl_op`, is presented in Figure 3.10. For brevity, we only present one of the many cases of this function. The function as parameter an operation `op`, a list of operand registers `args`, a destination register `res`. This transformation is defined using continuations, which is why the function takes as parameter `k`, the rest of the assembly code, to which to generated code is prepended. The function returns the generated assembly code. We present the case of the generation of a multiplication operation `Omul`. This operation has to be applied to exactly two operand registers `a1` and `a2`; any different number of operands would fail the compilation. The generated code contains monads to handle errors values. As such, the function first checks that the first operand `a1` and the destination register `res` are the same registers. This is achieved through CompCert's previous optimizations during register allocation. Next, the compiler checks that both registers `res` and `a2` are integer registers. If this condition is verified, then the code `Pimull_rr res a2` is generated, which corresponds to `IMUL` x86 instruction.

```
Inductive operation : Type :=
| Omove              (* [rd = r1] *)

(* constants *)
| Ointconst (n: int)     (* [rd] is set to the given integer constant *)

(* 32-bit integer arithmetic: *)
| Oneg               (* [rd = - r1] *)
| Osub               (* [rd = r1 - r2] *)
| Omul               (* [rd = r1 * r2] *)
| Omulimm (n: int)      (* [rd = r1 * n] *)
| Odiv               (* [rd = r1 / r2] *)
| Omod               (* [rd = r1 % r2] *)

(* 64-bit integer arithmetic: *)
| Omakelong           (* [rd = r1 << 32 | r2] *)
| Osubl              (* [rd = r1 - r2] *)
| Omull              (* [rd = r1 * r2] *)

(* Floating-point arithmetic: *)
| Onegf              (* [rd = - r1] *)
| Oabsf              (* [rd = abs(r1)] *)
| Oaddf              (* [rd = r1 + r2] *)

(* Conversions between int and float: *)
| Ointoffloat         (* [rd = signed_int_of_float64(r1)] *)
| Ofloatofint         (* [rd = float64_of_signed_int(r1)] *)

(* Comparisons: *)
| Ocmp (cond: condition)  (* [rd = 1] if condition holds, [rd = 0] otherwise. *)

(* 81 other cases omitted for brevity ... *)
```

(a) Definition of x86 operations in CompCert.

```
Inductive condition : Type :=
| Ccomp (c: comparison)     (* signed integer comparison *)
| Ccompu (c: comparison)     (* unsigned integer comparison *)
| Ccompl (c: comparison)    (* signed 64-bit integer comparison *)
| Ccompfs (c: comparison)    (* 32-bit floating-point comparison *)
(* 10 other cases omitted for brevity ... *)
```

(b) Definition of x86 comparison in CompCert.

```
Definition eval_operation
   (F V: Type) (genv: Genv.t F V) (sp: val)
   (op: operation) (vl: list val) (m: mem): option val :=
 match op, vl with
 | Omove, v1::nil ⇒ Some v1

 | Ointconst n, nil ⇒ Some (Vint n)

 | Oneg, v1::nil ⇒ Some (Val.neg v1)
 | Omulimm n, v1::nil ⇒ Some (Val.mul v1 (Vint n))
 | Odiv, v1::v2::nil ⇒ Val.divs v1 v2

 | Onegf, v1::nil ⇒ Some(Val.negf v1)

 | Ointoffloat, v1::nil ⇒ Val.intoffloat v1
 | Ofloatofint, v1::nil ⇒ Val.floatofint v1

 (* 90 other cases omitted for brevity ... *)

 | _, _ ⇒ None
 end.
```

Figure 3.9 – Semantics of x86 operations in CompCert.

```
Definition transl_op
  (op: operation) (args: list mreg) (res: mreg) (k: code) : Errors.res code :=
(* ... *)
 | Omul, a1 :: a2 :: nil ⇒
    assertion (mreg_eq a1 res);
    do r ← ireg_of res; do r2 ← ireg_of a2; OK (Pimull_rr r r2 :: k)
(* ... *)
```

Figure 3.10 – Semantics of x86 operations in CompCert.

3.4 CCT-preservation Breach in CompCert

In this section, we detail how the version 3.4 of CompCert does not preserve the constant-time policy. We specifically present details of the implementation that explicitly break the property by introducing a branch. We then present our approach to fix it.

3.4.1 Non-preservation of the Constant-time Policy by the Instruction Selection Pass

In `Cminor`, there are 5 kinds of constants, 33 unary operators and 40 binary operators. All the following intermediate languages, from `CminorSel` to `Mach`, share the same definition of operators, which are architecture dependent. To handle in a uniform way this large variety of shared operators, a generic notion of backend operator that we presented previously 3.3.2.

Among the Cminor operations are int-to-float and float-to-int conversions. Two of them may introduce conditional branches that are not present in the source program, to work around a limitation of the target instruction set architecture: in `x86`, there is no instruction allowing to convert an unsigned 32-bit integer to a float. The workaround is to emulate the generic source operations with several target instructions, sometimes involving new conditional branches. During instruction selection, there are two conversion operations and several comparison operations that generate conditional branches. Figure 3.11a shows in C syntax how the `unsigned int` to `float` conversion is defined in CompCert. It uses the `signed int` to `float` conversion (represented by the function `floatofints`), and introduces a new conditional branch to handle the conversion of an out-of-bound (i.e., greater than or equal to 2^{31}) unsigned integer value. Indeed, the `signed int` to `float` operator can only be applied on an integer whose value is in the interval $[-2^{31}\ ;\ 2^{31})$. The `floatofintu` operation is then implemented as follow in CompCert. If the operand x is in $[0;\ 2^{31})$, then we use the `floatofints` operator to convert it. Otherwise, the operand x is in $[2^{31};\ 2^{32})$. We then subtract 2^{31} from x, apply the `floatofints` operator to $x - 2^{31}$, which is in the correct range, then add back the float value 2^{31} too the result. This a correct implementation despite infamous float computation round errors. In our context, the most important point to remember is that this implementation insert a branch in the program, which breaks the constant-time policy.

In a similar way, the conversion from a `float` to an `unsigned int` operation does not exists in x86. Figure 3.11b shows in C syntax how the `float` to `unsigned int` conversion is defined in CompCert. It uses the `float` to `signed int` conversion (represented by the function `intsoffloat`). The `intsoffloat` can only convert float values in range $[-2^{31}\ ;\ 2^{31})$. Similarly to the previous case, the implementation `intuoffloat` in CompCert uses the `intsoffloat` operation by conditionally performing a shift on the operand when

need. Again, a branch is introduced by this implementations.

Moreover, on 32-bit architectures, comparing 64-bit integers introduces conditional branches. Indeed, in the CompCert memory model, a 64-bit integer is represented by a pair of two 32-bit integers. For efficiency reasons of the generated code, comparing two 64-bit integers consists in first comparing their upper-32 bits, and second in comparing their lower-32 bits only if necessary. Therefore, any comparison between 64-bit integers may introduce new conditional branches, as shown in the example of Figure 3.11c. Given two signed 64-bit integers called `x` and `y`, it shows the two conditional branches generated when compiling the instruction `if (x <= y) goto lthen; else goto lelse;`. The syntax `(xh, xl) = x;` introduces the pair of 32-bit integers representing `x`.

3.5 Modification of the CompCert Compiler

We previously presented three examples of operations that introduce branches during the **Instruction Selection** pass of CompCert. Consequently, it breaks the constant-time policy and our goal is to fix these implementations. We also want to keep the changes we make to CompCert as light as possible, and we want our modified CompCert to produce as efficient code as possible.

Fixing these implementations requires to replace the introduced branches by branchless choices. As discussed in Section 3.2, a branchless choice may be performed using arithmetic or bitwise manipulation, and also using the specific x86 `cmov` instruction. As the latter is significantly more efficient, we chose to use this instruction. Luckily, the `cmov` instruction is already implemented in CompCert, but it only exists in the last language of the compilation chain, `ASM`.

We want to be able to perform branchless choices in the **Instruction Selection** pass, but CompCert currently implements the `cmov` instruction much later in the compilation chain. We then have to choose.

— We can either extend the definitions of all the intermediate languages of the backend to also include a `cmov` instruction. We can then modify all the transformations to compile `cmov` instructions from `CminorSel` to `ASM`. Then, we can use the newly introduced `cmov` instruction in the **Instruction Selection** pass the implement the problematic operation in a branchless way.

— We can also introduce a selection operation, as the one discussed in Section 3.2, in all the languages of the backend, and modify the transformations accordingly. Then, the newly introduced selection operation can be compiled down to `cmov` instructions at **ASMgen** level.

Both these scenarios may seem similar, but we chose the second one for the following reason. Recall that CompCert provides a generic backend operation set that is shared by

```
float floatofintu(unsigned int x) {
  float y = 0x1p31; // 2^31
  if (x < 0x80000000)  // 2^31
    return floatofints(x);
  else
    return y + floatofints(x - 0x80000000);
}
```

(a) The `unsigned int` to `float` conversion.

```
unsigned int intuoffloat(float x) {
  float y = 0x1p31; // 2^31
  if (x < y)
    return intsoffloat(x);
  else
    return 0x80000000 + intsoffloat(x - y);
}
```

(b) The `float` to `unsigned int` conversion.

```
    (xh, xl) = x; (yh, yl) = y;
    if (xh ==_s yh)
      if (xl <=_u yl) goto lthen;
      else            goto lelse;
    else if (xh <_s yh) goto lthen;
       else goto lelse;
```

(c) Comparison $\leq$ for the signed 64-bit integers `x` and `y`, using signed (s) and unsigned (u) operators.

Figure 3.11 – Examples (in `C`-like syntax) of conditional branches introduced by CompCert.

```
Inductive operation : Type :=
(* All the previously defined cases *)
| Oselect: condition → typ → operation.
   (**r [rd = r1] if condition holds, [rd = r2] otherwise. *)
```

Figure 3.12 – New selection operation.

all the languages of the backend, that we presented in Section 3.3.2. Adding one operation to this set is a very light modification, that can modify all the backend of CompCert directly. However, the `cmov` instruction cannot be added to the set of these operations, for the following reason. The semantics used to evaluates operations (Figure 3.9) expects that evaluating an operation will produce a value that only depends on its operands. Consider the following pseudo-code:

```
r ← add [v1;v2]
```

It consists in evaluating the operation `add` on operands `v1` and `v2`, and storing the result in `r`. The operation `Eop add [v1;v2]` produces a value `v1+v2`, which indeed depends on the operands. If we now consider the following hypothetical code:

```
r ← cmov [cond;v]
```

It consists in performing a `cmov` on the value `v`, depending on condition `cond`, and storing the result in `r`. However, we cannot express the value produced by the operation using only the operands. Indeed, the value eventually stored in `r` would be `if cond then v else r`. In other words, the `cmov` instruction performs a side-effect and thus cannot be implemented as a backend operation without heavily modifying the compiler.

On the other hand, a selection operation perfectly fits the requirements of the set of operations of the backend, and we thus chose this option.

3.5.1 Modification of the Set of Operations of the Backend Compiler

Our first modification consists in introducing a operations to the set we presented in Figure 3.9. We extend this definition in Figure 3.12. We introduce the `Osel` operation. It is parameterized with a condition (similar to the `Ocmp` operation) and a type, which corresponds to the type (i.e., integer of floating point values) of the operands. Note that the condition does not belong to the operand list, mainly because of the peculiar way conditions are evaluated on `x86` assembly (i.e., using flags registers). This operation has to be applied to two operands `r1`, and `r2`, and will return `r1` if the condition holds, `r2`

otherwise. We then define a semantics for this operation, that follows this description.

3.5.2 Modification of the Assembly Generation Pass

Our next modification focuses on the assembly generation pass. There, the new selection operation is compiled down to the `cmov` instruction on `x86` architecture.

At `Mach` level, `res` ← `(Oselect c ty) [r1;r2]` operates over two registers `r1` and `r2`, and stores the result in a register `res`. In order to generate efficient code and to apply CompCert optimizations (e.g., register allocation) on the selection operation as well, we defined `Oselect` as a two-address operation (i.e., its first argument `r1` and its result `res` lie in the same location). We then only have to compile operations such as `r1` ← `(Oselect c ty) [r1;r2]` (note that the first operand and the target register are the same). For an integer selection operation (when `ty = int`), this is done directly using only one `cmov` instruction and the negation of the condition `c`. So, the `Mach` operation `r1` ← `(Oselect c ty) [r1;r2]` is compiled down to the `x86` instruction `cmov (negate c) r1 r2`.

Moreover, the `cmov` instruction requires `c` to be a testable condition (usable for a conditional jump), which in our case requires to transform `c` because of the way comparisons proceed on `x86` architecture. More precisely, the `x86` hardware provides a way to handle most comparisons of the source language; we call them testable conditions. Two comparisons (namely equality and inequality between floating point numbers) cannot be handled by the hardware, and require a software workaround. The idea is to split these comparisons into several testable conditions, and to combine them with logical `and` or `or` operators.

The case of equality between floating point numbers introduces a logical `and` in the condition, we then need to compile the operation `r1` ← `(Oselect (c1 && c2) ty) [r1;r2]`. It is done with the following sequence:

```
cmov (negate c1) r1 r2 ; cmov (negate c2) r1 r2
```

Finally, the case of inequality ($\neq$) between floating point numbers introduces a logical `or` in the condition, which is impossible to compile to a sequence of `cmov` as the previous case. The solution we adopted consist in detecting these cases earlier in the compilation chain (namely during the operation selection pass), and introduce the negation of the condition there, while swapping the operands of the selection of operation. We then consider that the problematic case of compiling a logical `or` may never during assembly generation.

Figure 3.13 presents our modifications to the `transl_op` function. Our modifications introduce two auxiliary functions named `transl_sel` and `mk_sel`. First, in the `transl_op` function, we check that the first operand register and the target register are the same,

```
Definition transl_op (op: operation) (args: list mreg)
    (res: mreg) (k: code) : Errors.res code :=
(* All the previously defined cases *)
| Osel c ty, a1 :: a2 :: args ⇒
   assertion (mreg_eq a1 res);
   do r ← ireg_of res; do r2 ← ireg_of a2;
   transl_sel c args r r2 k.
```

```
Definition transl_sel (cond: condition) (args: list mreg)
    (rd r2: ireg) (k: code) : res code :=
  do k1 ← mk_sel (testcond_for_condition cond) rd r2 k;
  transl_cond cond args k1.
```

```
Definition mk_sel (cond: extcond) (rd r2: ireg) (k: code) :=
  match cond with
  | Cond_base c ⇒
     OK (Pcmov (negate_testcond c) rd r2 :: k)
  | Cond_and c1 c2 ⇒
     OK (Pcmov (negate_testcond c1) rd r2 ::
        Pcmov (negate_testcond c2) rd r2 :: k)
  | Cond_or c1 c2 ⇒
     Error (msg "Asmgen.mk_sel") (* should never happen *)
  end.
```

Figure 3.13 – Semantics of x86 operations in CompCert.

as explained previously. We then check that both operand registers are integer registers. Note that we only handle the case of integers, as the `cmov` instruction can only operate over integer registers. Implementing a selection operation that can handle floating point register would require a different workaround, which wasn't necessary in our use case, as we will develop later.

Our second function `transl_sel` generates the code needed to evaluate the condition `cond` of the selection operation. To this end, we use the function `transl_cond`, which already exists in CompCert. Next, we turn the condition into a testable condition with the function `testcond_for_condition`, which also already exists in CompCert). We then study the possible outcomes in the function `mk_sel`. There are three possible outcomes:

— If the condition is directly testable by the hardware (case `Cond_base`), then we can proceed directly by generating one `cmov` instruction, and negating the condition as explained before. This yields the following code: `Pcmov (negate_testcond c) rd r2`.

— If the condition isn't directly testable by the hardware, and and need a logical `and` as a workaround (case `Cond_and`), then we generate a sequence of two `cmov`.

— The last case (for which a logical `or` is required) may never happen as wee prevent it earlier in the compilation chain (see next section). If it happens, the compilation fails and throws an error.

3.5.3 Modification of the Instruction Selection Pass

Using the newly introduced `select` operation, we can now fix the implementations of the operation introducing new branches. Still, these modifications are not straightforward. Consider Figure 3.14a, that present a fix of the original implementation in which we replaced the `if` statement by our `select` operation. This implementation in not correct for two reasons. First, both branches are always executed, which can lead to call the `floatofints` operation with an incorrect argument. For example, if `x = 0`, we will always compute `floatofints(-0x80000000)` which is undefined. The generated code is thus incorrect. Second, this code is incorrect as the `select` operation can only be applied to integer parameter, while here we are trying to apply it to float values.

Section 3.5.2 only describes the compilation of the integer selection operation. Indeed, the `cmov` instruction only operates over integer registers, and there is no `x86` conditional move instruction over floating point registers. Instead of implementing such an operation, we were able to only use the integer selection operation, as shown in Figure 3.14b. In this example, the conversion from `unsigned int` to `float` first computes the values of temporaries `a` and `b` using two integer selection operations, and these values are converted to floating point values. The value of `a` is always in the range of the `float` to the signed `int` operation, and so is the value of `b`, with respect to the condition `x < C`. Figure 3.14d shows our implementation of the comparison of 64-bit integers, which uses the new selection

operation instead of conditional branches.

3.6 Integration to CompCert 3.6

All the changes that we presented in this chapter were discussed with Xavier Leroy. Xavier Leroy integrated them into CompCert version 3.6. They extended this work to the other target architectures: PowerPC, ARM and RISC-V. A selection operation is now available in the backend compiler for all the target architectures.

This selection operation can be used during operation selection to implement some operations, as we detailed in Section 3.5.3. Both original and branchless implementations are available. To enable the branchless implementation, the flag `-Obranchless` must be used.

This work has also been extended by Xavier Leroy to the frontend compiler. It is implemented as a built-in function called `__builtin_sel(a,b,c)`, available in the source language. This function behaves as our `select` operation, i.e., it always evaluates `b` and `c`, and performs a branchless choice between these values, depending on `a`. During operation selection, this built-in function is compiled to the `select` operation. This built-in function may then be used by the programmer to implement constant-time code.

3.7 Experimental Evaluation

All this chapter explained how we modified CompCert in order to make it constant-time preserving. We now present an experimental evaluation of this modified compiler. We focus on the performance of the generated code, compared to other compilers. We measure the execution time on a benchmark of cryptographic code, selected from the literature. We note that our experimental evaluation is primarily used to validate that our approach is reasonable.

We first compare our version of CompCert to the original CompCert (version 3.4), and to `gcc`, with and without optimizations. We test these compilers on a benchmark of common cryptographic programs that were shown to be constant-time in [Alm+16; BPT19]. They include cryptographic primitives such as an implementation of elliptic curve arithmetic operations over Curve25519 [Ber06; Lan15], and TEA [WN94], together with implementations from commonly used cryptographic libraries such as NaCl [BLS12] and mbedTLS [ARM16]. These are C implementations that we experiment with in order to evaluate our compiler, but it should be reminded that if performance is an issue, it is generally better to use hand-optimized assembly code at the cost of portability.

We first measured the execution times (using an Intel i7-8550U CPU 1.8GHz, with 16GB of RAM), which are shown in Figure 3.15. We compiled these programs using the

```
float floatofintu(unsigned int x) {
  float y = 0x1p31; // 2^31
  return select(x < 0x80000000, floatofints(x),
                                y + floatofints(x - 0x80000000));
}
```

(a) Incorrect implementation of the `unsigned int` to `float` conversion.

```
float floatofintu(unsigned int x) {
  unsigned int C = 0x80000000; // 2^31
  int a = select(x < C, 0, -C);
  int b = select(x < C, x, x - C);
  return floatofints(b) - floatofints(a);
}
```

(b) Constant-time implementation of the `unsigned int` to `float` conversion.

```
unsigned int intuoffloat(float x) {
  float y = (float) 0x80000000; // 2^31
  return select(x < y, (int) x, 0x80000000 + (int) (x - y));
}
```

(c) Constant-time implementation of the `float` to `unsigned int` conversion.

```
(xh, xl) = x; (yh, yl) = y;
if (select(xh ==_s yh, xl <=_u yl, xh <_s yh))
  goto lthen;
else
  goto lelse;
```

(d) Comparison $\mathtt{x} \leq \mathtt{y}$

Figure 3.14 – Constant-time implementations (in C-like syntax) of the examples of Figure 3.11.

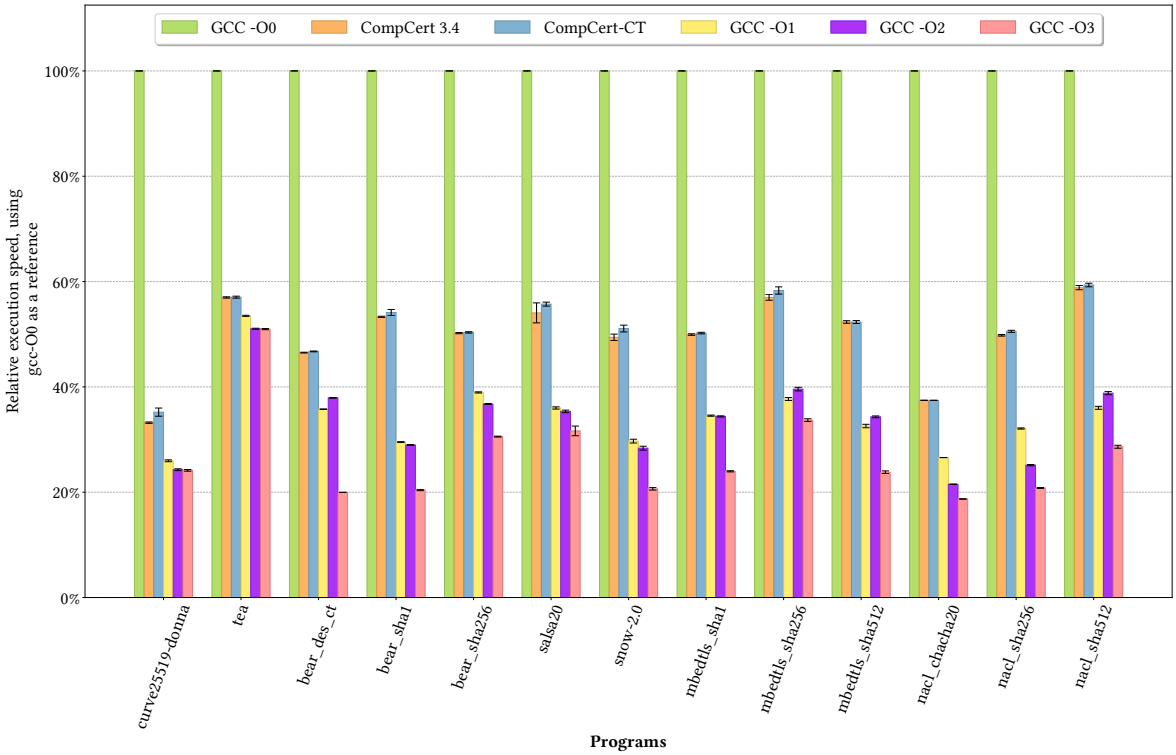


Figure 3.15 – Relative execution times of our benchmark.

We compare the original CompCert, our modified CompCert, and `gcc` from `-O0` to `-O3`. We normalized the measured execution times with the execution times of `gcc -O0`. The error bars represent the 99% confidence intervals of our measurements.

original CompCert 3.4, our modified version of CompCert, and `gcc` at different levels of optimization. As all programs have a very short execution time, we executed them from 10^6 to 10^9 times depending on the program, and measure the average execution time. We obtain shorter execution times than `gcc` without optimization. This is a promising result, as cautious users who are wary of too aggressive compiler optimizations breaking constant-time security would use it to compile cryptographic implementations. Yet, `gcc -O1` and further optimizations remain more efficient. Moreover, we also noticed that our modified CompCert is as efficient as the original CompCert. On average, programs compiled with our modified CompCert are 1.45% slower than programs compiled with the original CompCert.

We further compared two representative cryptographic primitives of our benchmark (namely NaCl Chacha20 stream cipher and Poly1305 authenticator) against heavily optimized implementations using handwritten assembly or AVX instructions (that are not supported by CompCert) from the OpenSSL [Ope19] and HACL* [Zin+17] libraries, and implementations written in Jasmin [Alm+17; Alm+19]. For large message sizes, Jasmin is 20 times faster that CompCert when it runs on Chacha 20, and 5 times faster when it runs on Poly1305. These differences are reduced by a half when comparing HACL* and CompCert. These comparisons show that there is still room for progress in CompCert by adding support for extended instruction sets such as AVX for instance.

3.8 Conclusion

In this chapter, we first presented some minimalist examples breaking the constant-time policy on real-life compilers. Specifically, we presented two simple programs that are secure at source level with respect to the constant-time policy, but are compiled into unsecure code, with either GCC or Clang.

We then highlighted that these problems also exist on the version 3.4 of the CompCert compiler. Our goal was then to tackle this issue by modifying the compilation chain, so that the constant-time policy is strictly preserved.

We then presented a detailed background about the CompCert compiler, with specific details on its architecture, including the intermediate languages, the different passes, and the set of operations of the compiler. After that, we presented the detail of our modification, that impacted two of the passes of the compiler.

Last, we discuss how the changes presented in this chapter were integrated in CompCert (in version 3.6), and also presented an experimental evaluation of our modifications, showing that the impact on the performance of the generated code is very low.

Chapter 4

Constant-Resource Policy

Timing side-channel attacks consider an attacker capable of deducing the value of sensitive data by measuring the execution time of a program. As a consequence, we say that a program is secure against timing side-channel attacks if its timing behavior does not depend on secret values. Such a program is said to be *timing-secure*. This definition is natural, but enforcing this property is a complex task, as the execution time of a program depends, among other things, on architectural features of the machine executing it.

The Cryptographic Constant-Time policy (CCT), that we studied in the previous chapter, is commonly used as a protection, and offers strong security properties against timing side-channel attacks. By ensuring that (a) the execution flow of a program is secret-independent and (b) every instruction whose execution time depends on the operands is secret independent, the CCT policy ensures the independence between the secret values of a program and its execution time. As such, any CCT program is secure w.r.t. timing side-channel attacks. In other words, any CCT-secure program is timing-secure. Many cryptographic implementations have been modified in order to respect the CCT policy.

However, the CCT policy presents a few drawbacks. On the one hand, complying to the CCT policy is a complex task, and a CCT implementation usually has lower performance than a natural implementation. On the other hand, the CCT policy may be considered to be too restrictive, as program may be timing-secure but CCT-secure. Timing-security may be achieved by using time-balanced secret-dependent branches, i.e., secret branches that have identical execution time. Achieving such balance is challenging, and heavily depends on architectural features.

Yet, it is possible to find some implementations of common cryptographic primitives that do not strictly comply with the CCT policy, and present the pattern mentioned above. We illustrate this with an example inspired from Amazon's implementation of TLS [AP16]. We consider the following program:

```
repeat(n) { update(); }
```

In this example, we consider a secret value n, bounded between 0 and 32. We also consider a function `update`, whose execution time u is supposed to be known and constant. We also assume loop control instructions to have negligible execution time compared to u.

The program above performs n calls to the function `update`. The measured execution time t of this program is likely to be close to $n \times u$, and then $\frac{t}{u}$ is going to be close to the secret value n. In other words, the secret value n is leaked by the execution time. The CCT policy would simply forbid this program, as a secret-dependent branch is performed here. Instead, in [AP16], the authors suggest to repair it as follows:

```
repeat(n) { update(); }
repeat(32 - n) { dummyUpdate(); }
```

In this repaired program, we perform in addition $32 - n$ calls to a new function `dummyUpdate`, and assume that the function `dummyUpdate` does nothing and is precisely crafted in order to have an execution time as close as possible as `update` (i.e., u). Therefore, executing this whole program would have an approximate execution time of $n \times u + (32 - n) \times u = 32u$. In other words, the execution time is now constant and independent from the secret value n. The program has been repaired by introducing *padding*, i.e., code that does nothing but takes time to execute.

In this chapter, we focus on what we call the Constant-Resource (CR) policy, which captures the relaxed CCT policy used by some practitioners, i.e., allows secret-dependent balanced branchings, as illustrated in the previous example. We then focus on the preservation of this policy during compilation.

The CR policy defines a notion of resource consumed during the execution of a program. A resource can be a counter measuring the number of arithmetic operations, memory accesses or function calls. A more precise resource model can take branch prediction and cache into account to model execution time on a given architecture. The CR policy states that an attacker capable of measuring the resources consumed during the execution of a program cannot deduce any information on the secrets of the program.

```
if(secret) {x+=2;} else {y+=3;}
```

(a) A program with balanced branches

$\boxed{\texttt{if(secret) \{}p_1\texttt{\} else \{}p_2\texttt{\}}}\texttt{;}\ p_3$

(b) A program with an atomic annotation

Figure 4.1 – Examples of CR-secure programs

The code snippet presented in Figure 4.1a consists of a branching on a secret value, and two branches performing the same kind of operations (i.e., incrementing a variable by a constant value). In a resource model counting the number of arithmetic operations, this snippet is considered CR-secure, as the resource consumption is constant and does not depend on the secret value of the snippet. Unlike the CCT policy, the CR policy tolerates

a branching depending on a secret value, as long as the branches are balanced in terms of costs. Because leakages are not constrained in these branches, the CR policy does not satisfy the non-cancellation property.

In this chapter, we study the preservation by compilation of the CR policy and present a proof methodology to prove that a transformation preserves the CR policy. Compiler optimizations may easily break the CR policy. Indeed, as a CR-secure program may contain balanced branches, any optimization that reduces the resource consumption in one of the branches would directly break the balance.

To solve this issue, a first solution is to use information-flow typing to guide the transformation. Type systems can detect high branches and forbid or restrict optimization inside them. For example, in [Aga00] a standard type system [VIS96] is used to *repair* a typable program that may contain unbalanced high branches. This is an elegant approach but in our work we want to avoid the use of an information-flow type-system inside the compilation chain. Modern compilers perform their optimizations at a low-level program representation and running a taint analysis at this level is likely to conservatively declare all the contents of the memory as secret-dependent.

Instead, our methodology relies on a new syntactic annotation that we call an *atomic* annotation. It restricts the compiler, without asking it to perform a taint analysis at every level of the compilation chain. Figure 4.1b presents an example of *atomic* annotation, that we represent as a box notation around a part of the program. The *atomic* constructs can be inserted at source level using a source type system or any program logic that detects high branches. We introduce a more flexible policy, called CR$^{\#}$, which combines elements from the CCT [Alm+16] and CR [Ngo+17] policies (i.e., leakages and costs). CR$^{\#}$ leakages track costs as well as the CCT boolean leakages of some branchings. The *atomic* construct is used to decide whether or not the CR$^{\#}$ policy tracks a branching statement.

4.1 An Introduction to CR-security

In this section, we introduce the CR policy, first informally through an example, then with formal definition for our language $\mathscr{L}$. We also focus on the problem of secure compilation of CR programs, i.e., how to ensure that the CR policy is preserved during a program transformation. We then motivate the need for the CR$^{\#}$ policy.

4.1.1 Example: Common Subexpression Elimination

This section first gives examples of CR-secure programs. Then, it explains through the example of Common Subexpression Elimination (CSE) how to adapt a transformation to make it CR-preserving.

```
if (cond) {
 x = a*b;
 y = (a*b)+c+d;
} else {
 x = a+b;
 y = (a+b)*c*d;
}
```

(a) A balanced program with common subexpressions `a*b` and `a+b`

```
if (cond) {
 x = a*b;
 y = x+c+d;
} else {
 x = a+b;
 y = x*c*d;
}
```

(b) The unbalanced optimized program (with CSE)

```
if (cond) {
 δ(T1);
 x = a*b;
 y = x+c+d;
} else {
 δ(T2);
 x = a+b;
 y = x*c*d;
}
```

(c) The padded optimized program

```
if (cond) {
 δ(T1 − T);
 x = a*b;
 y = x+c+d;
} else {
 δ(T2 − T);
 x = a+b;
 y = x*c*d;
}
```

(d) The padded optimized program with minimal padding

Figure 4.2 – Example of branching programs
where $T_1 = K^{mult} + K^{var}$, $T_2 = K^{add} + K^{var}$ and $T = \min(T_1, T_2)$.

Example of CR-secure Programs

Figure 4.2 presents some code snippets written in `C` syntax. In the first one, if the condition `cond` is secret, then this snippet is considered unsecure by the CCT policy. In general, branchings on secrets are unsecure because an attacker able to measure their execution time could determine which branch was executed, and thus the secret condition. However, because both branches consume the same amount of resources (i.e., six accesses to variables, two additions, two multiplications, and two assignments), we better consider that these branches are indistinguishable from the perspective of such an attacker. This program is then an example of CR-secure program.

The code snippet in Figure 4.2a presents some redundant computations, and it is a good candidate for the CSE optimization. Figure 4.2b shows the optimized code snippet, which is not CR-secure. Indeed, both branches perform five accesses to variables and two assignments, but the `then` branch performs two additions and one multiplication, while the `else` branch performs one addition and two multiplications. Hence, these branches are no longer balanced. This illustrates how a simple transformation can break the CR policy.

A CSE optimization that Preserves the CR Policy

Preserving the CR policy requires to carefully keep track of all the branches of the program. Any optimization performed only in one branch could unbalance the whole program. Our approach consists in preserving the balance by introducing a minimal amount of padding in every unbalanced branch. We illustrate it with the example of the CSE

optimization.

We add two steps to the CSE optimization to make it CR-preserving. First, our new CSE adds padding to balance the consumption of resources between the branches, using a new padding instruction called $\boldsymbol{\delta}$. It is parameterized by an integer n and executing $\boldsymbol{\delta}(n)$ consumes n resources. Second, our CSE performs a pass called $\mathcal{M}$ that minimises the padding by factorising and removing as many $\boldsymbol{\delta}$ instructions as possible, as long as the CR policy is preserved. More precisely, any modification of the resource consumption stemming from CSE is compensated by an adequate $\boldsymbol{\delta}$ instruction.

In the `then` branch of Figure 4.2c, our CSE factorises a redundant evaluation and modifies the resource consumption: the optimized program has one less multiplication and one less variable access. So, our CSE compensates these changes by adding the instruction $\boldsymbol{\delta}(T_1)$ in the `then` branch, with $T_1 = K^{mult} + K^{var}$, where the K^{mult} and K^{var} constants are statically computed and represent the cost of respectively a multiplication and a variable access. The added $\boldsymbol{\delta}$ instruction consumes the same amount of resources spared by CSE, hence preserving the resource consumption of the whole program. Similarly, the padding instruction $\boldsymbol{\delta}(T_2)$ is added in the `else` branch, with $T_2 = K^{add} + K^{var}$. Figure 4.2d shows the final code snippet, where the padding of both branches is reduced by $T = \min(T_1, T_2)$. This last step minimises the padding while keeping both branches balanced.

Our modified version of the CSE optimization introduces padding to preserve the balance of costs between branches. Then, it minimises the inserted padding as much as possible, while preserving the CR policy. However, this behaviour may not always be desired, as it makes the output program less efficient in terms of consumed resources. Indeed, some branches could be secret dependent and balanced, while some other branches could be non secret-dependent. The former branches would need a careful and restrained optimization, similar to the example above. However, the latter branches could benefit from a more aggressive optimization, without requiring any padding. To distinguish between both cases, we introduce a syntactic annotation, called *atomic*, which delimits the areas of the program to be carefully optimized. The padding insertion and minimisation passes are then only performed in these areas.

The proof that our modified CSE preserves the CR policy consists of the individual proofs of each of its steps. This proof effort led us to define a stronger security policy, that we call CR$^{\#}$, and whose definition depends on the atomic annotations. This CR$^{\#}$ policy is discussed in the next section. Some proofs are trickier than others, and we define in this chapter two other policies that are stronger than the CR$^{\#}$ policy but facilitate our proofs. In particular, we decompose $\mathcal{M}$ into two steps, and each of them is proved using a different policy. We also prove that any of these policies implies the CR$^{\#}$ policy. Interestingly, these policies are not peculiar to CSE and could be reused to prove other optimizations.

4.1.2 Motivation of the CR$^{\#}$ policy

This section first motivates the choice of our security policy. Our work is based on the CR security property presented in [Ngo+17], which we describe first. We then present a stronger property, called CR$^{\#}$, and motivate this definition. CR$^{\#}$ is the security property we will focus on during the rest of this chapter.

The CR Policy

In [Ngo+17], the authors present the CR policy. It captures the fact that a given notion of resources consumed during the execution of a program does not reveal any information on the secret values of this program. More formally, let us first consider a language $\mathcal{L}$, and its big-step semantics judgement $\langle p,\, \sigma\rangle \Downarrow (\sigma', q)$ instrumented to observe the resource consumption of an execution. We read it as follows: the execution of a program $p \in \mathcal{L}$ from an initial state σ to a final state σ' consumes q resources, where q is an integer. States map variable identifiers to values, and every variable is marked as either secret or public.

Next, we consider a notion of indistinguishability between semantic states. Two states σ_1 and σ_2 are indistinguishable, written as $\sigma_1 \sim \sigma_2$, if every public variable has the same value in both states. Then, the CR policy is defined as follows. A program is CR if any pair of executions whose initial states only differ on secret values consume the same amount of resources. This captures the idea that resource consumption does not reveal any information on the secrets of a CR program.

Definition 4.1: CR-security.

Let p be a program, and states σ_1 and σ_2 such that $\sigma_1 \sim \sigma_2$. Suppose that we have two executions of p: $\langle p,\, \sigma_1\rangle \Downarrow (\sigma'_1, q_1)$ and $\langle p,\, \sigma_2\rangle \Downarrow (\sigma'_2, q_2)$. The program p is CR-secure (written as $CR(p)$) when $q_1 = q_2$.

Secure Compilation of CR Programs

Our goal is to implement program transformations that preserve the CR policy, then prove that these transformations always preserve the policy. Formally, we say that a transformation $\mathcal{T}$ is CR-preserving if for any program p, we have $CR(p) \implies CR(\mathcal{T}(p))$.

A first possibility is to use a type system, as in [Ngo+17], where the type system is designed so that any well-typed program p, denoted as $\vdash p$, is CR-preserving: $\vdash p \implies CR(p)$. We could then prove that $\mathcal{T}$ preserves such a type system. Formally, we would prove the following property: $\vdash p \implies \vdash \mathcal{T}(p)$, stating that the type system enforces the CR policy on both source and transformed programs. This approach would work in the

context of a simple type system and a simple language. However, we argue that it would not scale to a more realistic compiler, such as CompCert or LLVM, as type-preserving compilers typically do not scale to realistic languages. As far as we know, no realistic compiler includes a type system to verify the preservation of non-interference properties. We refer to chapter 2 for more explanations. Another drawback of this approach is that the compiler must explicitly know the security level (i.e., secret or public) of every variable.

Our methodology to prove that a transformation is CR-preserving does not rely on a type system, but rather on an extended language and its instrumented semantics. Firstly, we extend the language $\mathcal{L}$ with a syntactic annotation, that we call an *atomic* annotation. Secondly, we extend the semantics of $\mathcal{L}$, by instrumenting it with leakages that track the branchings encountered during the execution. The information leaked is a partial control-flow of the program, represented by a list of booleans values, that contains some of the guards evaluated during the execution. The choice of leaked control-flow depends on the atomic annotations, as secret branching conditions only appear inside of atomic annotations. Last, we extend the CR policy to facilitate our proofs. We introduce a stronger policy, called $\mathrm{CR}^{\#}$, that is defined with respect to this newly introduced leakage.

We use the $\mathrm{CR}^{\#}$ policy in the following way. Firstly, we annotate any program p into $p^{\#}$, so that $CR(p) \implies \mathrm{CR}^{\#}(p^{\#})$. This will be discussed in Section 4.3.1. Secondly, we prove that the transformation $\mathcal{T}$ we are focusing on preserves the $\mathrm{CR}^{\#}$ policy. Formally, for any program p, $\mathrm{CR}^{\#}(p) \implies \mathrm{CR}^{\#}(\mathcal{T}(p))$. Last, as $\mathrm{CR}^{\#}$ is stronger than CR, we directly have for any program p, $\mathrm{CR}^{\#}(p) \implies CR(p)$.

In the following section, we will give precise definitions of the CCT and CR policies, for an example language, and compare these policies. Then, in section Section 4.3, we will give formal definitions of the $\mathrm{CR}^{\#}$ policy and of the atomic annotation.

4.2 Comparison between the CCT and CR Policies

In this section, we extend the language $\mathscr{L}$ presented in chapter 2, and extend its semantics. This extended semantics emits a new kind of leakage, that captures the resource consumption. We then use this leakage to formally define both the CCT and the CR policies. Last, we compare these policies, and prove that any CCT-secure program is also CR-secure.

4.2.1 Language Definition

We introduce below the new syntax of $\mathscr{L}$. The only difference is the introduction of a new padding instruction $\boldsymbol{\delta}$, parameterized by a quantity of consumed resources (i.e., the execution of $\boldsymbol{\delta}(n)$ where n is a constant consumes n resources).

Definition 4.2: Extended syntax of $\mathscr{L}$.

$$
\begin{aligned}
\langle exp \rangle &::= \langle int \rangle \mid \langle ident \rangle \mid \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{] } \mid \langle exp \rangle \diamond \langle exp \rangle \mid \\
\langle stmt \rangle &::= \texttt{skip} \\
&\mid \langle ident \rangle \texttt{ := } \langle exp \rangle \\
&\mid \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{] := } \langle exp \rangle \\
&\mid \langle stmt \rangle \texttt{ ; } \langle stmt \rangle \\
&\mid \texttt{if(} \langle exp \rangle \texttt{) \{ } \langle stmt \rangle \texttt{ \} else \{ } \langle stmt \rangle \texttt{ \}} \\
&\mid \texttt{while(} \langle exp \rangle \texttt{) \{ } \langle stmt \rangle \texttt{ \}}
\end{aligned}
$$

Next, we extend the notion of leakage introduced in subsection 2.5.2. We add a new event $e_R(q)$, that captures the consumption of $q \in \mathbb{Z}$ resources. The new definition of the set of event $\mathcal{E}$ is the following:

Definition 4.3: Set of event $\mathcal{E}$.

$$\langle event \rangle \ ::= \ e_B(b) \mid e_M(id, n) \mid e_R(q)$$

$e_B(b)$ is an event emitted at a branch, where b is a boolean value. b is the truth value to which to condition has evaluated to. $e_M(id, n)$ is an event emitted at a memory access, where id is an array identifier, and n is the index of the value accessed in this array. $e_R(q)$ is an event emitted by any instruction that consumes q resources.

We now present the new instrumented semantics of $\mathscr{L}$. It is identical to the semantics presented in **Definition 2.18** and **Definition 2.19**. The only difference is the emission of resource consumption events by several language constructs, as we as a semantics of the new $\boldsymbol{\delta}$ operator. The semantics is parameterized by a set of constants denoted by $K^{\cdots}$ and representing the unitary costs of basic syntactic constructs. For example, K^{asn} is the cost of an assignment. The following definitions describe this semantics, both of expressions and statements.

Definition 4.4: Evaluation of expression $\langle e, \sigma \rangle \Downarrow (v, \ell)$.

VAR

$$\frac{\sigma[x] = v}{\langle x,\, \sigma\rangle \Downarrow (v, e_R(K^{var}))}$$

ARRAY

$$\frac{\sigma[x] = [v_1; ...; v_n; ...v_m] \qquad \langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad 1 \leq n \leq m}{\langle x\,\texttt{[}e\texttt{]},\, \sigma\rangle \Downarrow (v_n, (\ell \cdot e_M(x, n) \cdot e_R(K^{array})))}$$

CONST

$$\frac{}{\langle n,\, \sigma\rangle \Downarrow (n, e_R(K^{const}))}$$

OP_BIN

$$\frac{\langle e_1,\, \sigma\rangle \Downarrow (v_1, \ell_1) \qquad \langle e_2,\, \sigma\rangle \Downarrow (v_2, \ell_2)}{\langle e_1 \diamond e_2,\, \sigma\rangle \Downarrow (v_1 \diamond v_2, (\ell_1 \cdot \ell_2))}$$

Definition 4.5: Evaluation of statement $\langle s,\, \sigma\rangle \Downarrow (\sigma', \ell)$.

SKIP

$$\frac{}{\langle \texttt{skip},\, \sigma\rangle \Downarrow (\sigma, \epsilon)}$$

PADDING

$$\frac{}{\langle \boldsymbol{\delta}\texttt{(}n\texttt{)},\, \sigma\rangle \Downarrow (\sigma, e_R(n))}$$

ASSIGN

$$\frac{\langle e,\, \sigma\rangle \Downarrow (v, \ell)}{\langle id\texttt{:=}e,\, \sigma\rangle \Downarrow (\sigma[id \leftarrow v], (e_R(K^{asn}) \cdot \ell))}$$

ASSIGN_ARRAY

$$\frac{\begin{array}{c}\sigma[x] = [v_1; ...; v_n; ...v_m] \\ \langle e_1,\, \sigma\rangle \Downarrow (n, \ell_1) \qquad \langle e_2,\, \sigma\rangle \Downarrow (v', \ell_2) \qquad 1 \leq n \leq m \qquad v' \in \mathbb{Z}\end{array}}{\langle a\,\texttt{[}e_1\texttt{]}\texttt{:=}e_2,\, \sigma\rangle \Downarrow (\sigma[a \leftarrow [v_1; \ldots; v_{n-1}; v'; v_{n+1}; \ldots; v_m]], (\ell_1 \cdot \ell_2 \cdot e_M(x, n) \cdot e_R(K^{asn_array}))}$$

SEQ

$$\frac{\langle p_1,\, \sigma\rangle \Downarrow (\sigma', \ell_1) \qquad \langle p_2,\, \sigma'\rangle \Downarrow (\sigma'', \ell_2)}{\langle (p_1\texttt{;}\ \ p_2),\, \sigma\rangle \Downarrow (\sigma'', (\ell_1 \cdot \ell_2))}$$

IF_TRUE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n \neq 0 \qquad \langle p_1,\, \sigma\rangle \Downarrow (\sigma', \ell_1)}{\langle \texttt{if(}e\texttt{)}\ \texttt{\{}p_1\texttt{\}}\ \texttt{else}\ \texttt{\{}p_2\texttt{\}},\, \sigma\rangle \Downarrow (\sigma', (\ell \cdot e_B(\text{true}) \cdot \ell_1))}$$

IF_FALSE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n = 0 \qquad \langle p_2,\, \sigma\rangle \Downarrow (\sigma', \ell_2)}{\langle \texttt{if(}e\texttt{)}\ \texttt{\{}p_1\texttt{\}}\ \texttt{else}\ \texttt{\{}p_2\texttt{\}},\, \sigma\rangle \Downarrow (\sigma', (\ell \cdot e_B(\text{false}) \cdot \ell_2))}$$

WHILE_TRUE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n \neq 0 \qquad \langle p,\, \sigma\rangle \Downarrow (\sigma', \ell') \qquad \langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma'\rangle \Downarrow (\sigma'', \ell'')}{\langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma\rangle \Downarrow (\sigma'', (\ell \cdot e_B(\text{true}) \cdot \ell' \cdot \ell''))}$$

WHILE_FALSE

$$\frac{\langle e,\, \sigma\rangle \Downarrow (n, \ell) \qquad n = 0}{\langle \texttt{while(}e\texttt{)}\ \texttt{\{}p\texttt{\}},\, \sigma\rangle \Downarrow (\sigma, (\ell \cdot e_B(\text{false})))}$$

4.2.2 Semantic Properties of $\mathcal{L}$

In this section, we state two semantic properties of $\mathcal{L}$ that will be used in the rest of this chapter. First, the semantics is deterministic, both for the output state and the emitted leakage.

Lemma 4.1: Determinism.

Let p be a program and σ an initial state. If we have two executions $\langle p, \sigma\rangle \Downarrow (\sigma_1, \ell_1)$ and $\langle p, \sigma\rangle \Downarrow (\sigma_2, \ell_2)$, we then have $\sigma_1 = \sigma_2$ and $\ell_1 = \ell_2$.

The second property focuses on the non-cancellation of the leakage emitted by a program. We assume to have two arbitrary executions of a program p, respectively emitting leakages ℓ_1 and ℓ_2. The lemma states that for any arbitrary leakages ℓ'_1 and ℓ'_2, if we have $\ell_1 \cdot \ell'_1 = \ell_2\ell'_2$.

Lemma 4.2: Non-cancellation.

If we have the following hypotheses:

(E_1) $\langle s, \sigma_1\rangle \Downarrow (\sigma'_1, \ell_1)$
(E_2) $\langle s, \sigma_2\rangle \Downarrow (\sigma'_2, \ell_2)$
(EQ) $l_1 \cdot l'_1 = l_2 \cdot l'_2$,

then we have $l_1 = l_2$ and $l'_1 = l'_2$.

Proof of Lemma 4.2.

We prove this lemma by reasoning by induction on the execution (E_1). We focus on some representative cases; the other cases use similar reasoning.

— Case SEQ. s is then a sequence of statements, say $s = (s_1\texttt{;}\ s_2)$. We apply the induction hypothesis on statements s_1 and $(s_2\texttt{;}\ p)$ to find that both executions of s_1 emit the same leakage, and so do both executions of $(s_1\texttt{;}\ s_2)$. We then apply the induction hypothesis on s_2 to conclude.

— Case IF. s is then an if-branch, say $s = \texttt{if(}e\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}$. From the hypothesis (EQ), we can deduce that e evaluates to the same truth value in both executions of s. We consider that e evaluates to *true*. We then necessarily have two executions of s_1. We conclude by applying the induction hypothesis on s_1.

□

4.2.3 Definition of the CCT and CR Policies

Using this new instrumented leakage, we can define both the CCT and the CR policies, as instances of ONI. First, similarly to the definition presented in subsection 2.5.2, CCT

is now defined as ONI_{π_1}, where π_1 is the projection function that ignores the resource consumption events.

Example 8: Example of leakage projection with π_1.

$$\pi_1(e_B(\text{true}) \cdot e_R(2) \cdot e_M(x, 2) \cdot e_R(4)) = e_B(\text{true}) \cdot e_M(x, 2)$$

More generally, $e_R(n)$ events are removed, and the other events are not modified.

This new definition is strictly equivalent to the definition presented in subsection 2.5.2.

Next, we define the CR policy as another instance of ONI. This policy focuses on the global cost of an execution, rather than on the list of individual costs. We define the global cost of an execution as the sum of the individual costs, described by events $e_R(n)$ in the leakage. Therefore, we define CR as ONI_{π_2}, where π_2 is the projection function that sums all the resource consumption of all the $e_R(n)$ events, and ignores any other event.

Example 9: Example of leakage projection with π_2.

$$\pi_2(e_B(\text{true}) \cdot e_R(2) \cdot e_M(x, 2) \cdot e_R(4)) = 6$$

More generally, $e_R(n)$ events are summed, and the other events are ignored.

Note that these definitions are close, the only difference being that they both selectively focus on one part of the leakage, through the projection functions. However, the CCT policy is more restrictive than the CR policy. In fact, we can prove that any CCT program is also CR. This is a consequence of the following result.

Lemma 4.3: CCT leakage implies CR leakage.

Let p be a statement. We assume to have the following 2 executions:

(E_1) $\langle p, \sigma_1 \rangle \Downarrow (\sigma'_1, \ell_1)$
(E_2) $\langle p, \sigma_2 \rangle \Downarrow (\sigma'_2, \ell_2)$

If we have $\pi_1(\ell_1) = \pi_1(\ell_2)$, then $\pi_2(\ell_1) = \pi_2(\ell_2)$.

This lemma states that if a pair of arbitrary executions of a program p emit the same CCT leakage (i.e., list of memory accesses and branchings), then they consume the same amount of resources. We provide a proof of this lemma below:

Proof of Lemma 4.3.

The proof of this lemma heavily relies on the non-cancellation **Lemma 4.2**. We prove this lemma by reasoning by induction on any execution, for instance (E_1). We focus on some representative cases; the other cases use similar reasoning.

— Case SEQ. p is then a sequence of statements, say $p = (p_1\texttt{;}\ p_2)$. We then have two executions of p_1 and two executions of p_2. We use the non-cancellation lemma to conclude that both executions of p_1 produce the same leakage, and so do both executions of p_2. We can the conclude by applying the induction hypothesis on both p_1 and p_2.

— Case IF. p is then an if-branch, say $p = \texttt{if}(e)\ \{p_1\}\ \texttt{else}\ \{p_2\}$. As both executions have the same boolean leakage, they necessarily take the same path, i.e., e evaluates to the same truth value in both executions. We can then conclude by applying the induction hypothesis on either p_1 or p_2.

□

With the lemma above, we can directly prove the following main theorem.

Theorem 4.4: CCT implies CR.

Any CCT-secure program is also CR-secure.

Proof of Theorem 4.4.

Let p be a CCT program. Let E_1 and E_2 be two indistinguishable executions of p, emitting respective leakages ℓ_1 and ℓ_2. As p is CCT, we have $\pi_1(\ell_1) = \pi_1(\ell_2)$. Using **Lemma 4.3**, we can deduce that the resource consumption are also equal, i.e., $\pi_2(\ell_1) = \pi_2(\ell_2)$, and thus p is also CR. □

This concludes this section whose goal was to compare the CCT and the CR policies, and we proved that any CCT-secure program is also CR-secure. In the next section, we introduce our CR$^\#$ policy, as introduced in 4.1.2. This new policy is defined as a mix between the CCT and the CR policies.

4.3 Our Approach using a Syntactic Annotation

4.3.1 CR$^\#$, a more Flexible Policy

In this section, we extend our language $\mathcal{L}$ into $\mathcal{L}^\#$, in order to introduce a new syntactic annotation called *atomic*. Then, we discuss the relation between the CCT, CR and CR$^\#$ policies.

Semantics of the $\mathcal{L}^{\#}$ Language

We extend the language $\mathcal{L}$ into $\mathcal{L}^{\#}$, which contains one new feature: an annotation called *atomic*. The syntax of $\mathcal{L}^{\#}$ is given in the following definition, and the annotation is denoted using a box notation: $\boxed{p}$ is the atomic version of p.

Definition 4.6: Extended syntax of $\mathscr{L}$.

$$
\begin{array}{rcl}
\langle exp \rangle & ::= & \langle int \rangle \mid \langle ident \rangle \mid \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{]} \mid \langle exp \rangle \diamond \langle exp \rangle \mid \\
\langle stmt \rangle & ::= & \texttt{skip} \\
& \mid & \langle ident \rangle \texttt{ := } \langle exp \rangle \\
& \mid & \langle ident \rangle \texttt{ [} \langle exp \rangle \texttt{] := } \langle exp \rangle \\
& \mid & \langle stmt \rangle \texttt{ ; } \langle stmt \rangle \\
& \mid & \texttt{if(} \langle exp \rangle \texttt{) \{} \langle stmt \rangle \texttt{ \} else \{} \langle stmt \rangle \texttt{ \}} \\
& \mid & \texttt{while(} \langle exp \rangle \texttt{) \{} \langle stmt \rangle \texttt{ \}} \\
& \mid & \boxed{\langle stmt \rangle}
\end{array}
$$

We also extend the notion of leakage by introducing a new event to the set of event $\mathcal{E}$. We call this event an *atomic* event, and denote it with a similar box notation around a leakage ℓ: $\boxed{\ell}$ is the atomic version of ℓ.

Example 10: Example of leakage with atomic event.

For example, the following sequence of events is a valid leakage:

$$e_B(\text{true}) \cdot e_R(2) \cdot \boxed{e_B(\text{false}) \cdot e_R(4)} \cdot e_M(x, 2)$$

It consists 4 events. The 3rd event is an atomic event, that consists of a leakage $e_B(\text{false}) \cdot e_R(4)$.

We give below the extended definition of the set of events $\mathcal{E}$:

Definition 4.7: Set of event $\mathcal{E}$.

$$\langle event \rangle \ ::= \ e_B(b) \mid e_M(id, n) \mid e_R(q) \mid \boxed{\ell}$$

$e_B(b)$ is an event emitted at a branch, where b is a boolean value. b is the truth value to which to condition has evaluated to. $e_M(id, n)$ is an event emitted at a memory access, where id is an array identifier, and n is the index of the value accessed in this array. $e_R(q)$ is an event emitted by any instruction that consumes q resources. $\boxed{\ell}$ is an event emitted by an atomic annotation ; it contains a leakage ℓ.

The semantics of $\mathcal{L}^{\#}$ is almost identical to the semantics of $\mathcal{L}$. We only add the semantic of the atomic annotation, that we give in the following definition:

Definition 4.8: Semantics of an *atomic* annotation.

$$\textsc{Atomic}\quad \frac{\langle p,\, \sigma\rangle \Downarrow (\sigma', \ell)}{\langle \boxed{p},\, \sigma\rangle \Downarrow (\sigma', \boxed{\ell})}$$

The execution of $\boxed{p}$ executes p, and marks the whole leakage ℓ produced by p as atomic (i.e., emits $\boxed{\ell}$). Intuitively, the goal of the atomic annotation is to allow secret-dependent branches inside of these annotations, as long as the branches as balanced with respect to the resource consumption. Outside of atomic annotations, secret-dependent branches are never allowed. The CR$^{\#}$ policy exactly captures this behavior, and is defined as an instance of ONI, using an adequate projection function that selectively erase some parts of the leakage, using the atomic events. We define it precisely in the following section.

Note that nested atomic annotations are allowed, both in the syntax of the language and in the leakage. Especially, this gives our notion of leakage a surprising structure of tree. However, nesting atomics has no effect, either on the execution of a program nor on the CR$^{\#}$ policy. We will only give example that do not contain nested atomics.

Semantic Definition of the CR$^{\#}$ Policy

We now define the CR$^{\#}$ policy. It is another instance of ONI and is defined as follows.

Definition 4.9: CRplus-security.

We define CR$^{\#}$ as ONI$_{\pi_3}$, with π_3 a projection function. $\pi_3(\ell)$ returns a pair (ℓ', q) where ℓ' is a leakage and $q \in \mathbb{Z}$ is an amount of resources, such that:

— $q = \pi_2(\ell)$, i.e., q is the sum of all the $e_R(n)$ events encountered during an execution. This applies recursively to atomic events.

— ℓ' is the leakage ℓ in which we removed every $e_R(n)$ and atomic events. In other words, we only keep the CCT leakage, and ignore anything happening inside an atomic event.

We give below an example projected leakage using the π_3 projection.

Example 11: Example of leakage projection with π_3.

$$\pi_3(e_B(\text{true}) \cdot e_R(2) \cdot \boxed{e_B(\text{false}) \cdot e_R(4)} \cdot e_M(x, 2))$$
$$= (e_B(\text{true}) \cdot e_M(x, 2),\ 6)$$

We can see that event $e_B(\text{false})$ has been filtered out, as it is a branching event happening inside an atomic event. The cost of the execution is 6, as we sum both $e_R(2)$ and $e_R(4)$ events, even though the latter appears inside an atomic event.

This projection function selectively erase some part of the leakage that appears inside atomic events. Compared to the CCT policy, this is a relaxation, as we no longer expect two indistinguishable executions to emit the same list of memory accesses and branchings inside of atomic annotation. In other words, secret-dependent branchings and memory accesses are then allowed by the CR$^\#$ policy, inside of atomic annotations only.

However, this relaxation is limited by the resource consumption. Indeed, the CR$^\#$ policy always expects two indistinguishable executions to have an equivalent resource consumption, regardless of atomic annotations. As a consequence, all secret-dependent branches in a program will have to be balanced, or to balance each other, in order to maintain a constant global resource consumption for such executions.

Examples of CR$^\#$-secure Programs

We present below a few programs, and discuss whether or not they respect the CR$^\#$ policy.

Example 12: Examples of CR$^\#$-secure programs.

P_1: `if(b) {δ(1)} else {δ(2)}`

P_2: `if(b) {δ(1); δ(2)} else {δ(3)}`

P_3: `if(b) {δ(2)} else {δ(3)}`; `δ(4)`; `if(b) {δ(4)} else {δ(3)}`

P_4: `if(b) {δ(2)} else {δ(3)}`; `δ(4)`; `if(b) {δ(4)} else {δ(3)}`

For a program without atomic statement, the CR$^\#$ policy is equivalent to the CCT policy. Therefore, P_1 is CR$^\#$-secure if and only if its condition `b` is public. Both

branches of P_2 are balanced with a cost of 3. P_2 is CR$^\#$-secure if and only if its condition is public, just like P_1. However, $\boxed{P_2}$ is always CR$^\#$-secure, as its branches are balanced and inside atomic statements. P_3 contains two unbalanced atomic branches that balance each other. Indeed, whatever the initial value of b, the total amount of consumed resources is 7. Therefore, P_3 is always CR$^\#$-secure. This example illustrates the fact that CR$^\#$ leakages are not non-cancelling: a CR$^\#$-secure program can be composed of several non-CR$^\#$-secure programs, that balance each other. P_4 is similar to P_3, but only its second branch is atomic. If b is public, then P_4 is CR$^\#$-secure. However, if b is secret, P_4 may never be CR$^\#$-secure because of the first branch, even if as previously, branches balance each other. It illustrates that only atomic branches can be used to balance each other.

We now focus on the problem of the preservation of the CR$^\#$ policy by a transformation. A transformation $\mathcal{T}$ is CR$^\#$-preserving when given a CR$^\#$-secure program P, then $\mathcal{T}(P)$ is a CR$^\#$ program. The underlying hypothesis is that P starts from two indistinguishable states and so does $\mathcal{T}(P)$. However, these two pairs of state are not related. In the same way, there is no relation between the two leakages observed during the two executions of P and $\mathcal{T}(P)$. This definition expresses the preservation of our CR$^\#$ policy, but it is too general to be proved by a simple induction. For that reason, we define in Section 4.3.3 and Section 4.3.4 less general preservation properties that fit to our program transformations and are easier to prove.

Definition 4.10: CR$^\#$ preservation.

A transformation $\mathcal{T}$ is CR$^\#$-preserving when, for any program p:

$$\mathrm{CR}^\#(p) \implies \mathrm{CR}^\#(\mathcal{T}(p))$$

Note that this a direct instance of **Definition 2.22**, using the CR$^\#$ policy.

Relations between CCT, CR and CR$^\#$

We highlight here some interesting consequences of the definition of the CR$^\#$ policy. Firstly, we can express the CR policy with the CR$^\#$ policy. For a program p, we have $CR(p) \iff \mathrm{CR}^\#(\boxed{p})$. In other words, CR is an instance of CR$^\#$, obtained by annotating the whole program with an atomic annotation. Secondly, we can also express the CCT policy with the CR$^\#$ policy. For a program p without any atomic annotation, then CR$^\#(p)$ forbids any secret-dependent branch in p. For such a program p, we then have the equivalence $\mathrm{CCT}(p) \iff \mathrm{CR}^\#(p)$. As a consequence, it is relevant to consider the

CR$^\#$ policy as a flexible mix between the CR and the CCT policies. The CR$^\#$ policy can observe both behaviors, depending on the added atomic annotations. It behaves as CR inside of annotations, and as CCT outside of them.

Last, if we consider a program p which is CR-secure, then it is always safe to assume that there exists a way to annotate it (denoted $p^\#$) so that we have CR$^\#(p^\#)$. Indeed, $\boxed{p}$ is always a valid candidate. However, we will see in the next section that atomic annotations also restrict the compiler. Therefor, finding candidates that contain fewer annotations yields more efficient optimizations.

4.3.2 Implementation of Control-flow Preserving Transformations

This section explains how to adapt in a CR$^\#$-preserving way three optimizations that are likely not to preserve the CR$^\#$ policy: constant folding, CSE and dead-store elimination. These three usual transformations allow us to fully illustrate our approach. Constant folding replaces any expression evaluating to a constant value by the value itself. Every expression $e_1 \diamond e_2$, where e_1 and e_2 evaluate respectively to n_1 and n_2 is replaced by $n_1 \diamond n_2$. This transformation modifies expressions (e.g., `1+2` becomes `3`), hence the resource consumption, and is thus likely to break the CR$^\#$ policy. For example, this transformation modifies `x := 1+2` into `x := 3`, whose evaluation cost is lower. If the instruction `x := 1+2` appears in a balanced secret-dependent if-branch, the balance could be broken.

CSE computes available expressions at every program point. Any available expression is replaced by the variable storing the result of its evaluation. CSE also introduces assignments to store intermediary results, hence modifying the resource consumption. Dead-store elimination aims at removing any occurrence of an assignment that is not used later. Again, removing an instruction reduces the resource consumption and may break the CR$^\#$ policy.

The above examples share a similar structure: the composition of a data-flow analysis, which does not modify the program, and of elementary transformations performing the optimization. They consist mainly of substituting an expression with another expression, introducing an assignment instruction, and removing an assignment instruction.

We implement each elementary transformation by compensating any modification in the resource consumption with a padding $\boldsymbol{\delta}$ instruction. Introducing an adequate padding requires to compute the resource consumption of the evaluation of an expression. The resource consumption of the execution of an expression does not depend on the current state. We thus introduce a cost function $\mathcal{Q} : exp \rightarrow Z$, that statically computes the cost of an expression.

Given an expression e, the substitution transformation $\mathcal{S}$ replaces the expression in the right-hand side of an assignment with e. In order to preserve the resource consumption,

$\mathcal{S}$ uses $\mathcal{Q}$ to add padding, hence $\mathcal{S}$ is parameterized by an expression e and a statement, and we denote $\mathcal{S}_e$ the substitution $\mathcal{S}$ with parameter e. $\mathcal{S}_e : stmt \rightarrow stmt$ is defined as follows: $\mathcal{S}_e(id \texttt{ := } e') = id \texttt{ := } e\texttt{;}\ \boldsymbol{\delta}\texttt{(}\mathcal{Q}(e') - \mathcal{Q}(e)\texttt{)}$. Any other statement is not modified by this transformation.

The insertion transformation stores the result of a temporary computation e into a fresh variable. It inserts an assignment before an arbitrary statement. The insertion is compensated with a negative padding. The insertion transformation $\mathcal{I}_e : stmt \rightarrow stmt$ is defined as follows: $\mathcal{I}_e(p) = tmp \texttt{ := } e\texttt{;}\ \boldsymbol{\delta}\texttt{(}-K^{asn} - \mathcal{Q}(e)\texttt{)}\texttt{;}\ p$, where tmp is a fresh variable identifier never used in p, and K^{asn} represents the cost of an assignment. The removal $\mathcal{R}$ transformation replaces an assignment with an adequate padding. More precisely, $\mathcal{R} : stmt \rightarrow stmt$ is defined as $\mathcal{R}(id \texttt{ := } e) = \boldsymbol{\delta}\texttt{(}K^{asn} + \mathcal{Q}(e)\texttt{)}$.

By using these three elementary transformations (substitution, insertion and removal), and heuristic functions, any control-flow preserving transformation, such as CSE or constant propagation, can be implemented. We do not detail any further the complete definitions of such transformations, and only focus on the three elementary transformations presented above. Indeed, we argue that they constitute the building blocks of any control-flow preserving transformation. Our goal is now to prove that these elementary transformations preserve the CR$^{\#}$ policy.

4.3.3 Using Leakage Preservation to prove CR$^{\#}$ preservation

In order to prove that transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ are CR$^{\#}$-preserving, we can not use standard induction reasoning. This is a consequence of the fact that CR$^{\#}$ leakages are not non-cancelling. Our solution is to proceed in two steps and conduct simpler proofs. First, we define a property called leakage preservation that is more constrained than CR$^{\#}$-preservation, and we prove that each transformation is leakage preserving. Then, we prove once for all that leakage preservation implies CR preservation. Moreover, we apply this proof scheme to the first pass of the $\mathcal{M}$ transformation introduced in Section 4.1.1 to minimise padding.

Leakage Preservation implies CR$^{\#}$ Preservation

We define a transformation as leakage preserving when it does not modify the leakage (i.e., resource consumption and emitted boolean leakages). It is then more constrained than the CR$^{\#}$ preservation. We define leakage preservation as a backward property that is required by theorem **Theorem 4.6**: given a property of the transformed program, it states a property of the source program.

Definition 4.11: Leakage preservation.

Let π be a leakage projection function. A transformation $\mathcal{T}$ is leakage preserving with respect to π if, for any program p, given an execution $\langle \mathcal{T}(p), \sigma \rangle \Downarrow (\sigma_2, \ell_2)$ of the transformed program, there exists an environment σ_1 and a leakage ℓ_1 such that we have $\langle p, \sigma \rangle \Downarrow (\sigma_1, \ell_1)$ and $\pi(\ell_1) = \pi(\ell_2)$. We denote it as LP_π.

Lemma 4.5: $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ leakage preserving.

Transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ are LP_{π_3}, i.e., they preserve the leakage projected with function π_3.

We do not detail the proof of this lemma, as it is a direct consequence of the definition of the transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$

Theorem 4.6: Leakage preservation implies ONI preservation.

Let π be a leakage projection function. Any LP_π transformation is ONI_π-preserving.

Proof of Theorem 4.6.

Let π be a leakage projection function. Let $\mathcal{T}$ be a LP_π transformation, we want to prove that $\mathcal{T}$ is ONI_π-preserving. Let p be a $\mathrm{CR}^\#$-secure program, we need to prove that $\mathcal{T}(p)$ is ONI_π-secure as well. To this end, we assume having two indistinguishable executions of the transformed program $\mathcal{T}(p)$, and we then need to prove that both executions emit the same leakage. As $\mathcal{T}$ is LP_π, we can find two similar executions of p, which are indistinguishable as well, and emit the same projected leakages; this highlights the fact that we need leakage-preservation to be a backward property. Next, as we know that p is ONI_π-secure, we can deduce that both executions of p emit the same projected leakage. As these leakages are identical to the one emitted by the two executions of $\mathcal{T}(p)$, this concludes the proof. □

We can now combine these results to prove that transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ all preserve the $\mathrm{CR}^\#$ policy. This is a direct consequence of **Lemma 4.5** and **Theorem 4.6**.

Theorem 4.7: Elementary transformations are $\mathrm{CR}^\#$-preserving.

Elementary transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ are $\mathrm{CR}^\#$-preserving.

Leakage Preservation of the Normalisation Transformation

This section defines the normalisation transformation $\mathcal{N}$ and shows that it is a leakage-preserving pass. The transformations $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{R}$ may introduce many $\boldsymbol{\delta}$ instructions, that increase the overall resource consumption, and are then factorised and

minimised by the $\mathcal{M}$ pass. $\mathcal{M}$ is designed to minimize $\boldsymbol{\delta}$ instructions locally to every atomic block, and will not try to balance out $\boldsymbol{\delta}$ instructions across different atomic blocks. We decompose $\mathcal{M}$ into a normalisation pass $\mathcal{N}$ followed by a deletion pass $\mathcal{D}$ and we prove that $\mathcal{N}$ is leakage preserving, contrary to $\mathcal{D}$ (that is discussed in Section 4.3.4). We define the $\mathcal{N}$ pass as follows:

Definition 4.12: Transformation $\mathcal{N}$.

The $\mathcal{N}$ pass repeatedly performs the four following basic operations until convergence, in order to merge and factorise as many $\boldsymbol{\delta}$ instructions as possible. These four operations preserve the resources consumed during an execution, and are defined as rewrite rules, using the $\Rrightarrow_{\mathcal{N}}$ notation .

1. *Move upwards.* First, $\boldsymbol{\delta}$ instructions are moved as upward as possible, in order to further group them together. Formally, $\mathcal{N}$ performs the following operation:

$$p;\ \boldsymbol{\delta}(n) \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}(n);\ p$$

2. *Merge.* Then, $\boldsymbol{\delta}$ instructions are merged:

$$\boldsymbol{\delta}(n);\ \boldsymbol{\delta}(m) \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}(n+m)$$

3. *Factorise ticks out of branches.* Next, whenever a $\boldsymbol{\delta}$ instruction appears in an if-branch, it is factorised when it appears on the opposite branch as well. Formally:

$$\texttt{if}(b)\ \{\boldsymbol{\delta}(n_1);\ p_1\}\ \texttt{else}\ \{\boldsymbol{\delta}(n_2);\ p_2\} \Rrightarrow_{\mathcal{N}}$$

$$\boldsymbol{\delta}(n);\ \texttt{if}(b)\ \{\boldsymbol{\delta}(n_1-n);\ p_1\}\ \texttt{else}\ \{\boldsymbol{\delta}(n_2-n);\ p_2\}$$

where n is the minimum between n_1 and n_2. Any resulting $\boldsymbol{\delta}(0)$ instruction is deleted. For example, we have:

$$\texttt{if}(b)\ \{\boldsymbol{\delta}(3);\ p\}\ \texttt{else}\ \{\boldsymbol{\delta}(5);\ s\} \Rrightarrow_{\mathcal{N}}$$

$$\boldsymbol{\delta}(3);\ \texttt{if}(b)\ \{p\}\ \texttt{else}\ \{\boldsymbol{\delta}(2);\ s\}$$

4. *Move out of atomic.* Whenever a $\boldsymbol{\delta}$ appears in an atomic annotation, outside of a branch, we move it out. This is the main step of the normalisation pass; it reduces the amount of $\boldsymbol{\delta}$ instructions inside of atomic annotations, to prepare for the deletion pass $\mathcal{D}$. Formally:

$$\boxed{\boldsymbol{\delta}(n);\ p} \Rrightarrow_{\mathcal{N}} \boldsymbol{\delta}(n);\ \boxed{p}$$

$\mathcal{N}$ performs these operations, along with recursive calls for the sequence of two instructions, if-branch, while loop, and atomic constructs.

We can additionally notice that $\mathcal{N}$ does not move any $\boldsymbol{\delta}$ instruction outside of a loop. We argue that it is not necessary to try to do so, for the following reason. If a loop is present outside an atomic annotation, the following $\mathcal{D}$ transformation will optimize it identically (see Section 4.3.4). If a loop is present inside an atomic annotation, it may appear in a secret-dependent branch. This practice may be dangerous, but is still tolerated by our CR$^{\#}$ policy: such a choice is the programmer's responsibility. However, we prefer not to optimize the loop in this case, as this situation is not realistic.

Lemma 4.8: $\mathcal{N}$ is leakage preserving.

Transformations $\mathcal{N}$ is LP$_{\pi_3}$, i.e., it preserves the leakage projected with function π_3. $\mathcal{N}$ is leakage preserving.

By combining **Lemma 4.8** and **Theorem 4.6**, we can conclude that $\mathcal{N}$ preserves the CR$^{\#}$ policy.

Theorem 4.9: $\mathcal{N}$ is CR$^{\#}$-preserving.

Transformations $\mathcal{N}$ is CR$^{\#}$-preserving.

4.3.4 A Cornerstone non Leakage-Preserving Transformation

This section is devoted to our last and trickiest transformation to prove CR$^{\#}$ preserving, the second pass $\mathcal{D}$ of the $\mathcal{M}$ minimisation of $\boldsymbol{\delta}$ instructions (introduced in Section 4.1.1). $\mathcal{D}$ deletes unnecessary $\boldsymbol{\delta}$ instructions while preserving balanced branches in atomic annotations. First, this section defines $\mathcal{D}$. Then, it justifies why it is CR$^{\#}$-preserving. Once again, we decompose the proof in two parts and define a stronger property than CR$^{\#}$ preservation.

Deletion Pass $\mathcal{D}$

The $\mathcal{D}$ pass is given in the following definition:

Definition 4.13: Transformation $\mathcal{D}$.

The $\mathcal{D}$ pass is defined in using rewrite rules, using notation $\Rrightarrow_{\mathcal{D}}$.

$$\frac{}{\texttt{skip} \Rrightarrow_{\mathcal{D}} \texttt{skip}} \qquad \frac{}{\boldsymbol{\delta}(n) \Rrightarrow_{\mathcal{D}} \texttt{skip}}$$

$$\frac{p_1 \Rrightarrow_{\mathcal{D}} p_1' \qquad p_2 \Rrightarrow_{\mathcal{D}} p_2'}{p_1\,;\ p_2 \Rrightarrow_{\mathcal{D}} p_1'\,;\ p_2'}$$

$$\frac{p_1 \Rrightarrow_{\mathcal{D}} p_1' \qquad p_2 \Rrightarrow_{\mathcal{D}} p_2'}{\texttt{if}(c)\ \{p_1\}\ \texttt{else}\ \{p_2\} \Rrightarrow_{\mathcal{D}} \texttt{if}(c)\ \{p_1'\}\ \texttt{else}\ \{p_2'\}}$$

$$\frac{p \Rrightarrow_{\mathcal{D}} p'}{\texttt{while}(c)\ \{p\} \Rrightarrow_{\mathcal{D}} \texttt{while}(c)\ \{p'\}} \qquad \frac{}{\boxed{p} \Rrightarrow_{\mathcal{D}} \boxed{p}}$$

Our CR$^{\#}$ policy considers that balance between branches must only hold inside of atomic annotations. So, deleting a $\boldsymbol{\delta}$ instruction outside of an atomic annotation has no effect on any balanced branch. The transformation $\mathcal{D}$ thus deletes any occurrence of a $\boldsymbol{\delta}$ instruction outside of an atomic annotation. However, $\boldsymbol{\delta}$ instructions inside of atomic annotations are not deleted, in order to preserve the balance between potential balanced secret-dependent branches. For example, we have $\boldsymbol{\delta}(2)\,;\ \boxed{\boldsymbol{\delta}(3)} \Rrightarrow_{\mathcal{D}} \boxed{\boldsymbol{\delta}(3)}$.

Proving that the Deletion Pass is CR$^{\#}$ Preserving

Contrary to all the passes presented so far, $\mathcal{D}$ deletes some $\boldsymbol{\delta}$ instructions, hence explicitly modifying the resource consumption. Therefor, $\mathcal{D}$ is clearly not leakage preserving. Instead, we rely on the property that $\mathcal{D}$ only removes $\boldsymbol{\delta}$ instructions that are outside of atomic annotations. As a consequence, its impact on resource consumption must not depend on secret input values, as all secret dependent if-branches appear inside an atomic annotation.

More precisely, for a given program, let us consider two executions emitting the same boolean leakage, meaning that outside of atomic annotations, both executions follow the same path during the execution. Transforming both executions with $\mathcal{D}$ will have the same impact on resource consumption. Indeed, outside of atomic annotations, as both executions follow the same path, the transformation will have a similar impact on resource consumption. Moreover, inside of atomic annotations, the program is not modified, and neither are the executions.

To capture this idea, we define a new policy called Leakage Preservaion with Offset (LPO). It is defined as follows:

Definition 4.14: Leakage Preservation with Offset (LPO).

Let π be leakage projection function. For any program p, we assume to have two of its

executions verifying:

$$\langle p,\, \sigma_1\rangle \Downarrow (\sigma'_1, \ell_1)$$
$$\langle p,\, \sigma_2\rangle \Downarrow (\sigma'_2, \ell_2)$$
$$\pi(\ell_1) = \pi(\ell_2)$$

If we can find leakages ℓ'_1, ℓ'_2 and r (we call r the offset leakage) such that we have:

$$\langle p,\, \sigma_1\rangle \Downarrow (\sigma'_1, \ell'_1)$$
$$\langle p,\, \sigma_2\rangle \Downarrow (\sigma'_2, \ell'_2)$$
$$\pi(r \cdot \ell_1) = \pi(\ell'_1)$$
$$\pi(r \cdot \ell_2) = \pi(\ell'_2)$$

we then say that program p is LPO with respect to π. We denote it as $\mathrm{LPO}_\pi(p)$.

LPO is not sufficient to imply that the transformation is $\mathrm{CR}^\#$-preserving. We further require the transformation to satisfy a property called "termination preservation". It states that if an execution of a transformed program from an initial state σ terminates, then an execution of the source program from the same initial state σ also terminates. Similarly to leakage preservation, termination preservation is a backward property.

Definition 4.15: Termination preservation.

A transformation $\mathcal{T}$ is termination preserving if, for any program p, supposing that we have an execution $\langle \mathcal{T}(p),\, \sigma\rangle \Downarrow (\sigma_2, \ell_2)$ of the transformed program, then there exists an output state σ_1 and a leakage ℓ_1 such that we have the execution $\langle p,\, \sigma\rangle \Downarrow (\sigma_1, \ell_1)$.

The following lemma states that any transformation complying to the two previous properties is ONI-preserving. We then prove that $\mathcal{D}$ complies to both properties, hence to $\mathrm{CR}^\#$ preservation.

Theorem 4.10: LPO implies ONI preservation.

Let π be a leakage projection function. Any LPO_π and termination preserving transformation is ONI_π-preserving.

Proof of Theorem 4.10.

Let $\mathcal{T}$ be a transformation, that is LPO_π and termination preserving. Let p be a ONI_π-secure program, we need to prove that $\mathcal{T}(p)$ is ONI_π-secure as well. To this end, we assume having two indistinguishable executions E_1 and E_2 of $\mathcal{T}(p)$, and we then need to prove that they emit the same leakage. As $\mathcal{T}$ is termination preserving, we can find two executions of p emitting leakages that we call ℓ_1 and ℓ_2. As p is ONI_π-secure, these leakages verify $\pi(\ell_1) = \pi(\ell_2)$.

As the source executions emit the same projected leakage, we use the fact that $\mathcal{T}$ is LPO_π to find leakages ℓ'_1, ℓ'_2 and a leakage offset r such that we have two target executions of $\mathcal{T}(p)$ with identical initial states and emitting respective leakages ℓ'_1 and ℓ'_2, and such that $\pi(r \cdot \ell'_1) = \pi(r \cdot \ell'_2)$. We deduce that $\pi(\ell'_1) = \pi(\ell'_2)$ from the hypotheses we have on the π function.

Finally, as the semantics of $\mathcal{L}^\#$ is deterministic (see **Lemma 4.1**), these two target executions of $\mathcal{T}(p)$ are exactly the executions E_1 and E_2. As E_1 and E_2 emit the leakages ℓ'_1 and ℓ'_2 verifying $\pi(\ell'_1) = \pi(\ell'_2)$, this concludes the proof. □

Now, we can use this strong result to show that $\mathcal{D}$ is $\text{CR}^\#$-preserving. First, we need to prove that it is termination preserving (we do not detail this proof here):

Lemma 4.11: $\mathcal{D}$ is termination preserving.

Transformation $\mathcal{D}$ is termination preserving.

We do not detail this proof, that can be conducted with a simple inductive reasoning on the semantics of the program being transformed. Next, we show that $\mathcal{D}$ satisfies the criterion LPO.

Lemma 4.12: $\mathcal{D}$ is LPO.

Transformation $\mathcal{D}$ is LPO_{π_3}, i.e., it is LPO with respect to leakage projection π_3.

Proof of Lemma 4.12.

Let p be a program, we assume to have two executions of p emitting the same boolean leakage l, and consuming respectively q_1 and q_2 resources. We need to find an offset $r \in \mathbb{Z}$, such that the associated executions of $\mathcal{D}(p)$ emit the same boolean leakage l, and consume respectively $q_1 + r$ and $q_2 + r$ resources. We reason by induction on the semantics of one of the source executions. We focus on some of the interesting cases.

- Case TICK. We have $p = \boldsymbol{\delta}\texttt{(}n\texttt{)}$ and $\mathcal{D}(p) = \texttt{skip}$. We choose $r = -n$.
- Case SEQ. We have $p = p_1\texttt{;}\ p_2$, two executions of p_1 and two executions p_2. As the boolean leakage is non-cancelling (see **Lemma 4.2**), we conclude that both executions of p_1 emit the same boolean leakage l_1, and both executions of p_2 emit the same boolean leakage l_2. Then we use the induction hypothesis and these two pairs of executions to conclude.
- Case IF. We have $p = \texttt{if(}e\texttt{)}\ \texttt{\{}p_1\texttt{\}}\ \texttt{else}\ \texttt{\{}p_2\texttt{\}}$. As both executions have the same boolean leakage, e evaluates to the same value in both executions. When e evaluates to true (resp. false), we have two executions of p_1 (resp. p_2), producing the same boolean leakage. We then use the induction hypothesis on the executions of p_1 (resp. p_2) to conclude.

□

By combining **Lemma 4.11**, **Lemma 4.12** and **Theorem 4.10**, we can conclude that $\mathcal{D}$ preserves the CR$^{\#}$ policy.

Theorem 4.13: $\mathcal{D}$ is CR$^{\#}$-preserving.

Transformations $\mathcal{D}$ is CR$^{\#}$-preserving.

4.3.5 Enforcing CR$^{\#}$ with a Type-System

In subsection 2.5.3, we showed how to adapt the Secure Flow type-system presented in section 2.2 in order to enforce the CCT policy. Similarly, we present here how to adapt this type-system in order to enforce the CR$^{\#}$ policy.

This type-system, that we call the CR$^{\#}$ type-system, is designed to keep track of the resource consumption in the type of program. In the type-system, a cost is an element from $\mathbb{Z} \cup \{\mathbf{?}\}$, i.e., an integer or a special element $\mathbf{?}$. We use element $\mathbf{?}$ whenever we can not statically conclude on the cost of a program construct, e.g., in loops. We extend the addition on integers to $\mathbb{Z} \cup \{\mathbf{?}\}$, such that for any $q \in \mathbb{Z} \cup \{\mathbf{?}\}$, $q + \mathbf{?} = \mathbf{?} + q = \mathbf{?}$. The type-system relies on the same set of constants $K^{\cdots}$ used to define the instrumented semantics of the language.

Our type-system is also able to insert atomic annotations in the program. Formally, the type-system takes as input a program $p \in \mathscr{L}$, i.e., that does not contain any atomic annotation. Typing p also produces a program $p^{\#}$ in $\mathscr{L}^{\#}$, which is similar to p, but only differs from p by the inserted atomic annotations.

The typing judgment for expressions is $\Gamma \vdash e : (\tau, q)$, with e an expression, τ an element of the security lattice $\{L, H\}$, and q an element of $\mathbb{Z} \cup \{\mathbf{?}\}$. It reads as e has type (τ, q) in Γ. If q is not $\mathbf{?}$, the type-system is designed to ensure that any execution of e costs q.

We define the type-system as follows, both for expressions and statements:

Definition 4.16: CR$^{\#}$ type-system for expressions $\Gamma \vdash e : \tau, q$.

$$\text{CONST}\ \frac{}{\Gamma \vdash n : (\tau, n)} \qquad \text{VAR}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : (\tau, K^{var})} \qquad \text{ARRAY}\ \frac{\Gamma[a] = L \qquad \Gamma \vdash e : (L, q)}{\Gamma \vdash a\texttt{[}e\texttt{]} : (L, K^{array} + q)}$$

$$\text{OP_BIN}\quad \frac{\Gamma \vdash e_1 : (\tau, q_1) \qquad \Gamma \vdash e_2 : (\tau, q_2)}{\Gamma \vdash (e_1 \diamond e_2) : (\tau, q_1 + q_2)} \qquad \text{SUBTYPE_E}\quad \frac{\Gamma \vdash e : (\tau_1, q) \qquad \tau_1 \leq \tau_2}{\Gamma \vdash e : (\tau_2, q)}$$

The typing judgment for statements is $\Gamma, s \vdash s^{\#} : (\tau, q)$, with $s \in \mathscr{L}$, $s^{\#} \in \mathscr{L}^{\#}$, τ an element of the security lattice $\{L, H\}$, and q an element of $\mathbb{Z} \cup \{\mathbf{?}\}$. It reads as s has type (τ, q) in Γ, and infers the annotated statement $s^{\#}$. If q is not $\mathbf{?}$, the type-system is designed to ensure that any execution of s costs q.

Definition 4.17: CR$^{\#}$ type-system for statements $\Gamma, s \vdash s^{\#} : \tau, q$.

$$\text{ASN}\quad \frac{\Gamma(x) = \tau \qquad \Gamma \vdash e : (\tau, q)}{\Gamma, (x \texttt{ := } e) \vdash (x \texttt{ := } e) : (\tau, K^{asn} + q)}$$

$$\text{ARRAY_ASN}\quad \frac{\Gamma[a] = L \qquad \Gamma \vdash e_1 : (L, q_2) \qquad \Gamma \vdash e_2 : (L, q_2)}{\Gamma, (a\texttt{[}e_1\texttt{]} \texttt{ := } e_2) \vdash (a\texttt{[}e_1\texttt{]} \texttt{ := } e_2) : (L, K^{array_asn} + q_1 + q_2)}$$

$$\text{SEQ}\quad \frac{\Gamma, s_1 \vdash s_1^{\#} : (\tau, q_1) \qquad \Gamma, s_2 \vdash s_2^{\#} : (\tau, q_2)}{\Gamma, (s_1; s_2) \vdash (s_1^{\#}; s_2^{\#}) : (\tau, q_1 + q_2)}$$

$$\text{IF_LOW}\quad \frac{\Gamma \vdash e : (L, q) \qquad \Gamma, s_1 \vdash s_1^{\#} : (L, q_1) \qquad \Gamma, s_2 \vdash s_2^{\#} : (L, q_2)}{\Gamma, (\texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) \vdash (\texttt{if}\, e\, \texttt{then}\, s_1^{\#}\, \texttt{else}\, s_2^{\#}) : (L, \mathbf{?})}$$

$$\text{IF_BALANCED}\quad \frac{\Gamma \vdash e : (\tau, q) \qquad \Gamma, s_1 \vdash s_1^{\#} : (\tau, q') \qquad \Gamma, s_2 \vdash s_2^{\#} : (\tau, q')}{\Gamma, (\texttt{if}\, e\, \texttt{then}\, s_1\, \texttt{else}\, s_2) \vdash (\texttt{if}\, e\, \texttt{then}\, s_1^{\#}\, \texttt{else}\, s_2^{\#}) : (\tau, q + q')}$$

$$\text{SKIP}\quad \frac{}{\Gamma, \texttt{skip} \vdash \texttt{skip} : \tau} \qquad \text{WHILE}\quad \frac{\Gamma \vdash e : (\tau, q) \qquad \Gamma, s \vdash s^{\#} : (\tau, q')}{\Gamma, (\texttt{while}\, e\, \texttt{do}\, s) \vdash (\texttt{while}\, e\, \texttt{do}\, s^{\#}) : (\tau, \mathbf{?})}$$

$$\text{SUBTYPE_S}\quad \frac{\Gamma, s \vdash s^{\#} : (H, q) \qquad q \neq \mathbf{?}}{\Gamma, s \vdash \boxed{s^{\#}} : (\tau, q)}$$

The idea of this type-system is to syntactically annotate the program anytime the

subtyping rule is used. We therefore ensure that high-security parts any only present inside atomics in the annotated program. The type-system also ensures that loops and unbalanced branching are not present in high-security parts (i.e., atomics).

We can prove the following soundness result:

Theorem 4.14: Soundness of the CR$^{\#}$ type-system.

Let p be a program. If p is well-typed, i.e., $\Gamma, p \vdash p' : (\tau, q)$, then p is CR-secure and the inferred program $p^{\#}$ is CR$^{\#}$-secure.

4.4 Related Work

Our work focuses on a timing non-interference policy, which was first introduced in [Aga00]. In [Aga00], the authors define a type system in which well-typed programs do not leak secret information. This is a direct adaptation of [VIS96] but it adds control of timing leaks on high branches. Their type system is undecidable because they rely on an undecidable semantic judgement to check that high branches have equal timing costs. But their approach can be refined with a more conservative judgement to become executable and directly adapted to enforce the CR policy. When high branches are not time-balanced but the program is typable with respect to the type system of [VIS96], they also show how to repair the program by suitably padding both branches. Using this approach in our setting could help repairing CR after a unsecure compiler transformation but will require running the type checking of [VIS96] after each compilation pass, even on low-level languages where taint analysis is often too conservative to succeed. Our approach avoids running a taint analysis inside the compiler, thanks to our atomic annotations.

In [Ngo+17], the authors introduce a timing non-interference policy called CR. We extended this policy to also include control-flow leakages. They present a type system used to verify that an implementation respects their policy. They also show how this type system can automatically remove vulnerabilities from a program. In other works [HAH11; Çiç+17], the authors use a similar notion of resource consumption to establish precise bounds for worst and best cases resource usage. The main difference with our work is that we focus on the preservation of a variation of this security policy. Our current paper does not put to much emphasis on enforcing CR and CR$^{\#}$ at source level because the work of [Aga00] can be easily adapted to do it, by positioning carefully atomic annotations at source level.

The work that is closest to ours is [Ath+18], where the authors use another relaxation of the CCT policy called time balancing and defined as negligibly influenced by secrets. To ensure that a program respects this policy, a global timing counter is added to measure

the timing differences between branchings. Then, the Boogie deductive verifier checks for each program the constraints required by the policy. This work is not formally verified with a proof assistant but a tool was implemented to verify that the Amazon's s2n implementation of TLS respects the time-balancing policy. Similarly to our work, the author use padding or dummy instructions in order to balance branches in the program. Our work differs as we use an instrumented operational semantics to model the timing behavior of a program. Their policy also differs, as it allows pairs of executions with different execution time, as long as this difference is bounded by a given constant value.

4.5 Conclusion

This concludes this chapter. We formalised the $CR^{\#}$ security policy, a strong policy defined as a mix between the CCT policy, and the CR policy which is used by some cryptographic practitioners. We also formalised a methodology to adapt program transformations so that they become $CR^{\#}$ preserving; it relies on adding padding to balance secret branchings and minimisation of padding. We proved that different transformations used by compiler optimizations are $CR^{\#}$ preserving. Last, we introduced an annotation called atomic, used to delimit the high-security parts of the program. This annotation indicates where to restrict the compiler optimizations in order to preserve the $CR^{\#}$ policy.

Chapter 5

Conclusion

In this thesis, we studied the problem of preservation of security policies against timing side-channel attacks, from two perspectives. We first tackled this problem from a compilation perspective. We modified a realistic compiler, CompCert, by adapting 2 of the 17 passes of the compilation chain, in order to make sure that the cryptographic constant-time security policy is preserved during the compilation. We also ran experimental validation to show that the impact on the performance of the generated code is very low. At first, modifying the code of a large project such as CompCert was intimidating; the main difficulty we encountered was to find the right transformations to modify in order to keep the changes to the compiler as light as possible, and then to adapt the proof of correctness of the modified transformations.

We then tackled this question by focusing on a different security policy called Constant-Resource (CR). To the best of our knowledge, the preservation of this policy during compilation had never been tackled in the literature. This problem has proved to be challenging, especially since usual proof techniques, used to study the preservation of the constant-time policy for instance, can not be used in our case. This led us to introduce a more flexible security policy. We studied this policy, introduced new proof schemes allowing to show that a transformation preserves this new policy, and applied this methodology to usual control-flow preserving optimizations.

In section 5.1, we summarize the results presented in this thesis, and we discuss short-term improvements and perspectives in section 5.2.

5.1 Summary

In chapter 2, we first formally defined the Non-Interference (NI) policy, and presented a secure-flow type-system classically used to enforce the NI policy. We then conducted a didactic proof of the soundness of this type-system, in the case of a language equipped with a small-step semantics with continuations. Next, based on this development, we illustrated that such type-systems are unpractical when used in a compilation chain. Specifically, we presented a simple (and realistic) transformation, aiming at concatenating arrays, and showed that the compiled program can not always be typable by this type-system. Last, we introduced the Observational Non-Interference (ONI) policy, a generic policy, based on

leakages, that we later used to instantiate other security policies, such as constant-time or constant-resource. We studied the preservation of ONI policies during a transformation, in a way that does not rely on a taint-tracking mechanism, such as a secure-flow type-system.

In chapter 3, we first illustrated how real life compilers (e.g., GCC or Clang), break the constant-time policy. More precisely, we highlighted examples of secure C programs, that are compiled into unsecure programs. These behaviors existed as well in the version 3.4 of CompCert, and we then presented our work aiming at removing them from CompCert, so that it strictly preserves the constant-time policy. We then detailed our modifications, that impacted two of the passes of the compiler, namely *Instruction Selection* and *Assembly Generation*. Prior to our work, Instruction Selection was responsible for the non-preservation of the constant-time policy. Indeed, this pass introduced branching when compiling specific conversion operations, e.g., between unsigned integer and floating point values. We fixed these conversion operations, by replacing the introduced branchings with a new instruction called *select*. During Assembly Generation, this newly introduced instruction is then compiled to `cmov` x86 instructions, which are usually used to perform an efficient branchless choice in constant-time implementations. Last, we discussed how these changes were integrated in CompCert (in version 3.6), before experimentally evaluating our modifications. We showed that our modifications have a very low impact on the performance of the generated code.

In chapter 4, we studied the problem of preservation of a ONI policy called Constant-Resource (CR). Studying this policy proved to be challenging, as usual proof techniques used for other ONI policies can not be applied in the case of the CR policy. In particular, existing methodologies critically rely on leakage non-cancellation, which is not verified in our case. This led us to propose a new methodology to use in order to show that a transformation preserves the CR policy.

The CR policy relies on a resource consumption model, and is defined with respect to an instrumented semantics that tracks the amount of resource consumed during the execution of a program. The CR policy only ensures that the resource consumption is independent from the secret values in the program. Therefore, the policy tolerates program containing secret-dependent branches, as long as these branches are balanced with respect to the resource model.

We then highlighted through a few examples that this policy is extremely fragile. Indeed, as we chose to focus on a compiler that is not aware of the taints of the variables, any optimization performed in a branch is likely to break the balance in resource consumption, and thus prone to break the security policy. For this reason, we introduced a syntactic annotation called *atomic*, that we assume to be introduced by a prior analysis. We use this annotation to syntactically identify the high-security parts of the program, by selectively choosing the leakage produced by an execution. We then introduced $CR^{\#}$,

a more flexible security policy, defined as an instance of ONI. The atomic annotations are used to selectively ignore some part of the leakage, allowing the policy to accept secret dependent balanced branches. We showed how CR$^{\#}$ can be seen as a mix between the CR and the constant-time policies. Last, we detailed the proof that usual control-flow preserving transformation can be adapted to preserve the CR$^{\#}$ policy, by introducing a minimal amount of padding.

5.2 Perspectives

5.2.1 Experimental Validation

As a first perspective for future work, we consider the experimental validation of the constant-resource policy of chapter 4. The question we consider here is the following: is it possible to provide a non-trivial CR program whose actual execution time does not leak data? As execution time only makes sense at machine-code level, we try to answer this question for machine-code programs only. Our approach relies on the use of an experimental tool called dudect [RBV17], that we now present in detail.

The dudect tool

The dudect [RBV17] tool is designed to evaluate whether or not a program runs with a constant execution-time, given a set of input values. The dudect tool works as follows:

— First, it performs empirical measurements on the tested program, on two sets of inputs: a constant user-defined one (i.e., identical inputs for every measurement), and a randomized one. Execution time is measured using architecture specific cycle counting instructions, which gives the most precise measurements possible. This step then provides two statistical distributions.

— Second, the obtained statistical distributions are post-processed, in order to discard abnormally long executions, which might be triggered by external causes, such as an OS interruption.

— Last, classic statistical test methods are applied in order to prove that the two statistical distributions are different.

In practice, the tool repeats measurements indefinitely and reports any difference in the statistical distribution as a potential leakage. If no leakage is detected, the tool runs forever, and the tested program can then be considered to *probably* have a constant execution-time with respect to input data.

In [RBV17] the authors run their tool on constant-time implementations of cryptographic primitives and their tool did not manage to find any leakage, which empirically validates the constant-time policy. Their experiments still managed to find a leakage in

an implementation of the `curve25519-donna` primitive, when executed on a specific x86 processor performing a variable-time multiplication.

Proposed methodology

We tried to experimentally validate the Constant-Resource (CR) policy using the dudect tool. Our goal here is to find a program that contains a secret-dependent branch, where the branches are balanced so that the dudect tool does not detect any information leakage. In other words, our goal here is to find a program that is not constant-time secure, but that is not experimentally rejected by the dudect tool. We tried to craft such a program by directly crafting it in assembly language and by precisely considering the number of cycles needed to perform each instruction on the processor we run our experiments on.

Unfortunately, microprocessor vendors typically never provide precise information on the timing or number of cycles required to perform an instruction. Our experiments then only rely on values measured empirically [1]. Yet, despite our best efforts, we were not successful in crafting such a program that is not detected as leaking data by the dudect tool. We suspect the speculation mechanism to be responsible for this leaked data, as it makes the number of cycles needed to perform a branch instruction hard to predict. We also tried to use fences instructions, which is typically done to try to reduce Spectre vulnerabilities, as it allows to prevent speculative execution [Wan+19; Bar+21a]. It experimentally made the leakage harder to detect by the dudect tool, but did not remove the leakage entirely.

Still, as explained by the authors in [RBV17], the dudect tool may detect timing leaks that are very hard to detect and may be impossible to exploit in practice. One may then still consider an implementation presenting tiny leakage as usable in practice, as it is done in Amazon's implementation of TLS [Ath+18]. The CR policy is then still usable in practice, even though the absence of leakage seems very hard to achieve.

As future work, we intend to try to apply the proposed methodology to simpler architectures, such as RISC, for which the execution-time is easier to predict. We hope to find non constant-time programs that do not produce timing leakage according to the dudect tool.

5.2.2 Extending the Atomic Annotation to Memory Accesses

In chapter 4, we introduced a new security policy, CR$^{\#}$, that relies on annotations called *atomic*. The main idea of this policy is to syntactically mark some branchings as secret dependent. The CR$^{\#}$ policy is then designed to tolerate secret-dependent branchings, as long as they are marked by an annotation. The program transformations we

1. https://www.agner.org/optimize/instruction_tables.pdf

presented afterwards take the annotation into account, by preserving the resource consumption between the annotated branches.

Another perspective we consider is to extend our approach to take memory accesses into account. Intuitively, we could design a security policy that forbids secret-dependent memory accesses, just like the constant-time or the CR$^{\#}$ policies, but tolerates secret-dependent memory accesses, as long as they are marked by the atomic annotation.

This idea is close to the work presented in [Bar+14], in which the authors present an annotation called *stealth*, whose goal is similar to ours. The motivation of their work comes from the development of system-level mechanisms, allowing to provide a program with a private cache in which even secret-dependent memory accesses, can be performed safely. Such memory locations are called *stealth addresses*. Memory accesses marked as stealth must then naturally be performed at stealth addresses. They then introduce a relaxed security policy called *S-constant-time*, which relies on the stealth annotation they introduce. Intuitively, a program is S-constant-time secure if it does not branch on secrets and only memory accesses to *stealth addresses* may depend on secret. They then present an analysis allowing to verify that a program is secure with respect to the policy.

As a perspective, we then consider mixing our contribution with the one presented above. Our goal is to define a relaxed constant-time policy, in which both secret-dependent branches and memory accesses can be tolerated thanks to syntactic annotations. We then plan to study the preservation of this policy through standard optimization transformations.

Another recent work [Coh+21] also considers the problem of secure compilation relying on syntactic annotations, called *opaque* annotations. The authors introduce a security policy enforcing that any observation occurring in an opaque area must be preserved through compilation. Opaque annotations also restrict the optimizations performed by the compiler, which resemble our atomic annotation. They study the preservation of their policy from C code to machine code, although their implementation is not formally verified. This work contains similarities with our work on the CR$^{\#}$ policy, and we wish to inspire from their contribution in future work.

5.2.3 Spectre Vulnerabilities

New types of attacks have emerged during the last few years, including the threatening Spectre attacks [Koc+19]. These attacks come in several variants, and exploit the speculative execution performed by modern processors, that can speculate on the result of the calculation. Spectre attacks try to discover secret data by recovering data that have been wrongly stored in the cache after a mispeculated execution.

Spectre attacks were discovered in 2018, and have been since extensively studied. This led to the definition of a new security policy called speculative constant-time [Cau+20],

acting as a countermeasure against side-channel attacks. Spectre attacks also raise several challenges, such as the detection of spectre vulnerabilities. This challenge was tackled in particular in [DBR21]. In this paper, the authors present Haunted, a tool that is designed to detect variants of Spectre, called PHT and STL, at binary level. Their tool rely on an instrumented semantics to model speculative execution. They use *symbolic execution*: for a set of symbolic input, a logical formula records the list of branching condition encountered during the symbolic execution. They then use a SMT solver to detect violation of the speculative constant-time policy. They present sketches of proof of the correctness and completeness of their tool.

The problem of repairing a program containing vulnerabilities is another challenge, that was tackled in [Vas+21]. In this paper, the authors present a tool called Blade, designed to automatically repair vulnerable programs. They use a similar speculative constant-time policy, and a notion of speculation leakage modeling the effects of speculative execution. Their approach is based on the insertion of annotations called *protect* in the source code, allowing to prevent the speculation in specific parts of the code. These annotations may then be implemented as memory fence instructions at assembly level, to prevent the speculation mechanism. A type-system also ensures that a minimal amount of annotations (to limit the performance overhead) are correctly inserted in a program.

As future work, we wish to consider the problem of formally verified secure-compilation of speculative constant-time programs. The problem of secure-compilation of speculative constant-time programs is also being considered in the case of the Jasmin compiler, as detailed in [Bar+21a], although it is not formally verified. In particular, designing program transformations that take advantage on syntactic annotations such as the protect annotations seems like a promising idea. At first, we wish to explore these ideas on a small prototype, such as a small language equipped with a speculative semantics, then study the preservation of speculative constant-time during standard optimization transformations. In the longer term, we hope to integrate these ideas into a realistic compiler such as CompCert.

Author's contributions

[Bar+19] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu, « Formal verification of a constant-time preserving C compiler », *in*: *Proceedings of the ACM on Programming Languages* 4.*POPL* (2019), pp. 1–30.

[Bar+21b] Gilles Barthe, Sandrine Blazy, Rémi Hutin, and David Pichardie, « Secure Compilation of Constant-Resource Programs », *in*: *IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021.

Bibliography

[19] *The Coq proof assistant reference manual*, Version 8.9.1, Inria, 2019, URL: http://coq.inria.fr.

[Aga00] Johan Agat, « Transforming out Timing Leaks », *in*: *POPL*, ACM, 2000, pp. 40–53.

[Alm+16] José Bacelar Almeida et al., « Verifying Constant-Time Implementations », *in*: *25th USENIX Security Symposium, USENIX Security 16*, 2016.

[Alm+17] José Bacelar Almeida et al., « Jasmin: High-Assurance and High-Speed Cryptography », *in*: *CCS*, ACM, 2017, pp. 1807–1823.

[Alm+19] José Bacelar Almeida et al., « The Last Mile: High-Assurance and High-Speed Cryptographic Implementations », *in*: *CoRR* abs/1904.04606 (2019), arXiv: 1904.04606, URL: http://arxiv.org/abs/1904.04606.

[AP16] Martin R. Albrecht and Kenneth G. Paterson, « Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS », *in*: *Advances in Cryptology - EUROCRYPT*, vol. 9665, LNCS, Springer, 2016, pp. 622–643.

[ARM16] ARM, *mbed TLS*, 2016, URL: https://tls.mbed.org/.

[Ath+18] Konstantinos Athanasiou et al., « SideTrail: Verifying Time-Balancing of Cryptosystems », *in*: *Verified Software, Theories, Tools, and Experiments (VSTTE)*, Springer, 2018, pp. 215–228.

[Aya16] Luis Ayala, « Cybersecurity for hospitals and healthcare facilities », *in*: *Berkeley, CA* (2016).

[Bar+14] Gilles Barthe et al., « System-level non-interference for constant-time cryptography », *in*: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1267–1279.

[Bar+19] Gilles Barthe et al., « Formal verification of a constant-time preserving C compiler », *in*: *Proceedings of the ACM on Programming Languages* 4.*POPL* (2019), pp. 1–30.

[Bar+21a] Gilles Barthe et al., « High-Assurance Cryptography in the Spectre Era », *in*: (2021).

[Bar+21b] Gilles Barthe et al., « Secure Compilation of Constant-Resource Programs », *in*: *IEEE 34th Computer Security Foundations Symposium (CSF)*, 2021.

[BDJ19] Frédéric Besson, Alexandre Dang, and Thomas Jensen, « Information-flow preservation in compiler optimisations », *in*: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, IEEE, 2019, pp. 230–23012.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy, « Formal verification of a C compiler front-end », *in*: *International Symposium on Formal Methods*, Springer, 2006, pp. 460–475.

[Ben01] Mordechai Ben-Ari, « The bug that destroyed a rocket », *in*: *ACM SIGCSE Bulletin* 33.*2* (2001), pp. 58–59.

[Ber05] Daniel J Bernstein, *Cache-timing attacks on AES*, 2005, URL: `http://cr.yp.to/papers.html`.

[Ber06] Daniel J Bernstein, « Curve25519: new Diffie-Hellman speed records », *in*: *International Workshop on Public Key Cryptography*, Springer, 2006, pp. 207–228.

[BGL18] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte, « Secure Compilation of Side-Channel Countermeasures: the Case of Cryptographic "Constant-Time" », *in*: *Computer Security Foundations Symp. (CSF)*, IEEE, 2018, pp. 328–343.

[BL09] Sandrine Blazy and Xavier Leroy, « Mechanized semantics for the Clight subset of the C language », *in*: *Journal of Automated Reasoning* 43.*3* (2009), pp. 263–288.

[BLS12] Daniel J Bernstein, Tanja Lange, and Peter Schwabe, « The security impact of a new cryptographic library », *in*: *International Conference on Cryptology and Information Security in Latin America*, Springer, 2012, pp. 159–176.

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk, « A certified lightweight non-interference java bytecode verifier », *in*: *European Symposium on Programming*, Springer, 2007, pp. 125–140.

[BPT19] Sandrine Blazy, David Pichardie, and Alix Trieu, « Verifying constant-time implementations by abstract interpretation », *in*: *Journal of Computer Security* 27.*1* (2019), pp. 137–163.

[Cau+19] Sunjay Cauligi et al., « FaCT: a DSL for Timing-Sensitive Computation », *in*: *PLDI*, 2019, pp. 174–189.

[Cau+20] Sunjay Cauligi et al., « Constant-time foundations for the new spectre era », *in*: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 913–926.

[Çiç+17] Ezgi Çiçek et al., « Relational Cost Analysis », *in*: *ACM SIGPLAN Notices* 52.*1* (2017), pp. 316–329.

[Coh+21] Albert Cohen et al., « Secure Optimization Through Opaque Observations », *in*: (2021).

[DBR21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk, « Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE », *in*: NDSS, 2021.

[Den76] Dorothy E Denning, « A lattice model of secure information flow », *in*: *Communications of the ACM* 19.*5* (1976), pp. 236–243.

[DP10] Dominique Devriese and Frank Piessens, « Noninterference through secure multi-execution », *in*: *2010 IEEE Symposium on Security and Privacy*, IEEE, 2010, pp. 109–124.

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song, « The Correctness-Security Gap in Compiler Optimization », *in*: *Symp. on Security and Privacy Workshops*, IEEE, 2015, pp. 73–87.

[DS07] David Delmas and Jean Souyris, « Astrée: From research to industry », *in*: *International Static Analysis Symposium*, Springer, 2007, pp. 437–451.

[GM82] Joseph A Goguen and José Meseguer, « Security policies and security models », *in*: *1982 IEEE Symposium on Security and Privacy*, IEEE, 1982, pp. 11–11.

[GM84] Joseph A Goguen and José Meseguer, « Unwinding and inference control », *in*: *1984 IEEE Symposium on Security and Privacy*, IEEE, 1984, pp. 75–75.

[HAH11] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann, « Multivariate Amortized Resource Analysis », *in*: *POPL*, 2011, pp. 357–370.

[HS06] Sebastian Hunt and David Sands, « On flow-sensitive security types », *in*: *ACM SIGPLAN Notices* 41.*1* (2006), pp. 79–90.

[Jou+15] Jacques-Henri Jourdan et al., « A formally-verified C static analyzer », *in*: *ACM SIGPLAN Notices* 50.*1* (2015), pp. 247–259.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun, « Differential power analysis », *in*: *Annual international cryptology conference*, Springer, 1999, pp. 388–397.

[KMM13] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, *Computer-aided reasoning: ACL2 case studies*, vol. 4, Springer Science & Business Media, 2013.

[Koc+11] Paul Kocher et al., « Introduction to differential power analysis », *in*: *Journal of Cryptographic Engineering* 1.*1* (2011), pp. 5–27.

[Koc+19] Paul Kocher et al., « Spectre attacks: Exploiting speculative execution », *in*: *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[Koc96] Paul C Kocher, « Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems », *in*: *Annual International Cryptology Conference*, Springer, 1996, pp. 104–113.

[Koo14] Phil Koopman, « A case study of Toyota unintended acceleration and software safety », *in*: *Presentation. Sept* (2014).

[Kum+14] Ramana Kumar et al., « CakeML: a verified implementation of ML », *in*: *POPL*, ACM, 2014, pp. 179–192.

[KWH11] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf, « Timing- and termination-sensitive secure information flow: Exploring a new approach », *in*: *2011 IEEE Symposium on Security and Privacy*, IEEE, 2011, pp. 413–428.

[Lan15] Adam Langley, *curve25519-donna*, 2015, URL: `https://code.google.com/archive/p/curve25519-donna`.

[Ler+12] Xavier Leroy et al., « The CompCert verified compiler », *in*: *Documentation and user's manual. INRIA* 53 (2012).

[Ler09] Xavier Leroy, « A formally verified compiler back-end », *in*: *Journal of Automated Reasoning* 43.*4* (2009), pp. 363–446.

[Mur+16] Toby Murray et al., « Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference », *in*: *Computer Security Foundations Symp. (CSF)*, IEEE, 2016, pp. 417–431.

[Ngo+17] Van Chan Ngo et al., « Verifying and synthesizing constant-resource implementations with types », *in*: *Symp. on Security and Privacy (SP)*, IEEE, 2017, pp. 710–728.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*, vol. 2283, Springer Science & Business Media, 2002.

[Ope19] OpenSSL, *OpenSSL*, 2019, URL: `https://www.openssl.org/`.

[PG21] Marco Patrignani and Marco Guarnieri, « Exorcising spectres with secure compilers », *in*: (2021), pp. 445–461.

[Pla02] Strategic Planning, « The economic impacts of inadequate infrastructure for software testing », *in*: *National Institute of Standards and Technology* (2002).

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede, « Dude, is my code constant time? », *in*: *Design, Automation & Test in Europe Conf. (DATE)*, IEEE, 2017, pp. 1697–1702.

[Ric53] Henry Gordon Rice, « Classes of recursively enumerable sets and their decision problems », *in*: *Transactions of the American Mathematical Society* 74.*2* (1953), pp. 358–366.

[SM19] Robert Sison and Toby Murray, « Verifying that a compiler preserves concurrent value-dependent information-flow security », *in*: *arXiv preprint arXiv: 1907.00713* (2019).

[Vas+21] Marco Vassena et al., « Automatically eliminating speculative leaks from cryptographic code with blade », *in*: *Proceedings of the ACM on Programming Languages* 5.*POPL* (2021), pp. 1–30.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith, « A sound type system for secure flow analysis », *in*: *Journal of computer security* 4.*2-3* (1996), pp. 167–187.

[Wan+19] Guanhua Wang et al., « oo7: Low-overhead defense against spectre attacks via program analysis », *in*: *IEEE Transactions on Software Engineering* (2019).

[WN94] David J Wheeler and Roger M Needham, « TEA, a tiny encryption algorithm », *in*: *International Workshop on Fast Software Encryption*, Springer, 1994, pp. 363–366.

[Wu+18] Meng Wu et al., « Eliminating Timing Side-Channel Leaks using Program Repair », *in*: *International Symp. on Software Testing and Analysis*, 2018, pp. 15–26.

[Yan+11] Xuejun Yang et al., « Finding and understanding bugs in C compilers », *in*: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger, « CacheBleed: a timing attack on OpenSSL constant-time RSA », *in*: *Journal of Cryptographic Engineering* 7.*2* (2017), pp. 99–112.

[Zha+12] Jianzhou Zhao et al., « Formalizing the LLVM intermediate representation for verified program transformations », *in*: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 427–440.

[Zin+17] Jean Karim Zinzindohoué et al., « HACL*: A Verified Modern Cryptographic Library », *in*: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, ed. by Bhavani M. Thuraisingham et al., ACM, 2017, pp. 1789–1806, DOI: 10.1145/3133956.3134043, URL: https://doi.org/10.1145/3133956.3134043.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

école normale supérieure rennes

Titre : Compilation vérifiée et sécurisée contre les canaux cachés temporels

Mot clés : Vérification formelle, Compilation sécurisée, Canaux Cachés, Coq, CompCert

Résumé : Notre société est de plus en plus dépendante des systèmes informatiques. Assurer leur sécurité est essentiel pour éviter les conséquences dramatiques des attaques contre ces systèmes. Dans cette thèse, nous nous concentrons sur une classe de d'attaques appelée attaques par *canaux cachés temporels*. Nous étudions les protections existantes contre ces attaques, telles que les politiques *constant-time* et *constant-resource*, et nous nous concentrons sur leur interaction avec la compilation. La compilation est le processus de transformation d'un programme écrit par un humain dans un langage source, en code machine exécutable par un ordinateur. Notre objectif est de s'assurer que la compilation n'introduit aucune vulnérabilité dans le code compilé, par rapport aux politiques de sécurité auxquelles nous nous intéressons ; c'est ce qu'on appelle la *compilation sécurisée*. Notre travail s'appuie également sur des *méthodes formelles* pour donner des garanties formelles sur les résultats que nous présentons. Une de nos contributions s'appuie sur le compilateur formellement vérifié CompCert. Tous les résultats présentés dans cette thèse sont également vérifiés mécaniquement en utilisant l'assistant de preuve Coq.

Title: Verified Secure Compilation against Timing Side-Channels

Keywords: Formal Verification, Secure Compilation, Side-Channels, Coq, CompCert

Abstract: Our society is increasingly dependent on computer systems. Ensuring their security is essential to avoid the dramatic consequences of attacks against these systems. In this thesis, we focus on a class of attacks called *timing side-channel* attacks. We study existing protections against these attacks, such as the *constant-time* and the *constant-resource* policies, and focus on their interaction with compilation. Compilation is the process of transforming a program written by a human in a source language, into machine code executable by a computer. Our goal is to ensure that compilation does not introduce any vulnerability in the compiled code, with respect to the security policies we focus on; this is called *secure compilation*. Our work also relies on *formal methods* to give formal guarantees on the results we present. One of our contribution relies on the formally verified CompCert compiler. All the results presented in this thesis are mechanically verified using the Coq proof assistant.