Secure compilation against side channel attacks

Gautier Raimondi

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

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

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

Par
Gautier RAIMONDI

Secure compilation against side-channel attacks

Thèse présentée et soutenue à IRISA, le 18 Décembre 2023
Unité de recherche : IRISA

Rapporteurs avant soutenance :

M. Sylvain CONCHON Professeur, Université Paris-Sud
M. Stephan MERZ Directeur de Recherche, Inria Nancy

Composition du Jury :

Président :	Mme Sandrine BLAZY	Professeure, Université de Rennes
Examinateurs :	M. Sylvain CONCHON	Professeur, Université Paris-Sud
	M. Stephan MERZ	Directeur de Recherche, Inria Nancy
	M. Julien SIGNOLES	Chercheur, CEA LIST
Dir. de thèse :	M. Thomas JENSEN	Directeur de Recherche, Inria Rennes
Co-dir. de thèse :	M. Frédéric BESSON	Chargé de Recherche, Inria Rennes

Remerciements

Avant de rentrer dans le vif du sujet, je tiens à remercier les personnes sans qui ce document n'aurait jamais vu le jour. Tout d'abord, Thomas, pour m'avoir permis de réaliser cette thèse et m'avoir redirigé lorsque je m'égarais. Ensuite, Frédéric, dont l'accompagnement constant aussi bien sur le plan scientifique que méthodologique a été le véritable moteur de cette thèse.

Ma reconnaissance va aussi vers Sylvain Conchon et Stephan Merz pour avoir accepté de rapporter cette thèse, ainsi que Sandrine Blazy et Julien Signoles pour leur présence à la soutenance en tant que présidente et membre du jury. Je les remercie pour les remarques qu'ils ont pu formuler sur ce document et les travaux qu'il présente.

J'aimerais également remercier l'ensemble de l'équipe CELTIQUE/EPICURE pour leur accueil, et plus particulièrement ceux avec qui j'ai partagé un bureau : Victoire, Santiago Sara et Alexandre. Merci pour les discussions que nous avons pu avoir.

Je remercie ma mère, mon père, ainsi que ma sœur et mes frères pour leur soutien, bien qu'ils n'aient pas attendu le début de cette thèse pour cela. De la même façon, merci beaucoup à Gaël, Paul, Mathilde et Romain, qui ont toujours été disponibles pour servir de canards ou pour me permettre une évasion temporaire.

Enfin, je remercie Ana, pour qui la période de rédaction a été aussi éprouvante que pour moi, et qui n'a pas cessé de me soutenir.

Résumé en Français

Que l'on parle d'une fusée, d'un smartphone, d'un pacemaker ou même d'un frigo, tous ont un point commun : un ordinateur. Pas forcément un ordinateur au sens usuel du terme, avec un clavier, une souris et un écran, mais un appareil exécutant du code. Seulement, ce code étant pensé, théorisé, puis écrit par un humain, il n'est pas infaillible. On y retrouve ce que l'on appelle des *bugs*, c'est-à-dire des erreurs de conception et/ou de réalisation. Ces bugs peuvent prendre de multiples formes. On peut par exemple citer le dépassement d'entier qui a causé l'explosion de la fusée Ariane V en 1996 [Ben01] ou l'implémentation imprécise de la logique dans les machines de radiothérapie Therac-25 [Lev95], provoquant des surdoses de radiation dans les années 1980 et causant la mort d'au moins 5 personnes. Un autre type de bug, cette fois plus innocent, est un oubli dans la gestion des tailles de messages provoquant un crash des iPhones lors de l'affichage de la notification de réception de certains messages contenant un savant mélange d'anglais et d'arabe [Tho15]. Bien que fondamentalement différents en termes de dangerosité et d'impact économiques, ces trois exemples sont des bugs dits *fonctionnels* : ils ont un impact direct sur le fonctionnement de l'outil ou du logiciel, qui ne se comporte pas comme prévu. Ainsi, la fusée explose, le Therac-25 envoie beaucoup trop de radiations et le téléphone crash. Il existe néanmoins un autre type de bug portant sur un domaine tout autre : les bugs de sécurité. Ceux-ci permettent à un agent extérieur, souvent mal intentionné, d'accéder soit à des données, soit à des outils auxquels il n'aurait normalement pas accès. Par exemple, il a été révélé en 2008 que les pacemakers étaient sensibles à des attaques radios [Hal+08].

Étant donné les enjeux importants soulevés par ces problématiques, une des solutions utilisées est celle des méthodes formelles. Ce sont des règles et techniques afin de raisonner de manière rigoureuse sur un programme, généralement dans le but de prouver une propriété. Pour faire le lien avec les bugs, on exige régulièrement d'un programme qu'il soit *sûr*, c'est-à-dire que toute exécution de ce programme se comporte exactement comme prédit. En particulier, cela signifie que l'exécution se termine proprement. Pour permettre le raisonnement sur un programme, ainsi que l'expression des propriétés désirées, on définit une *sémantique*, c'est-à-dire une spécification rigoureuse des opérations du programme et de leurs effets. Cela permet d'exprimer les propriétés dans le même

cadre que le comportement du programme lui-même, sans devoir se reposer sur le langage naturel, qui est sujet à interprétation.

Avec une spécification précise de la propriété et du programme, l'utilisation de méthodes formelles consiste à démontrer rigoureusement que le programme satisfait (ou non) la propriété. Les méthodes utilisées pour cela sont le typage, à savoir se limiter à certains programmes qui vérifient certaines contraintes, le model checking, qui analyse toutes les exécutions possibles, et l'analyse statique, qui sur-approxime les états accessibles par le système. Pour ce faire, on raisonne souvent à haut niveau, par exemple au niveau du programme source. Néanmoins, les propriétés à ce niveau ne sont que rarement d'intérêt, puisque l'on préfère obtenir des garanties sur le code compilé et exécutable directement. Il est tout à fait possible de raisonner directement sur le code machine, au plus bas niveau possible, mais cela signifie sacrifier les multiples simplifications offertes par le langage de programmation. Ainsi, il est souvent préférable de prouver la propriété à haut niveau, puis montrer que cette propriété est préservée par les différentes transformations (comme la compilation). Cette thèse traite des transformations de programmes de haut niveau, mais repose sur des compilateurs vérifiés pour préserver certaines propriétés le long de la compilation.

Un compilateur, vérifié ou non, est un programme prenant en entrée un programme écrit en un langage source et renvoyant un programme équivalent écrit en un langage cible. Par exemple, le compilateur GCC traduit un programme écrit en C en code assembleur. On attend généralement d'un compilateur deux choses : produire un programme équivalent au programme source et rendre ce produit aussi efficace que possible. À cette fin, les compilateurs utilisent des optimisations et sont donc de larges et complexes logiciels. Puisque ce sont eux-mêmes des programmes, les compilateurs ne sont pas exempts de bugs. Ainsi, une étude empirique de 2016 [Sun+16] a montré que GCC et LLVM, les deux plus gros compilateurs de C sur le marché, souffrent régulièrement de bugs, malgré leur utilisation quotidienne. La majorité du temps, ces bugs sont sur des cas spécifiques et négligeables par rapport aux bugs présents dans les programmes sources. Néanmoins, dans le cas de programmes utilisés dans des domaines tels que la santé ou l'aéronautique, ces risques supplémentaires sont de trop. Ainsi, s'est développée la compilation formellement vérifiée, qui joint à un compilateur une preuve de correction. Cela garantit que le comportement du programme cible est identique à celui du programme source, dans les sémantiques respectives des langages source et cible. Immédiatement, cela induit que les garanties de sûreté sont conservées. Plusieurs projets de compilateur vérifié ont vu le jour,

parmi lesquels CakeML [Kum+14], qui compile du ML vers de l'assembleur, et CompCert [Ler09], qui raisonne sur un sous-ensemble de C99. Dans cette thèse, nous utilisons Jasmin [Alm+17] afin de préserver les propriétés à la compilation.

Jasmin est un cadre de programmation, c'est-à-dire à la fois un langage et un compilateur de vérifié. Le langage est hybride, pensé pour les primitives cryptographiques, permettant à l'utilisateur d'écrire du code en utilisant des mécanismes de bas niveau (registres, instructions x86...), tout en permettant l'utilisation d'abstractions de haut niveau (boucles, variables, fonctions...). De plus, Jasmin garantit la préservation sémantique du programme et est interfacé avec EasyCrypt [Bar+13]. Cela permet la construction automatique de preuves de préservation pour des propriétés de sécurité. En plus de cette interface avec EasyCrypt, Jasmin garantit la préservation d'une certaine propriété : *constant-time* [Alm+16 ; Amm+22].

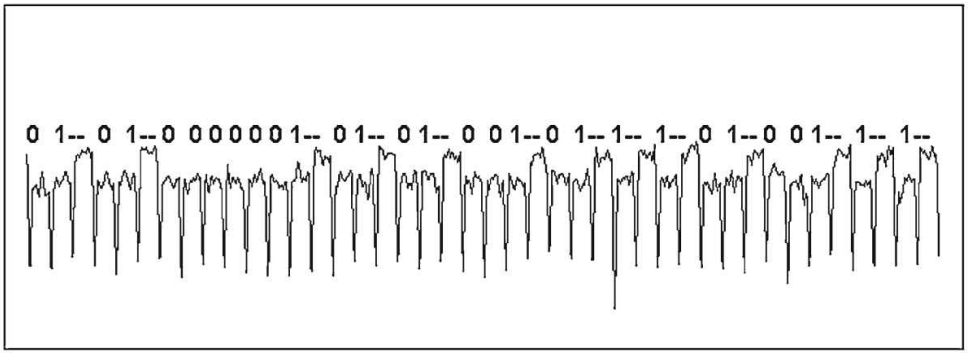


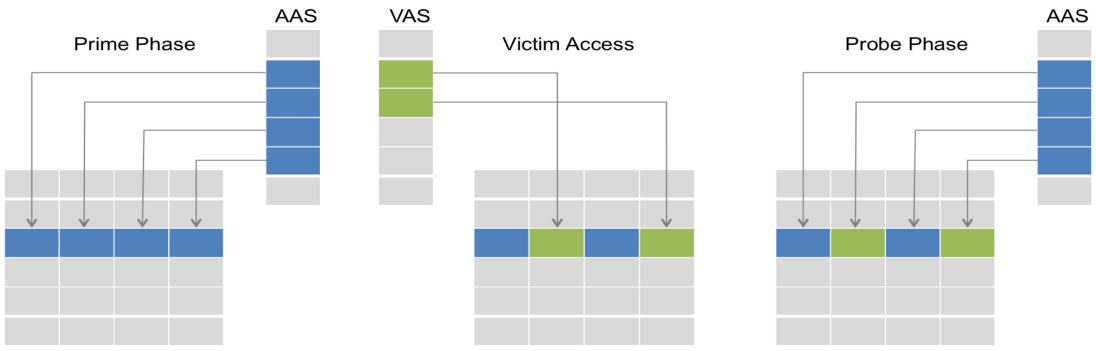
Figure 1 – Analyse de Puissance de [Koc+11]

Afin de définir ce qu'est constant-time, il convient de commencer par définir le type d'attaque que cette propriété prévient : les attaques par canaux auxiliaires. Une attaque par canaux auxiliaires est l'exploitation de la réalité physique des systèmes informatiques. Si un algorithme, ou une primitive cryptographique, est pensé et programmé dans un monde mathématique abstrait, où certaines propriétés peuvent *facilement* être garanties, son implémentation est différente. En fonction de l'exécution, un appareil ne va pas dégager la même chaleur, produire le même son, prendre le même temps ou consommer le même montant d'énergie. Ainsi, il est possible de déduire des informations sur l'exécution en mesurant ces réalités physiques. Par exemple, comme détaillé dans [Koc+11], si on s'intéresse à la consommation électrique d'une puce calculant l'exponentiation modulaire de RSA [RSA78], on obtient une courbe comme celle présentée dans la Figure 1. L'intuition derrière l'exponentiation modulaire est d'itérer sur les bits de la clé secrète, et

de ne réaliser la multiplication que si ce bit est vrai. Lorsque l'on regarde la courbe de consommation, on peut remarquer des pics et des creux, correspondant respectivement aux 1 et aux 0 de la clé, étant donné que la multiplication demande plus de puissance.

Comme mentionné précédemment, on peut aussi regarder le temps d'exécution et essayer d'en déduire des informations [Koc96]. Ces attaques sont connues comme des attaques temporelles et sont réalisables en pratique, même à travers un réseau [BB05 ; Ber05]. En utilisant l'exemple de l'exponentiation modulaire, la multiplication prendrait plus de temps que de ne rien faire, et on sait donc que le temps pris par l'exécution est directement proportionnel au nombre de bits à 1 dans la clé. Cela simplifierait grandement le nombre de clés à tester pour un attaquant. Cette thèse se concentre sur ce type d'attaque puisqu'elles sont à la fois les plus courantes et les plus dangereuses, ne nécessitant pas d'accès physique au système.

Heureusement, il existe des façons de se prévenir contre ces canaux auxiliaires. L'une d'entre elles, la politique de programmation constant-time, requiert que les programmes suivent des règles strictes pour être sécurisés. Ces règles sont : aucun contrôle de flux dépendant des secrets et aucun accès à la mémoire dépendant des secrets. Notons que le nom constant-time n'est pas suffisamment précis. La politique n'impose pas qu'un programme s'exécute toujours en un temps constant, indépendamment de tout facteur, mais seulement que ce temps d'exécution soit indépendant des secrets. La raison pour interdire un contrôle de flux dépendant des secrets est assez simple : s'il y a une condition sur un secret, les deux branches ne prennent pas nécessairement le même temps pour s'exécuter et la valeur du secret utilisé peut être déduite par le temps d'exécution. Pour les accès mémoire, cette restriction est due à l'existence d'attaques sur le cache.



(a) Attacker's Prime phase (b) Victim's memory accesses (c) Attacker's Probe phase

FIGURE 2 – Illustration de l'attaque sur le cache Prime+Probe [Mus+20]

Afin d'illustrer une telle attaque sur le cache, considérons la Figure 2 représentant

le cache d'une machine partagée par un programme et l'attaquant à différents instants. Lorsqu'il en gagne le contrôle, l'attaquant peut réaliser des accès mémoire, remplaçant ainsi dans le cache des lignes choisies (étape de *prime*). Après avoir retourné la priorité au programme victime puis l'avoir laissé tourner, l'attaquant peut à nouveau accéder la mémoire. En fonction du temps requis pour accéder au contenu de la mémoire, il obtient des informations sur la victime (étape de *probe*). Si l'accès mémoire est lent, l'information recherchée n'est plus en cache et le programme attaqué a accédé à une information contenue dans cette ligne. Dans le cas où, par contre, l'accès est rapide, cela n'offre aucune information à l'attaquant, si ce n'est que le programme attaqué n'a pas réalisé d'accès sur cette ligne, sans forcément savoir pourquoi. Par exemple, AES [Dwo23] accède aux secrets directement. Une telle attaque révélerait que la valeur utilisée comme index est contenue dans un intervalle strictement inférieur à la taille du tableau.

Pour illustrer la politique constant-time, la Figure 3 montre trois programmes. Le premier représente une conditionnelle sur une variable de boucle et est constant-time, puisqu'il représente un branchement sur une variable publique. Les deux derniers sont respectivement un branchement secret et un accès mémoire secret, et ne sont donc pas constant-time.

```
...
if i == 0 then
    ...
else
    ...
```

(a) Conditionnelle constant-time

```
if secret then
    ...
else
    ...
```

(b) Conditionnelle non constant-time

```
x = t[secret];
```

(c) Accès mémoire secret

FIGURE 3 – Exemples de programmes (non) constant-time

Comme proposé précédemment, la politique constant-time n'est pas la seule contre-mesure existante contre ces attaques. Par exemple, Agat [Aga00] a essayé d'équilibrer les deux branches en ajoutant de fausses instructions. Dans [KM07], Köpf et Mantel montrent qu'unifier les branches peut réduire le nombre de fausses instructions nécessaires. Cependant, [ZS18] montre qu'en combinant des attaques temporelles et de puissance, les fausses instructions peuvent être reconnues et le réel temps d'exécution déduit.

Pour garantir qu'un programme respecte la politique constant-time, plusieurs méthodes peuvent être choisies. Si JASMIN, et d'autres [Bar+20a], préfèrent garantir la

préservation de la propriété, il reste qu'écrire du code constant-time est dur. Écrire du code cryptographique est généralement source d'erreur, mais cela devient pire lorsque des contraintes sont ajoutées sur la forme (pas de conditionnelles secrètes) ou sur le fond (pas d'accès secrets). Des outils tels que FaCT [Cau+17] visent à simplifier l'écriture de primitives cryptographiques constant-time. Le langage propose des constructions de haut niveau, qui sont transformées durant la compilation. FaCT est vérifié pour produire du code constant-time, mais utilise aussi l'outil dudect [RBV16] pour vérifier *à posteriori* que les optimisations réalisées par le compilateur LLVM préservent la propriété. D'autres langages ont été créés, tel que VALE [Bon+17], un langage assembleur de haut niveau qui vérifie des primitives cryptographiques en utilisant DAFNY [Lei10] ou encore F*[Swa+16], et prouve la propriété constant-time en utilisant l'analyse teintée de F*. De façon similaire, HACL* [Zin+17] est une librairie cryptographique vérifiée, programmée et prouvée en F*, et compilée avec le compilateur vérifié Kremlin [Pro+17] qui préserve la propriété constant-time. SC-eliminator [Wu+18] utilise l'analyse teintée pour détecter et réparer les fuites en place, c'est-à-dire directement dans le code source. Pour ce faire, la transformation utilise une suppression de branches similaire à FaCT. Cette approche a été améliorée par Soares et Pereira [SP21] pour éviter de générer des programmes avec des accès en dehors des bornes. Dans [BPT19], l'interprétation abstraite est utilisée afin de vérifier la propriété constant-time au niveau source. D'autre solutions [Bar+14 ; Bar+20b] préfèrent vérifier après compilation que le code machine respecte bien la politique.

Contributions et organisation du document Dans cette thèse, nous présentons une solution à la production de code constant-time par compilation sécurisée. En particulier, nous abordons trois questions :

1. Comment transformer un programme en équivalent constant-time ?
2. Est-ce que tous les programmes peuvent être rendus constant-time ?
3. Quel est le coût d'une telle transformation ?

Nous répondrons à ces questions au long de ce document. La première de ces questions a découlé en la définition de plusieurs passes de transformations qui, lorsque appliquées dans le bon ordre, garantissent que le programme résultant est constant-time.

La seconde question requiert la définition d'un système de type et de preuves qu'être accepté par ce système de type garantit que la version transformée du programme est constant-time.

La troisième et dernière question demande de l'expérimentation pour être répondue. Si le travail réalisé sur les deux premières questions a été formalisé sur un petit langage, nous avons implémenté notre transformation dans le compilateur Jasmin pour observer son fonctionnement.

Les deux premières questions ont été le sujet d'un travail préliminaire présenté au 6^{th} Workshop on Principles of Secure Compilation (PriSC') début 2022. Une version plus aboutie de ce même travail, contenant des résultats expérimentaux, a été publiée au 25^{th} Symposium on Principles and Practice of Declarative Programming (PPDP) dans [BJR23].

Le reste de cette thèse s'articule comme suit. Dans le Chapitre 1, nous présentons notre petit langage théorique, ainsi qu'une définition formelle de la propriété constant-time. Ce premier chapitre contient également une explication détaillée de la plus simple des transformations de programme vers constant-time. Dans le Chapitre 2, nous présentons nos transformations de programme, d'abord de façon informelle avant de formaliser leur définition et prouver leur propriété de préservation sémantique. Nous introduisons aussi ici quelques informations qui guident la transformation et seront fournies par le système de type. Les contributions de ce chapitre sont principalement le mécanisme de *scope-increase*, réalisant du déplacement d'instructions problématiques, et la version améliorée de l'usuel *branch removal*. Le Chapitre 3 contient la définition de notre système de type différenciant les flux directs et indirects, et montre le lien entre ce système et le système classique pour constant-time. Dans le chapitre 4, nous montrons la sécurité de notre transformation. Nous revenons sur l'implémentation de la transformation et sur les expériences utilisant le compilateur Jasmin dans le Chapitre 5. Enfin, nous concluons cette thèse et présentons des perspectives sur nos contributions.

Table of Contents

List of Definitions

List of Theorems

List of Lemmas

List of Programs

INTRODUCTION

Whether we're talking about a rocket, a smartphone, a pacemaker or even a fridge, they all have one thing in common: a computer. Not necessarily a computer in the usual sense of the word, with a keyboard, a mouse, and a screen, but a device that executes code. However, this code, conceived, theorized, and written by a human, is not unerring. It contains what is known as *bugs i.e.*, errors in design and/or implementation. These bugs take many different forms. The integer overflow that caused the Ariane V rocket to explode in 1996 [Ben01] or the imprecise implementation of logic in the Therac-25 radiotherapy machines [Lev95] that caused radiation overdoses resulting in at least 5 deaths in the 80s are two such forms... Another one, perhaps more innocuous, would be the case of an oversight in the management of message sizes causing iPhones to crash when displaying notifications of receipt of a certain message combining English and Arabic [Tho15]. Regardless of the difference in cost and danger, these three examples are functional bugs: they have a direct impact on the operation of the tool or software, which doesn't behave as expected: the rocket explodes, the Therac-25 sends out far too much radiation, and the phone crashes. There is, however, another type of bug in a completely different field: security bugs. These enable an external agent, usually with malicious intent, to access data or tools to which it would not normally have access. For example, it was revealed in 2008 that pacemakers were susceptible to radio attacks [Hal+08].

Given the high stakes raised by these issues, one of the solutions used is formal methods. These are rules and techniques for rigorously reasoning about a program, usually to prove a property. To make the connection with bugs, a program is often required to be safe *i.e.*, any execution of the program must behave exactly as predicted. In particular, this means that execution must finish cleanly. To enable reasoning about a program, as well as the expression of desired properties, we define a semantics *i.e.*, a rigorous specification of the program's operations and their effects. This allows properties to be expressed in the same framework as the program's behavior itself, without having to use natural language, which is open to interpretation.

With a precise specification of the property and the program, the use of formal methods consists of rigorously demonstrating that the program satisfies (or not) the property. The

methods used for this are typing *i.e.*, limiting oneself to certain programs that verify certain constraints, model checking, which analyzes all possible executions, and static analysis, which over-approximates the states accessible by the system. To do this, we often reason at a high level *i.e.*, on the source program. However, source-level properties are rarely of interest, as we mainly wish to have guarantees on executable, compiled programs. It is perfectly possible to reason directly on machine code, at the lowest possible level, but this means sacrificing all the simplifications offered by programming languages. Thus, it is often preferable to prove the property at a high level and that it is preserved by the transformation (*i.e.*, compilation). This thesis deals with high-level program transformations but relies on verified compilers to preserve certain properties through all compilation passes.

A compiler, whether verified or not, is a program that takes a program written in a source language and returns an equivalent program written in a target language. For example, the GCC compiler translates a program written in C into assembly code. A compiler is usually expected to do two things: produce a program equivalent to the source program, and make this output program as efficient as possible. To achieve this, compilers use optimizations and are therefore complex pieces of software. Since they are also programs, they are not free of bugs. For example, a 2016 empirical study [Sun+16] showed that GCC and LLVM, the two biggest C compilers on the market, often have bug reports, even though they are used in production and therefore tested. Generally, these bugs are on niche cases and negligible compared to the number of bugs in the source programs. However, when it comes to major issues such as health or aeronautics, these additional risks are too much. This has led to the development of formally verified compilation, which go with a compiler with a proof of correctness. This guarantees that the behavior of source and compiled programs is the same, in the respective semantics of the source and target languages. This immediately gives us that safety guarantees are preserved. Several verified compiler projects have emerged, including CakeML [Kum+14], which compiles from ML to assembler, and CompCert [Ler09], which reasons on a subset of C99. In this thesis, we use Jasmin [Alm+17] to preserve properties during compilation.

Jasmin is a programming framework *i.e.*, both a language and a verified compiler. The language is a hybrid, designed for cryptographic primitives, allowing the user to write code using low-level mechanisms (registers, x86 instructions...), while allowing the use of higher-level abstractions (loops, variables, functions...). In addition, Jasmin guarantees semantic preservation of the program and is interfaced with EasyCrypt [Bar+13]. This

enables it to automatically construct preservation proofs for security properties. There is, however, one type of property for which Jasmin guarantees preservation: constant-time [Alm+16; Amm+22].

To define what constant-time is, we need to start by defining the type of attack this property prevents: side-channel attacks. A side-channel attack is the exploitation of the physical reality of computer systems. While an algorithm, or cryptographic primitive, is imagined and programmed in an abstract, mathematical world, where certain properties are guaranteed, its implementation is quite different. Depending on the execution, a device will not give off the same heat, the same sound volume, take the same amount of time, or consume the same amount of power. Thus, it is possible to deduce information about execution by observing these physical measurements. For example, as detailed in [Koc+11], if we look at the power consumption of a chip computing the RSA [RSA78] modular exponentiation loop, we obtain a curve shown in Figure 1. The idea behind modular exponentiation is to iterate over the bits of the secret key, multiplying only if that bit is true. When we look at the power consumption curve, we can see peaks and troughs, corresponding respectively to the 1 and 0 bits of the key, as multiplication requires more resources.

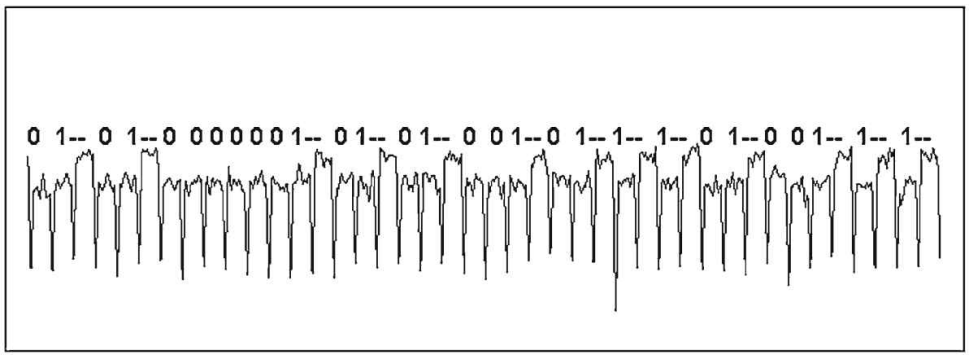
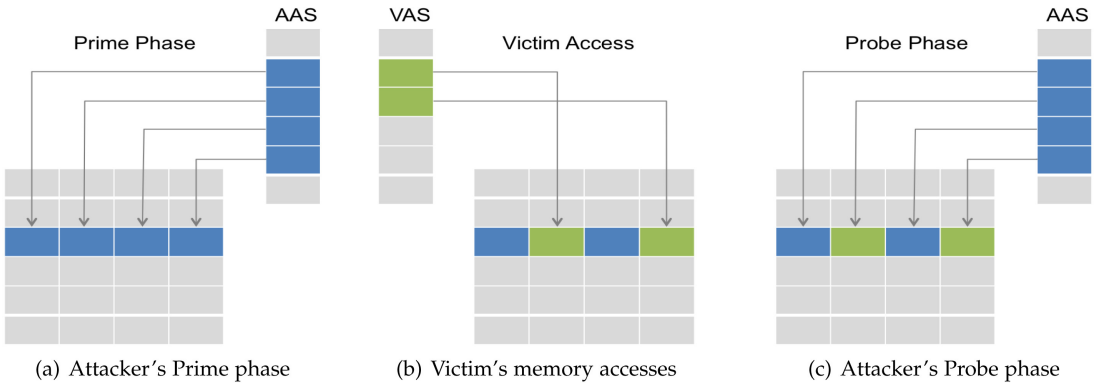


Figure 1 – Power Analysis from [Koc+11]

As mentioned above, we can also look at the execution time to try and deduce information [Koc96]. Such attacks are known as timing attacks and are in practice feasible even remotely over the network [BB05; Ber05]. Using the example of modular exponentiation, multiplication takes longer than doing nothing, and we know that the time taken is directly proportional to the number of bits set to 1 in the key. This greatly reduces the number of keys to be tested. This thesis focuses on this type of attack since they are the most common and the most dangerous, that can be carried out remotely.

Fortunately, there are ways of getting rid of this side-channel. One of them, the constant-time development policy, requires programs to follow strict rules to be secure. These rules are: no secret-dependent control flow and no secret-dependent memory access. Note that the name constant-time is not precise enough. The policy does not require that a program always run in a constant time, independent of any factor, but only that this execution time be independent of the various secrets. Why prohibiting a secret-dependent control flow is quite simple: if there is a condition on a secret, the two branches do not take the same time to execute, and the value of the secret can be deduced using the execution time. For memory accesses, on the other hand, this is due to the existence of attacks on the cache.



(a) Attacker's Prime phase (b) Victim's memory accesses (c) Attacker's Probe phase

Figure 2 – Illustration of the Prime+Probe Cache Attack [Mus+20]

To illustrate the attack, consider Figure 2 representing the cache of a machine shared by a program and an attacker at different times. When it gains control, an attacker can perform memory accesses, thus overwriting the cache by replacing lines (prime phase). After returning control to the victim program and letting it run for a given time, the attacker can access the memory again. Depending on the time required to access the memory content, he obtains information about the program under attack (probe phase). If the memory access is slow, this means that the information is no longer cached and that the attacked program has accessed the information contained in this line. If, on the other hand, the memory access is fast, this offers no information to the attacker, as the attacked program has simply made no access on this line, without being able to deduce why. For example, AES [Dwo23] accesses secrets directly. Such an attack would give the information that the value used as an index is contained in an interval strictly smaller than the size of the array.

To illustrate the constant-time policy, we show in Figure 3 three programs. The first

represents a conditional on a loop variable, and is constant-time, being a branching on a public variable. The last two are respectively a secret branch, and a secret memory access, and are therefore not constant-time.

```
...
if i == 0 then
    ...
else
    ...
```

(a) Constant-time conditional

```
if secret then
    ...
else
    ...
```

(b) Non constant-time conditional

```
x = t[secret];
```

(c) Secret dependent memory access

Figure 3 – Examples of (non) constant-time programs

As implied, the constant-time policy is not the only countermeasure against this kind of attack. For example, Agat [Aga00] have tried to balance the two branches by adding false instructions. In [KM07], Köpf and Mantel show that unifying the branches can reduce the number of false instructions needed. However, [ZS18] shows that by combining timing and power attacks, false instructions can be recognized and the execution time determined.

To ensure that a program respects the constant-time policy, several approaches are available. While Jasmin, and others [Bar+20a], guarantee property preservation, the fact remains that writing constant-time code is hard. Writing cryptographic code is generally error-prone, but it gets worse when constraints are added on syntax (no conditionals) or substance (no secret accesses). Tools such as FaCT [Cau+17] aim to simplify the writing of cryptographic primitives in constant time. The language offers high-level constructs, which are then transformed during compilation. FaCT is verified to produce constant-time code but also uses the dudect [RBV16] tool to check *a posteriori* that optimizations made by the LLVM compiler preserve the property. Other specific languages have been created, such as VALE [Bon+17], a high-level assembly language that checks cryptographic primitives using DAFNY [Lei10] or F*[Swa+16], and proves the constant-time property using F*-tainted analysis. Similarly, HACL*[Zin+17] is a verified cryptographic library programmed and proven in F*, and compiled with the Kremlin [Pro+17] verified compiler that preserves the constant-time property. SC-eliminator [Wu+18] uses tainted analysis to detect and repair leaks in place, i.e., directly in the source code. To achieve this, the transformation uses branch deletions similar to FaCT. This approach has been

improved by Soares and Pereira [SP21] to avoid generating programs with out-of-bounds accesses. In [BPT19], abstract interpretation is used to verify the constant-time property at source level. Other solutions [Bar+14; Bar+20b] prefer to check after compilation that the binary code complies directly with the policy.

Contributions and organization of this thesis In this thesis, we present a solution to the production of constant-time code by way of secured compilation. In particular, we tackle three questions:

1. How can we transform a program into a constant-time counterpart?
2. Are all programs equally *constant-time-able*?
3. What does it cost?

We answer those questions throughout this document. The first question has led to the definition of multiple passes of transformation that, when applied in the correct order, ensure that the resulting program follows the constant-time security policy.

The second question requires the definition of a type system and proofs that being accepted by the type system ensures that the transformed version of a program is constant-time.

The third and last question need some kind of experiments to be answered. While the work done on the first two questions has been formalized on a small language, we implemented our transformation within the Jasmin compiler as to observe its behavior.

The first two questions have been the scope of a preliminary work presented at the 6^{th} Workshop on Principles of Secure Compilation (PriSC') in early 2022. A more accomplished version of this same work, including some experiments, has been published at the 25^{th} Symposium on Principles and Practice of Declarative Programming (PPDP) in [BJR23].

The rest of the thesis answers to these questions as follows. In Chapter 1, we present our small, theoretical, language, as well as a formal definition of constant-time. This first chapter also contains a thorough explanation of the simplest program transformation for the constant-time policy. In Chapter 2, we present our program transformations, first in an informal way, before formalizing their definitions and proving their semantics preservation. We also introduce some information needs that will be fulfilled by the type system. The contribution in this chapter relies mostly on the *scope-increase* mechanism, performing code motion of problematic instructions, and on our updated version of the usual *branch removal*. In Chapter 3, we define our information flow tracking type system

discerning direct and indirect flows, and show how it relates to the usual information flow type system for constant-time. In Chapter 4, we prove our transformation's security. We report on the implementation of the transformation and experiments using the Jasmin compiler in Chapter 5. Finally, we conclude this thesis and present perspectives on our contributions.

Chapter 1

Background

1.1 Language

In this section, we define a simple language L by giving its syntax and semantics. This language will be used in all subsequent sections and chapters, and will be extended. We use a formal definition to allow us to write proofs on this language and associated transformations.

1.1.1 Syntax

We use an imperative language, manipulating integer constants, variables, and fixed-size arrays. The language contains the usual `if` and `for` instructions. Given that we aim to reason on crypto-focused code, the `for` loop tends to be more used than the `while` loop. The syntax is given in definition 1.

Definition 1: Syntax of L

Expressions:

$$\textit{expr} \ni e \quad ::= \quad x \mid c \mid e_1 \oplus e_2 \mid t[e]$$

Statements:

$$\begin{aligned} \textit{stmt} \ni s \quad ::= \quad & \textbf{skip} \\ & \mid x = e \\ & \mid t[e_1] = e_2 \\ & \mid s_1; s_2 \\ & \mid \textbf{if } x \textbf{ then } s_1 \textbf{ else } s_2 \\ & \mid \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s \end{aligned}$$

Language L is composed of expressions *expr* and statements *stmt*. An expression e may be a variable x, a constant integer c, a binary operator applied to arguments $e_1 \oplus e_2$ or an array access $t[e]$ where t is an array variable, and e a computed index. Arrays are statically assigned, and their size is known and is never modified during execution. A statement s may be a **skip** (equivalent to a *nop* operation), an assignment $x = e$ with x a variable, an array update $t[e_1] = e_2$ with t an array variable and e_1, e_2 two computed expressions, a sequence of two statements $s_1; s_2$, a conditional or a `for` loop.

Several syntactic constraints are also placed on language L:

1. Condition e of a conditional is restricted to be a variable, which is neither modified within the conditional nor after the conditional;
2. Memory accesses are only done in *simple* statements *i.e.*, either $y = t[x]$ or $t[x] = y$ where x and y are variables;
3. Loop bounds are known constants;
4. Loop index is not modified within the loop body.

The first and second constraints can be circumvented by adding a pre-processing to a broader language: we create new temporary variables such that the program satisfies the constraints. The constraints on loops, as well as the restriction on constant array size, are common for cryptographic code and can be found *e.g.*, in the input language of the Jasmin compiler [Alm+20], specifically designed for cryptography. These constraints offer some simplicity on the syntax, allowing for easier case reasoning when working on programs written in L. For example, the second constraint guarantees that we can think about memory accesses *locally*, without resorting to the syntax tree to check if the memory access holds another one within its index for example.

1.1.2 Semantics

Language L is given a big-step semantics, which is standard except that the execution also generates a trace of leakage.

This semantics is based on environments σ, and values v. In L, a value can either be an integer $n \in \mathbb{Z}$ or an array defined by its size and its content. Therefore, we define the set of values $\mathbb{V}$ as follows:

Definition 2: Set of values $\mathbb{V}$

$$\mathbb{V} \ni v \quad ::= \quad n \mid [m \mid n_1; \ldots; n_m]$$

If $t = [m \mid n_1; \ldots; n_m]$ is an array, we define $size(t)$ to be its size *i.e.*, m, and $t[i]$, with $i \in \mathbb{N}, i < m$ to be the value of index i *i.e.*, n_i.

Then, an environment σ is a mapping from variables x and arrays a to their values, $\mathbb{V}$. We define $\sigma[x]$ to be the value associated to x in σ, and $\sigma[x \mapsto v]$ to be the environment that is equal to σ, except for x which is now associated to v.

Our semantics generates a trace of leakage. A trace t is a list of values. We denote ϵ the empty list, and $\cdot$ the concatenation operation, both with other traces and with values. Following Barthe *et al.*[Bar+19], a trace is generated when evaluating a conditional statement or performing an array access. This leakage represents the information that could be leaked using a timing attack. Therefore, keeping trace of this leak allows us to check that a program follows the constant-time programming discipline. This can be intuited by the fact that we leak conditional guards and array indexes, the two sensitive characteristics in constant-time programming.

Semantics for expressions

The semantics of expressions is defined as a relation $(e, \sigma) \downarrow^t v$, where e is an expression, σ a mapping environment, t a leaking trace, and v the resulting value. It reads as e evaluated in σ results in v and produces the trace t. Its definition is given in Definition 3.

Definition 3: Leaky semantics for expressions

$$\frac{}{(c, \sigma) \downarrow^{\epsilon} c} \qquad \frac{\sigma[x] = v}{(x, \sigma) \downarrow^{\epsilon} v} \qquad \frac{\begin{array}{c}(e_i, \sigma) \downarrow^{t_i} v_i \quad i \in \{1, 2\} \\ v_3 = v_1 \oplus v_2\end{array}}{(e_1 \oplus e_2, \sigma) \downarrow^{t_1 \cdot t_2} v_3}$$

$$\frac{(e, \sigma) \downarrow^{t_e} i \quad 0 \leq i < size(t) \quad \sigma[t][i] = v}{(t[e], \sigma) \downarrow^{t_e \cdot i} v}$$

This semantics for expressions contains five rules. Evaluating a constant c produces the value c itself, independently of the environment σ, and does not produce any trace. Evaluating a variable x results in its value in the σ environment, and does not leak anything either. Evaluating a binary operation $\oplus$ between two expressions e_1 and e_2 requires to first evaluate e_1 and e_2 to v_1 and v_2. The binary operation $e_1 \oplus e_2$ then results in the operation between resulting values $v_1 \oplus v_2$. Similarly, the evaluation of e_1 and e_2 produces two traces t_1 and t_2. By supposing that all operations $\oplus$ are written as constant-time[1], the trace generated by $e_1 \oplus e_2$ is $t_1 \cdot t_2$. Lastly, the evaluation of an array access $t[e]$ requires first the evaluation of its index e to a value i. We also require that i is within the bounds of t, by asserting that $i \geq 0$ and $i < size(t)$, for safety. The generated trace of an array access is the trace of the evaluation of e, appended by the value of e, which is used as an address, and therefore leaked.

Semantics for instructions

The semantics of instructions is defined as a relation $(s, \sigma) \downarrow^t \sigma'$, where s is a statement, t a leaking trace, and σ, σ' two environments. It reads as s evaluated in σ results in the updated environment σ' and produces the trace t. It is defined as:

Definition 4: Leaky semantics for instructions

$$\frac{}{(\textbf{skip}, \sigma) \downarrow^{\epsilon} \sigma} \qquad \frac{(e, \sigma) \downarrow^{t} v}{(x = e, \sigma) \downarrow^{t} \sigma[x \mapsto v]} \qquad \frac{(s_1, \sigma_1) \downarrow^{t_1} \sigma_2 \quad (s_2, \sigma_2) \downarrow^{t_2} \sigma_3}{(s_1; s_2, \sigma_1) \downarrow^{t_1 \cdot t_2} \sigma_3}$$

$$\frac{(e_1, \sigma) \downarrow^{t_1} i \quad 0 \leq i < size(t) \quad (e_2, \sigma) \downarrow^{t_2} v}{(t[e_1] = e_2, \sigma) \downarrow^{t_1 \cdot t_2 \cdot i} \sigma[t \mapsto \sigma[t][i \mapsto v]]} \qquad \frac{(x, \sigma) \downarrow^{\epsilon} b \quad (s_b, \sigma) \downarrow^{t} \sigma'}{(\textbf{if } x \textbf{ then } s_{true} \textbf{ else } s_{false}, \sigma) \downarrow^{b \cdot t} \sigma'}$$

$$\frac{\forall_{i \in [c_1; c_2]} (x = i; s, \sigma_i) \downarrow^{t_i} \sigma_{i+1} \quad t = t_{c_1} \cdot \cdots \cdot t_{c_2}}{(\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s, \sigma_{c_1}) \downarrow^{t} \sigma_{c_2+1}}$$

This semantics contains six rules. Evaluating a skip, preserves the original environment, and does not leak anything. Evaluating an assignment needs first to evaluate the expression e, and results in an updated environment where x now has the value evaluated for e. The leaked trace is the information leaked when evaluating e. When evaluating a

1. This is for example usually not the case for the division.

sequence of two instructions, we first evaluate s_1 in our initial environment σ_1. This gives us a new environment σ_2, which we use to evaluate s_2. Similarly to the above, the trace is the concatenation of the two traces generated by the evaluation of s_1 and s_2. An array write is performed by first evaluating the value of the index, and checking it is within the array's bounds, just as an array read. The written expression is then evaluated, and the environment's value of the array is updated to now contain the value v at index i. The trace leaks the value of the index, as stated earlier. The evaluation of the conditional is standard: we first evaluate the value of the guard, and we execute the corresponding branch. To track the execution, we leak into the trace the value of the guard, along with the trace generated by the branch executed. Finally, `for` loop evaluations are straightforward. We evaluate $n = c_2 - c_1$ times the loop body, prepended by the assignment of the index value to the loop variable. We leak a trace concatenating the traces generated by each iteration.

1.1.3 Operators

To ease the use of this language and syntax, we craft several operators on statements and/or related objects.

Set of variables

We note $var(s)$ to be the set of variables either modifier or read by a statement s.

Set of modified variables

We note $mod(s)$ to be the set of variables modified by a statement s *i.e.*, all variables on the left side of an assignment.

Sequence of a set

Given a set of statements S, we can craft an arbitrary statement $seq(S)$ such that

$$seq(S) \in \{s_1; \ldots; s_n \mid \cup_i s_i = S \wedge n = |S|\}$$

This statement is an arbitrarily ordered sequence of all statements within S.

Semantically equivalence

Given two statements s_1 and s_2, we say that s_1 and s_2 are semantically equivalent, noted $s_1 \simeq s_2$, if they are syntactically equal, except for **skip**. Because **skip** does not have any impact on evaluation, adding or removing one to a statement doesn't fundamentally change it.

1.2 Constant-Time

The constant-time programming discipline as presented in the introduction also has a more formal definition: we say that a program P abides by the constant-time programming discipline if starting from environments that agree on the variables containing public values, the execution traces of leakage events are indistinguishable for an attacker. Given the leakage semantics, the constant-time property (see Definition 5) can be formalized as a non-interference property [Bar+19] with respect to a low-equivalence relation over environments. In the following definition, the set L is to be thought of as the set of public ("low") variables.

Definition 5: Constant-time

Let L be a set of variables and P be a program. The program P abides by the constant-time programming discipline for L, written $CT(P, L)$, if the following non-interference property holds:

$$CT(P,L) \triangleq \bigwedge \begin{pmatrix} \sigma_1 \equiv_L \sigma_2 \\ (P,\sigma_1) \downarrow^{t_1} \sigma_1' \\ (P,\sigma_2) \downarrow^{t_2} \sigma_2' \end{pmatrix} \Rightarrow t_1 = t_2$$

where $\sigma \equiv_L \sigma' \triangleq \forall x \in L, \sigma(x) = \sigma'(x)$.

Note that programs respecting the constant-time programming discipline do not have the same timing behavior for any input. The guarantee is that programs run with 2 distinct secrets execute the exact same sequence of instructions and perform the exact same memory accesses in the same order. In practice, this is an effective countermeasure protecting against micro-architectural timing leaks due to branch prediction and cache memory.

For example, we can try to apply this definition to the programs presented in Figure 3.

Example 1: Constant-time property of the examples of Figure 3

The first program, which we *know* is constant-time, is

```
...
if i == 0 then
    ...
else
    ...
```

The constant-time property of this program is based on the fact that i is a public variable. Hence, we would define $L = \{i\}$. And, given that the statements hidden behind the $\cdots$ are constant-time as well, the semantics rule for the evaluation of a statement states that the trace contains only the value of the guard (here $i == 0$), and the traces of the branches. Because the branches are constant-time, their traces follow the non-interference property. And, for two environments $\sigma_1 \equiv_L \sigma_2$, we have that $\sigma_1(i) = \sigma_2(i)$ because $i \in L$, and the traces are identical.

However, if we look at the third example, x = t[secret]; , with for example $L = \emptyset$, the semantics rule tells us that the trace of this statement contains the value of *secret*. And, two equivalent environments $\sigma_1 \equiv_L \sigma_2$ do not necessarily agree on the value of *secret*. Hence, the statement does not follow the non-interference property and is not constant-time either.

1.2.1 Constant-time Type System

The constant-time programming discipline can also be enforced by a flow-sensitive information flow control type system [Bar+14] in the style of Hunt and Sands [HS06]. A typing judgment is of the form $\Delta \vdash^{ct} \Gamma\{p\}\Gamma'$. The typing environments $\Gamma, \Gamma', \Delta : \mathit{Var} \to \{\mathbf{H}, \mathbf{L}\}$ map a program variable x to its type $\tau \in \{\mathbf{H}, \mathbf{L}\}$. Γ and Γ' assign types to scalar variables while Δ assigns types to array variables. As the type system is flow-sensitive for scalar variables, Γ is the typing environment before running P and Γ' is the typing environment obtained after running P. The typing environments for arrays Δ is not flow-sensitive. The rationale is that, unlike a variable assignment, an array update would be modeled as a *weak update*, and therefore it is unlikely flow-sensitivity would increase

precision. The typing judgment for expressions is of the form $\Delta, \Gamma \vdash^{ct} e : \tau$. Compared to the usual Volpano-Smith style type systems [VIS96; HS06], the type system is flow-sensitive and enforces the additional typing constraints that conditions and array indices must be of type $\mathbf{L}$. Therefore, we obtain the typing rules of Definition 6.

Definition 6: Constant Time Type System

$$\frac{}{\Delta, \Gamma \vdash^{ct} x : \Gamma(x)} \qquad \frac{}{\Delta, \Gamma \vdash^{ct} i : \mathbf{L}} \qquad \frac{\Delta, \Gamma \vdash^{ct} e : \mathbf{L}}{\Delta, \Gamma \vdash^{ct} t[e] : \Delta(t)}$$

$$\frac{\Delta, \Gamma \vdash^{ct} e_i : \tau_i \quad i \in \{1, 2\}}{\Delta, \Gamma \vdash^{ct} e_1 \oplus e_2 : \bigsqcup_i \tau_i}$$

$$\frac{}{\Delta \vdash^{ct} \Gamma\{\mathbf{skip}\}\Gamma} \qquad \frac{\Delta \vdash^{ct} \Gamma_1\{s_1\}\Gamma_2 \quad \Delta \vdash^{ct} \Gamma_2\{s_2\}\Gamma_3}{\Delta \vdash^{ct} \Gamma_1\{s_1; s_2\}\Gamma_3}$$

$$\frac{\Delta, \Gamma \vdash^{ct} e : \tau}{\Delta \vdash^{ct} \Gamma\{x = e\}\Gamma[x \mapsto \tau]} \qquad \frac{\Delta, \Gamma \vdash^{ct} e_1 : \mathbf{L} \quad \Delta, \Gamma \vdash^{ct} e_2 : \tau_2 \quad \tau_2 \sqsubseteq \Delta(t)}{\Delta \vdash^{ct} \Gamma\{t[e_1] = e_2\}\Gamma}$$

$$\frac{\Delta, \Gamma \vdash^{ct} c : \mathbf{L} \quad \Delta \vdash^{ct} \Gamma\{s_1\}\Gamma_1 \quad \Delta \vdash^{ct} \Gamma\{s_2\}\Gamma_2}{\Delta \vdash^{ct} \Gamma\{\mathbf{if}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\}\Gamma_1 \sqcup \Gamma_2}$$

$$\frac{\Gamma \sqsubseteq \Gamma' \quad \Gamma_1 \sqsubseteq \Gamma' \quad \Delta \vdash^{ct} \Gamma'[i \mapsto \mathbf{L}]\{s\}\Gamma_1}{\Delta \vdash^{ct} \Gamma\{\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s\}\Gamma'}$$

Theorem 1 states that the type-system of Definition 6 ensures that the program is constant-time according to Definition 5.

Theorem 1: Soundness of Constant Time Type System

Consider a program P typable according to the type-system of Definition 6

$$\Delta \vdash^{ct} \Gamma\{P\}\Gamma'$$

We have that $CT(P, L)$ for $L = \{x \mid \Gamma(x) = \mathbf{L} \vee \Delta(x) = \mathbf{L}\}$.

Proof Outline. By induction over the type derivation of P. The only complex cases are the branching and the array accesses. In both cases, the expression of the guard (resp. the index) is public, and its value should be shared between two equivalent environments. By the semantics rule, we can prove that the non-interference property is true. □

1.3 Conditional move

1.3.1 Definition

As stated above, for a program to be constant-time, the control flow shall not depend on secret values. Thus, if e is a secret, simple programs such as **if** c **then** $x = 0$ **else** $x = 1$ are insecure. However, such programs, performing a choice between two values, have a real use-case in cryptographic code. To allow for these kinds of instructions, a constant-time *selection* operation is introduced: **ctselect**. This **ctselect** operation is semantically equivalent to a `if` conditional. For example, $x =$ **ctselect**$(c, 0, 1)$ is the constant-time version of the program above.

To effectively implement such an operation, we can for example use bitwise manipulation or arithmetic operation. One such implementation is displayed in Example 2. This definition of **ctselect** is semantically correct because if c is true, $e_1 \& c$ boils down to e_1 and if c is false, $e_2 \& \neg c$ amounts to e_2, giving us the exact semantics of **ctselect**. Moreover, if c, e_1, and e_2 are themselves constant-time expressions, it is easy to prove using the $\oplus$ semantics rule that **ctselect**(c, e_1, e_2) is constant-time as well.

Example 2: Implementation of constant-time selection using bitwise manipulation

```
ctselect(c, e1, e2) = (e1 & c) | (e2 & ¬ c)
```

However, even such a construct is not flawless, and compilers might break it by re-introducing branches in some compilation passes.

Still, there exists a way to ensure the constant-time property of the **ctselect** operation: **cmove**. This is an x86 instruction designed for this purpose, often used to write constant-time machine code. The **cmove** instruction has a semantics similar to one of a branch: **cmove** $e\ r1\ r2$ moves the value of $r2$ to $r1$ if e is true. Otherwise, nothing happens. Even though its semantics makes it look like a branch, its execution does not depend on the condition, and we can assume this instruction is constant-time, as its trace is constant whatever the value of e is. We can then use the **cmove** instruction to implement the **ctselect** operation :

Example 3: Implementation of ctselect using cmove

$$r = \mathbf{ctselect}(e, r1, r2) \mapsto \begin{array}{l} \mathbf{cmove}\ e\ r\ r1 \\ \mathbf{cmove}\ (\sim e)\ r\ r2 \end{array}$$

From now on, we suppose that every ternary conditional operator behaves just as a **ctselect** operation and is constant-time *i.e.*,

$$\mathbf{ctselect}(e_1, e_2, e_3) \equiv e_1?e_2{:}e_3$$

1.3.2 Updated Definitions

Having defined this new **ctselect** operation, we can update the language defined in Definition 1 to add such an expression within our expressions :

Definition 7: Updated syntax of L with a ctselect operator

Expressions:

$$\begin{array}{lll} \textit{expr} \ni e & ::= & x \mid c \mid e_1 \oplus e_2 \mid e_1?e_2{:}e_3 \mid t[e] \end{array}$$

Statements:

$$\begin{array}{lll} \textit{stmt} \ni s & ::= & \textbf{skip} \\ & & \mid x = e \\ & & \mid t[e_1] = e_2 \\ & & \mid s_1; s_2 \\ & & \mid \textbf{if } x \textbf{ then } s_1 \textbf{ else } s_2 \\ & & \mid \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s \end{array}$$

By adding a ternary operator in the syntax, we have to give it a semantics as well. This is done in Definition 8 by showing the new rule for our conditional expression. The conditional expression is strict and evaluates all its arguments without leaking the value of the condition e_1. By evaluating $e_1?e_2{:}e_3$, we evaluate both e_2 and e_3, then e_1, and produce either of the values accordingly. This way, a conditional expression always generates the trace $t_1 \cdot t_2 \cdot t_3$

Definition 8: Updated leaky semantics for expressions with a ctselect operator

$$\frac{(e_i, \sigma) \downarrow^{t_i} v_i \quad i \in \{1,2,3\} \qquad v = if\ v_1\ then\ v_2\ else\ v_3}{(e_1?e_2{:}e_3, \sigma) \downarrow^{t_1 \cdot t_2 \cdot t_3} v}$$

Similarly, we update the constant-time type system given in Definition 6 to take into account the newly created ternary operator. We display in Definition 9 the typing rule used for this construction. Note that if the three expressions are accepted by the type system, the conditional move is accepted whatever the type of e_1 is.

Definition 9: Update on the Constant Time Type System

$$\frac{\Delta, \Gamma \vdash^{ct} e_i : \tau_i \quad i \in \{1,2,3\}}{\Delta, \Gamma \vdash^{ct} e_1 ? e_2 : e_3 : \bigsqcup_i \tau_i}$$

1.3.3 Branch Removal

For a program to be constant-time, there must be no secret conditionals. Instead of trying to bend the logic behind a program to satisfy this constraint, we can simply remove all secret branches. Such a pass, along with a whole compiler, is presented in FaCT[Cau+19]. This pass, named *Branch removal*, works by predicating each instruction by the control predicate, which is constructed from all conditions in the control flow leading to this instruction. Predicate an instruction by a value means that we use the **ctselect** operation defined above to ensure that the instruction is actually executed only if this value is true. In the case of an assignment, this means transforming $x = y$ to $x = h?y{:}x$, leaving x untouched if h is false. Note that in this case, the operator **ctselect** can be implemented at low-level with a single **cmove**. To keep track of the control flow leading to an instruction, we append '&e' (resp. '&$(\neg e)$') to the control predicate when we cross a **then** (resp. **else**) branch.

1.3.4 Examples

Consider the following code, which, depending on a secret h, assigns the variable x to either l_1 or l_2.

Example 4: Simple unsecure code snippet

```
if h then x = l1 else x = l2
```

After branch removal, we get the following branchless code, which eliminates the leakage due to the conditional.

Example 5: Transformed version of the code in Example 4

$x = h?l_1:x;\ \ x = \neg h?l_2:x$

Now, if we look at a snippet containing two nested conditionals, such as in Example 6, the transformation will now combine both conditional guards to form the control predicate, as shown in Example 7.

Example 6: Complex unsecure code snippet

if h_1 **then** $t[1] = l_3;$ **else** {**if** h_2 **then** $t[2] = l_4;$ **else skip**}

Example 7: Transformed version of the code in Example 6

$t[1] = h_1?l_3:t[1];\ \ t[2]=((\neg h_1)\&h_2)?l_4:t[2];$

1.3.5 Limitations

Public safety

An issue with *Branch removal*, such as presented above, is that it is not always a semantics-preserving transformation. The problem arises when the safety of memory accesses within the **then** (resp.the **else**) branch relies on whether the condition holds or not. For example, suppose that l_1 and l_2 in Examples 4, 5 perform a memory access *e.g.*, t[i] and that the condition h guards against out-of-bound accesses *i.e.*, $h := 0 \leq i < size(t)$. After *Branch removal*, the target code performs the memory access unconditionally and may perform an illegal access. To solve this issue, the FaCT compiler generates verification conditions to ensure that the memory accesses are still valid after transformation *i.e.*, that the expressions l_1 and l_2 in a predicated assignment $x = h?l_1{:}l_2$ are safe to evaluate, independently of the value of h. The compiler then outsources them to an external solver.

Indirect leakage

Because the *Branch removal* transformation is a local process, and only considers instructions one at a time, it is insufficient to remove indirect leakage due to assignment inside the conditional. Consider the Program P0.

Program P0: Program demonstrating *branch removal*'s limitation

```
if h then (x=l1; y=t[x]) else (x=l2; y=t[x])
```

By performing the transformation, the following **then** branch will be generated :

$$x = h?l_1{:}x; \quad y = h?t[x]{:}y;$$

Because the semantics of **cmove** is to evaluate all expressions, $t[x]$ is always computed. However, because of the first predicated statement, x is tainted by h, and the memory access is unsecure, potentially leaking the value of a secret.

1.4 Constant-Time Array Traversal

To avoid leaking an array access $y = t[x]$, an inefficient yet standard countermeasure consists of iterating over all the indices i of the array and selecting the relevant value using a conditional expression.

$$y = t[x] \rightarrow \textbf{for } i \textbf{ from } 0 \textbf{ to } size(t){-}1 \textbf{ do } y = (x == i)?t[i] : y$$

The semantics is preserved by this transformation. Indeed, the assignment $y = (x == i)?t[i{:}y$ amounts to $y = t[x]$ if $x = i$, and $y = y$ otherwise. In other words, the statement is significant only if $x = i$. And, we only apply these transformations on safe programs, so we know that $0 \leq x < size(t)$, and there is a i such that $x = i$. This means that the loop amounts to $y = t[x]$. Moreover, the conditional move is a constant-time operation, and the size of arrays is public, so nothing leaks from this array traversal. In the cases of memory write, the intuition is the same: we introduce a **for** loop and register the write only for the correct index. The only difference is that the $t[i]$ is present twice, and that when $x = i$, the assignment is equal to $t[i] = t[i]$, which is also harmless. This would look like :

$$t[x] = y \rightarrow \textbf{for } i \textbf{ from } 0 \textbf{ to } size(t)-1 \textbf{ do } t[i] = (x == i)?y : t[i]$$

This solution is a last resort one: we effectively multiply the complexity of the program by $size(t)$. However, it is a working one, and applying it just after if-conversion would allow us to circumvent the second limitation presented above. Because of its cost, we nonetheless aim to use it as little as possible.

Chapter 2

Program transformation

In this chapter, we aim to describe how we can improve the state-of-the-art transformations described in Chapter 1. We describe each step of our reasoning to propose new, improved transformations in a different section. We conclude with an overview of the whole transformation as we propose it. Note that these steps are described in a somewhat informal way: even though a formal description is given, we hide in a black box the type system which will give us the information needed to apply any transformation. Along each one of the steps is given, if possible, an outline of the proofs for semantics preservation. Security, and thus the constant-time property, of the transformed program will be tackled in Chapter 4, but we ensure here that the transformation shall not change the behavior of the program. Because the proofs in this chapter are just outlines, some formalization can be lacking. In most cases, a similar intuition is formalized and detailed in Chapter 4.

2.1 Delayed if-conversion

As shown earlier using Program P0, the usual *Branch removal* transformation is not enough to handle certain classes of programs, notably because the transformation is turning indirect flows into direct by introducing the predicated code. However, the predicated code is added *in place* and we introduce a direct flow where there would have been none in the original program, making the array access insecure. We then propose to *delay* the *if-conversion* pass *i.e.*, the introduction of the predicated code, at the cost of introducing extra variables. This allows for the array accesses to be done after a temporary assignment but before the predicated statement, without any dependence on the condition guard.

If we try to apply it to the Program P0, we get the Program P1. The resulting program has the same semantics as the original one: y is either $t[l_1]$ or $t[l_2]$ depending on h, and is constant-time, the only memory accesses being done are those on l_1 and l_2, two safe values.

Reminder: Program P0

```
if h then (x=l1; y=t[x]) else (x=l2; y=t[x])
```

Program P1: Result of the delayed if-conversion on Program P0

```
xt = l1; yt = t[xt]; xe = l2; ye = t[xe];
x = h?xt:xe; y = h?yt:ye;
```

However, this approach only works on scalar variables. If we try to apply it as it is on arrays, such as the Program P2, we are faced with a problem as to how to delay the predicated statement for the $t[x] = 0$ array write. The wrong solution is basically to apply the delayed if-conversion, and we get the Program P3. Unfortunately, such a resulting program does not have a consistent semantics: either $t[x_t]$ or $t[x_e]$ has been wrongfully rewritten, and does not hold the same value as it does after the evaluation of the original program.

Program P2: Program containing a conditional array write

```
if h then (x = l1; t[x] = 0) else (x = l2; t[x] = 0)
```

Program P3: Trying to apply delayed if-conversion on Program P2

```
xt = l1; t[xt] = 0;
xe = l2; t[xe] = 0;
x = h?xt:xe; t[x] = h?t[xt]:t[xe];
```

To ensure that the semantics are well-preserved, we limit the delaying of if-conversion to work only on scalar variables. When faced with a memory write, we immediately choose the correct value. Such a way to do things would result in Program P4. In such a program, the memory writes only have an effect if the correct branch is evaluated. Thus,

the semantics is preserved. Unfortunately, this introduces direct flow into the arrays, and we often can't transform memory accesses done using memory accesses as indexes, such as $t[t[x]]$. We will discuss more about this issue in Section 1.4.

Program P4: Trying to apply improved delayed if-conversion on Program P2

$$
\begin{aligned}
&x_t = l_1;\ \ t[x_t] = h?0:t[x_t];\\
&x_e = l_2;\ \ t[x_e] = \neg h?0:t[x_e];\\
&x = h?x_t:x_e;
\end{aligned}
$$

2.1.1 Prerequisites

The transformation presented above is quite straightforward: we do not need to have any information on the program to apply it, working statement by statement is enough. However, we need a formal structure to keep track of all the fresh variables and their renaming: *renaming maps*.

Fresh Variables

Our main tool to actually delay the transformation is the introduction of temporary variables. To do so, we need to be able to introduce fresh variables *i.e.*, variables whose names won't clash with already existing ones. To this end, we define a function yielding such *fresh* variables: `fresh`. The formal definition of this function can be found in Definition 10. Note that to be sure to have some fresh variables available, we use the set $\mathbb{P}_f(\mathcal{V})$ of finite set of variables, assuming that $\mathcal{V}$ to be infinite. Thus, $fresh(x)$ always exists. This isn't a true restriction because we usually use the set of variables of a program, which is obviously finite.

Definition 10: Definition of the `fresh` function

$$
\begin{aligned}
&\mathbb{P}_f(\mathcal{V}) \to \mathcal{V}\\
&V \mapsto x \text{ s.t } x \notin V
\end{aligned}
$$

In short, the `fresh` function takes a set of variables V as input and yields a variable name x not in V. This allows us to new names not interfering with the current set of

variables. For the sake of simplicity, we will sometimes omit the V input of the function, supposing that we use the current set of variables used by the program or in the transformation.

Renaming maps

Renaming maps, noted ρ, are defined as total functions from a set of variables $\mathcal{V}$ to the same set $\mathcal{V}$. These represent a renaming relation between two names. As such, we say that $\rho(x)$ is a *renaming* of x. To ensure a well-formed renaming, we introduce a few necessary properties on any map ρ:

— ρ is idempotent *i.e.*, $\forall x, \rho(\rho(x)) = \rho(x)$. This implies any renaming is finite. A value serving as a new variable name cannot be renamed itself.

— two different variables cannot be renamed to the same name *i.e.*, $\forall x, y, \rho(x) = \rho(y) \implies x = \rho(y) \vee y = \rho(x) \vee x = y$. Thanks to the idempotent property, the other direction of the implication is also true ($x = \rho(y) \implies \rho(x) = \rho(\rho(y)) = \rho(y)$).

On top of this crude definition, we define some operations on and using renaming maps. The most important of these operations are the update of a map, its definition on a subset of variables, and its application to expressions and statements. However, we also define how to compute the set of variables renamed by a map, or what it means for two maps to be disjoint or non-interfering.

Set of renamed variables. We note $\mathcal{V}^\rho$ the set containing all variables renamed by ρ *i.e.*, $\forall x \in \mathcal{V}, x \in \mathcal{V}^\rho \iff \rho(x) \neq x$, that is if x is not a fixed point of ρ.

Set of renamings. Similarly, we note $\mathcal{V}_\rho$ the set containing all variables renamed into by ρ *i.e.*, $\forall x \in \mathcal{V}, x \in \mathcal{V}_\rho \iff \exists y \neq x, \rho(y) = x$.

Disjoint maps. We say that two maps are disjoint if their sets of renamings are distinct *i.e.*, given two maps ρ and ρ', they are distinct if and only if $\mathcal{V}_\rho \cap \mathcal{V}_{\rho'} = \emptyset$.

Non-interfering maps. We say that two maps are non-interfering if their sets of renamed variables are disjoint *i.e.*, given two maps ρ and ρ', they are non-interfering if and only if $\mathcal{V}^\rho \cap \mathcal{V}^{\rho'} = \emptyset$.

Updating a renaming map. When using renaming maps, we may need to update the renaming of a variable, or even add one. We propose a notation to do so, $\rho[y \mapsto y']$, detailed in Definition 11.

Definition 11: Updating a renaming map: $\rho[y \mapsto y']$

$$dom(\rho) \to Im(\rho)$$
$$x \mapsto \begin{cases} \rho(x) \text{ if } x \neq y \\ y' \text{ otherwise} \end{cases}$$

This defines a new function behaving just as ρ, except on y, in which case the name yielded is the new renaming y'. Moreover, we can actually ensure that the well-formed properties are preserved when updating, as stated in Theorem 2.

Theorem 2: Update of a renaming map

Let ρ be a renaming map and y, y' two variables such that $y \notin \mathcal{V}^\rho, y' \notin \mathcal{V}^\rho$ and $y' \notin \mathcal{V}_\rho \vee y' = \rho(y)$ *i.e.*, y and y' are not renamed variables, and there isn't already a renaming to y' from a different variable than y. Then, the updated map $\rho[y \mapsto y']$ is also a renaming map.

Proof Outline. Because y' is not already a renaming of ρ, we have that $\rho'(\rho'(y)) = \rho'(y') = y'$, and, ρ being idempotent, ρ' is idempotent too. Moreover, y' and y are not renamings of ρ, so if no renaming of a variable by ρ' can be equal to y, and only y, y' can be renamed to y'. Thus, ρ' is a renaming map. □

Initializing a renaming map. The most simple renaming map possible is the identity function id, as shown in Theorem 3.

Theorem 3: Identity is a renaming map

The function $id : x \mapsto x$ is a renaming map.

Proof. For all x, y, $id(id(x)) = id(x) = x$, so id is idempotent, and if $id(x) = id(y)$, $x = y$. □

However, we often need more complex maps, which actually rename some of the variables of the set $\mathcal{V}$. In these cases, we use the *init* function described in Definition 12.

Definition 12: Initialization of a renaming map

$$init(v,u) = \begin{cases} let\ \rho = init(v \setminus x, u)\ in\ \rho[x \mapsto x'] & if\ \exists x, x', x \in v \wedge x' = \texttt{fresh}(\mathcal{V}_\rho \cup u) \\ id & otherwise \end{cases}$$

This *init* function allows us to create renaming maps renaming all the variables, and only those, within a set v, and without using any of the variables in the set u. Calling *init* with v and an empty set creates a map equivalent to $id[x_1 \mapsto x'_1] \dots [x_n \mapsto x'_n]$ with $v = \{x_1, \dots, x_n\}$ and $\{x'_1, \dots, x'_n\}$ fresh variables well-chosen so that there are not any renaming conflicts. Note that this function does not ensure that the created maps are not redundant. For example, the *id* function is an adequate yield for any $init(v, \emptyset)$. To prevent this, the first parameter must be repeated in the second *i.e.*, $init(v, v)$. Any function created using *init* is a renaming map, as stated in Theorem 4.

Theorem 4: Initialization of a renaming map

Let v be a set of variables, then $init(v, u)$ is a renaming map.

Proof Outline. By induction of v. If v is empty, we know that v is a renaming map by Theorem 3. Otherwise, by induction hypothesis and Theorem 2. □

Renaming of expressions. We can overload any renaming map to handle expressions instead of simply variables. Given a renaming map ρ, we can overload ρ into ρ_e by following the rules described in Definition 13.

Definition 13: Overloading of a renaming map for expressions

$$\begin{aligned} \rho_e(x) &= \rho(x) \\ \rho_e(c) &= c \\ \rho_e(e_1 \oplus e_2) &= \rho_e(e_1) \oplus \rho_e(e_2) \\ \rho_e(e_1 ? e_2 : e_2) &= \rho_e(e_1) ? \rho_e(e_2) : \rho_e(e_3) \\ \rho_e(t[e]) &= t[\rho_e(e)] \end{aligned}$$

2.1.2 Formal description

As shown at the start of this section, the delayed if-conversion applies only to conditional. Hence, if we were to try to formalize such a transformation, we would obtain a definition close to the one shown in Definition 14. Note that the DIC^{V_P} is parameterized by a V_P set, corresponding to the set of all variables used (*i.e.*, modified and read) by the overall program to be transformed. This set is used to ensure that any fresh variable is not in conflict with already used ones.

Definition 14: Formal definition of Delayed If-conversion

We call $DIC^{V_P} : stmt \mapsto stmt$ the function corresponding to the delayed if-conversion. It is defined as :

$$
\begin{array}{lcl}
DIC^{V_P}(\textbf{skip}) & = & \textbf{skip} \\
DIC^{V_P}(x = e) & = & x = e \\
DIC^{V_P}(t[e_1] = e_2) & = & t[e_1] = e_2 \\
DIC^{V_P}(s_1; s_2) & = & DIC^{V_P}(s_1); DIC^{V_P}(s_2) \\
DIC^{V_P}(\textbf{if } h \textbf{ then } s_t \textbf{ else } s_e) & = & DICIF^{V_P} if(\textbf{if } h \textbf{ then } s_t \textbf{ else } s_e) \\
DIC^{V_P}(\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s) & = & \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } DIC^{V_P}(s)
\end{array}
$$

where $DICIF^{V_P}$ is the result of the transformation described earlier and is approached below.

The approach explained to handle conditionals at the start of this section is made of three main steps: 1) initialization, which creates the fresh variables, and initiates them; 2) renaming, which uses the freshly created variables to rename both the then and the else statements and 3) finalization, which merges back the fresh variables into the initial ones. We explain in more detail and formalize each of those steps in the following paragraphs.

Initialization

The first step is to create new variables used as renamings for variables modified by the branch of the conditionals. To do so, we create two renaming maps ρ_t and ρ_e, using the *init* function:

$$\rho_t = init(mod(s_t), V_P) \quad \rho_e = init(mod(s_e), V_P \cup \mathcal{V}_{\rho_t})$$

These two maps are disjoint, and each renames exactly the variables modified by its corresponding branch. Once those names are generated, we can create the statements assigning correct values to the fresh variables: pre_t, pre_e initialize the variables respectively of the branch then and else. Exact definitions of these statements can be found in Definition 15.

Definition 15: Initializing statements for delayed if-conversion

$$\begin{aligned} pre_t &= seq(\{\rho_t(x) = x \mid x \in \mathcal{V}^{\rho_t}\}) \\ pre_e &= seq(\{\rho_e(x) = x \mid x \in \mathcal{V}^{\rho_e}\}) \end{aligned}$$

It would also be possible to create the map dynamically, each time the transformation encounters a new variable. However, this static way eases the writing of the transformation and allows for more flexibility, which we will discuss in Section 2.4.

Renaming

Once the renaming maps are created, and the variables are initiated, we may use those to rename both branches of the conditional. We do this inductively, statement by statement. For most of the cases, the renaming is simply a recursive call. However, when transforming a memory write, and as explained above, we have to do the if-conversion on-place *i.e.*, without delaying it. The results of the renaming for each case can be found in Definition 16. The renaming function takes three inputs: the renaming map, the statement to be renamed, and the guard of the current branch, to allow for direct conversion. We will call it with ρ_t, s_t, and h' (respectively ρ_e, s_e, and $\neg h'$) to get the two renamed branches: $Rn^{\rho_t}_{h'}(s_t)$ and $Rn^{\rho_e}_{\neg h'}(s_e)$.

Definition 16: Renaming a statement

$$\begin{aligned} Rn^\rho_h(\textbf{skip}) &= \textbf{skip} \\ Rn^\rho_h(x = e) &= \rho(x) = \rho_e(e) \\ Rn^\rho_h(t[e_1] = e_2) &= t[\rho_e(e_1)] = h?\rho_e(e_2):\rho_e(e_1) \\ Rn^\rho_h(s_1; s_2) &= Rn^\rho_h(s_1); Rn^\rho_h(s_2) \\ Rn^\rho_h(\textbf{if } h' \textbf{ then } s_t \textbf{ else } s_e) &= \textbf{if } h' \textbf{ then } Rn^\rho_h(s_t) \textbf{ else } Rn^\rho_h(s_e) \\ Rn^\rho_h(\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s) &= \textbf{for } \rho(x) \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } Rn^\rho_h(s) \end{aligned}$$

Finalization

At the end of the conditional, and to allow for the continuation of the program, we have to merge back the renamed variables to their original, while keeping track of which of the branches should be preserved. This is done with a single statement, which assigns to each variable the renaming of the correct branch:

$$post = seq(\{x = h'?\rho_t(x):\rho_e(x) \mid x \in \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}\})$$

When applying all those steps, we obtain the description of the DIC^{V_P} function for a conditional, such as described in Definition 17.

Definition 17: Delayed if-conversion of a conditional

$$\frac{\begin{array}{c}\rho_t = init(mod(s_t), V_p) \quad \rho_e = init(mod(s_e), V_p \cup \mathcal{V}_{\rho_t}) \\ pre_t = seq(\{\rho_t(x) = x \mid x \in \mathcal{V}^{\rho_t}\}) \quad pre_e = seq(\{\rho_e(x) = x \mid x \in \mathcal{V}^{\rho_e}\}) \\ post = seq(\{x = h?\rho_t(x):\rho_e(x) \mid x \in \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}\})\end{array}}{DIC^{V_P}(\textbf{if } h \textbf{ then } s_t \textbf{ else } s_e) = pre_t; pre_e; Rn_h^{\rho_t}(s_t); Rn_{\neg h}^{\rho_e}(s_e); post}$$

Example using Program P0

Reminder: Program P0

```
if h then (x=l1; y=t[x]) else (x=l2; y=t[x])
```

If we look again at Program P0, the whole program is a single conditional, so the only applicable rule is the one shown in Definition 17. If we actually try to apply it, we obtain $\rho_t = id[x \mapsto x_t][y \mapsto y_t]$ and $\rho_e = id[x \mapsto x_e][y \mapsto y_e]$. If not for the $pre_t = (x_t = x; y_t = y)$ and pre_e statements, which do not have any effects on the semantics of the program, the resulting program is equal to Program 1, shown earlier.

2.1.3 Semantics preservation

Once a formal description of the transformation has been given, we can provide some guarantees about it. We will focus on the semantics side of the transformation, and

try to prove that the behavior of the program is preserved and that the trace leaked is more secure than initially. This would basically ensure that if a statement s has a certain semantics, its transformed version $DIC^{V_P}(s)$ yields an equivalent environment and a more secure trace. To pursue this, we first have to investigate the effect of a renaming on an environment. Then, we will look at the renaming of expressions and the initialization statements, before tackling the renaming of statements. For simplicity's sake, we consider in this chapter only the environments limited to the variables of the program and their current renamings. That is, if two environments differ on a variable that is neither in the original program nor a current renaming, we ignore it. Again, this kind of consideration are lifted in the more detailed Chapter 4

Renaming of an environment

To adapt a mapping environment to a renaming map ρ, we overload ρ to ρ_s taking two environments as input, as described in Definition 18. This overloaded function maps the value of a variable x to its renaming $\rho(x)$. To keep track of the initial value of renamed variables, we use the σ_i environment. Because an array cannot be renamed, the evaluation of an array through a renamed environment will always use σ_i. Moreover, we suppose in the rest of this chapter that we write $\rho(\sigma_i, \sigma)$ only if, if $\rho(x) = x$ then $\sigma_i(x) = \sigma(x)$ (this constraint is ignored for arrays, as they cannot be renamed anyway). We will see in Chapter 4 the details of why such an assumption is necessary.

Definition 18: Renaming a environment

$$\rho_s(\sigma_i, \sigma)[x] = \begin{cases} \sigma[\rho^{-1}(x)] & \text{if } x \in \mathcal{V}_\rho \\ \sigma_i[x] & \text{otherwise} \end{cases}$$

Such a definition directly implies a property of invariability by renaming for the environments, which is stated in Lemma 1.

Lemma 1: Stability of renaming on environment

Let σ_i, σ be two environments, ρ a renaming map, and x a variable. The value of the renaming of x by ρ in the environment σ renamed by ρ is equal to the value of x in σ, that is:

$$\rho_s(\sigma_i, \sigma)[\rho(x)] = \sigma[x]$$

Proof. By definition of ρ_s: $\rho_s(\sigma_i, \sigma)[\rho(x)] = \sigma[\rho^{-1}(\rho(x))] = \sigma[x]$. □

Evaluating the initialization statements

The transformation starts by defining two renaming maps and introducing two statements initializing the fresh variables. Evaluating these statements yields a final environment renamed by ρ_t and ρ_e. To show this, we state in Lemma 2 that evaluating a *seq* statement such as pre_e yields the renamed version of the map. Hence, the sequencing of such statements yields a renaming map of the form of $\rho_{e_s}(\rho_{t_s}(\sigma, \sigma), \sigma))$.

Lemma 2: Renaming a *seq* statement

Let σ be an environment and ρ a renaming maps. The evaluation of the statement generated by ρ on σ yields a renamed environment $\rho_s(\sigma)$:

$$(seq(\{\rho(x) = x \mid x \in \mathcal{V}^\rho\}), \sigma) \downarrow^\epsilon \rho_s(\sigma, \sigma)$$

Proof Outline. By induction over the size of $\mathcal{V}^\rho$. If ρ is the identity function, $\rho_s(\sigma, \sigma) = \sigma$. Otherwise, there is ρ', x, x' such that $\rho = \rho'[x \mapsto x']$, and the statement generated by ρ is the statement generated by ρ' followed by $x' = x$. We conclude with the induction hypothesis and the definition of $\downarrow$. □

Evaluating a renamed expression

Because our renaming maps affect variables, it ends up modifying expressions. Hence, we have to make sure that the semantics of expressions is unchanged before and after renaming. This property is stated in Lemma 3.

Lemma 3: Evaluating a renamed expression

Let σ be an environment, ρ a renaming map, e an expression, t a trace, and v a value. If e evaluates to v, then $\rho_e(e)$ also evaluates to v in the according environment, that is:

$$(e, \sigma) \downarrow^t v \implies (\rho_e(e), \rho(\sigma, \sigma)) \downarrow^t v$$

Proof Outline. By induction over e, and by Lemma 1. □

Evaluating a renamed instruction

The end goal of the delayed if-conversion is to *apply* the renaming map to the whole conditional. Doing so obviously affects the semantics of both branches. The ideal result would be a preservation of the semantics: every statement evaluating before transforming should do so after. This intuition is stated in Lemma 4.

Lemma 4: Evaluation of a renamed instruction

Let σ, σ' be two environments, ρ be a renaming map, t a trace, s a statement, and h a variable, then if the evaluation of s within σ yields σ', the evaluation of its renaming by ρ within $\rho_s(\sigma_i, \sigma)$ yields $\rho_s(\sigma'_i, \sigma')$, where σ'_i is σ_i if h evaluates to $false$ (arrays are left unchanged), or $\sigma_i[t \mapsto \sigma'[t] \mid t \text{ an array}]$ otherwise, that is :

$$(s, \sigma) \downarrow^t \sigma' \implies (Rn^{\rho}_h(s), \rho_s(\sigma_i, \sigma)) \downarrow^{t'} \rho(\sigma'_i, \sigma')$$

where

$$\sigma'_i = \begin{cases} \sigma_i[t \mapsto \sigma'[t] \mid t \text{ an array}] & \text{if } h \\ \sigma_i & \text{otherwise} \end{cases}$$

with t' being larger than t due to the re-evaluation of e_1 in the array write rule, which duplicates the computation of $\rho(e_1)$, and thus the trace associated.

Proof Outline. By induction over s, and by Lemma 3. The σ_i is preserved for all but arrays because all assignments are done on variables renamed by ρ, by definition of ρ using the *init* function. For arrays, the disjunction between both cases is caused by the semantics of the conditional expression. □

However, we face an issue: memory accesses are made insecure by the branch removal. Indeed, if we look at Program P5, the transformation would yield Program P6, introducing potential unsafe memory access in the form of `t[i] = h'?0:t[i]`.

Program P5: Unsafe program w.r.t memory accesses

```
h = (i < size);
if h then t[i] = 0 else skip
```

Even with the prerequisite that Program P5 is safe, we have no way to ensure that the access t[i] is safe if the expression i < size(t) is false. This issue is actually blocking if we try to prove anything on the whole transformation: the semantics of **if** does not give any guarantee on the behavior of s_t if h evaluates to $false$ (or s_e if h evaluates to $true$). A solution to this peculiar issue is detailed in the following Section 2.2.

Program P6: Result of delayed if-conversion on Program P5

```
h = (i < size);
t[i] = h?0:t[i]
```

2.2 Index Sanitizing

As shown in the previous section, the delayed if-conversion as it is is unsafe: it introduces possible out-of-bounds accesses on memory access done within an unsecure context. To ensure that the resulting program is just as safe, we need to guarantee that all indexes are within the bounds of the array. Our assumption that the source program is safe means that it does not make any out-of-bounds array accesses. Hence, we have to make sure that the evaluation of the access is safe especially when the guard is false *i.e.*, when the value should be unchanged. FaCT [Cau+19] does so by using type constraints and a solver to solve those constraints.

Program P7: Unsafe program w.r.t memory read

```
h = (i<size);
if h then x = t[i] else skip
```

To avoid having to rely on an external solver, we can instead introduce dynamic bound checks to prevent any generation of unsafe memory access. Although this solution is usable in a general setting [SP21], the constant size of our arrays makes its expression easier. Our instrumentation transforms the array access $t[x]$ into $t[0 \leq x < size(t)?x{:}0]$. Because the program is safe, we have the invariant that $0 \leq x < size(t)$, so the conditional expression always evaluates to x. Hence, the semantics of the program remains unchanged and the program remains safe after *if-conversion*. For example, if we look at Program P7, this

transformation would yield Program P8. A subsequent delayed if-conversion would yield Program P9, which is safe thanks to the dynamic bound check.

Program P8: Result of our index sanitizing transformation on Program P7

```
h = (i < size);
if h then x = t[0<=i<size(t):i:0] else skip
```

We can see that whatever the value of x may be, the array access is done within the bounds of t. Indeed, if x itself is within the bounds, the access is done as $t[x]$, otherwise, it is on 0. In a normal execution, an index is always within the bounds of the array, so the semantics is left unchanged. However, when evaluating the incorrect branch, we may have to use the $t[0]$ solution. Thanks to the mechanism of delayed if-conversion, where the value is ignored when it is the wrong branch, we do not have to worry about introducing a wrong memory access. Moreover, the conditional expression being safe, this rewriting does not add any leak to the expression.

Program P9: Result of delayed if-conversion on Program P8

```
h = (i < size);
x_t = x;  x_e = x;
x_t = t[0<=i<size(t):i:0]
x = h?:x_t:x_e
```

Similarly, this transformation handles memory writes. For example, if we look at Program P5 of the previous section, a similar transformation would yield Program P10.

Program P10: Result of our updated index sanitizing transformation on Program P5

```
h = (i<size);
if h then t[0<=i<size(t):i:0]=0 else skip
```

And, applying delayed if-conversion to Program P10 yields Program 11. This last program is safe in all cases, because our initial program is safe, hence if h is true, i is within the bounds of t.

Program P11: Result of delayed if-conversion on Program P10

```
h = (i < size);
t[0<=i<size(t)?i:0] = h?0:t[0<=i<size(t)?i:0]
```

This instrumentation has a performance overhead by evaluating an additional time the value stored at $t[x]$ but optimizing compilers should be able to remove most of the redundant checks. This will be discussed in Chapter 5.

2.2.1 Formal Description

As shown in the examples above, the index sanitization transformation is very localized, affecting only memory writes and reads. Thus, we can start by defining two transformation rules for these two cases, as shown in Definition 19. In both cases and according to the examples above, the index i is replaced by the conditional expression $(0 \leq i < size(t))?i{:}0$ which returns i if the index is in-bounds and returns 0 otherwise. As array sizes are strictly positive, $t[0]$ is always a valid access. As a result, in both cases, we get a valid array access.

Definition 19: Bound checks for array accesses

$$\text{ARR-SAN} \qquad t[i] \rightsquigarrow t[(0 \leq i < size(t))?i{:}0]$$

$$\text{ARR-ASS} \qquad t[i] = e \rightsquigarrow t[(0 \leq i < size(t))?i{:}0] = e$$

From these definitions, we can go on to formalize the whole index sanitization transformation. Just as delayed if-conversion used a renaming for expression (ρ_e) and one for instructions, index sanitization is divided into two distinct parts: one for expression (IS_e) and for instructions (IS). The first one, detailed in Definition 20 is straightforward, we recursively call it and apply ARR-SAN when needed.

Definition 20: Index sanitization for expressions (IS_e)

$$
\begin{array}{lcl}
IS_e(x) & = & x \\
IS_e(c) & = & c \\
IS_e(e_1 \oplus e_2) & = & IS_e(e_1) \oplus IS_e(e_2) \\
IS_e(e_1?e_2{:}e_3) & = & IS_e(e_1)?IS_e(e_2){:}IS_e(e_3) \\
IS_e(t[e]) & = & t[(0 \leq IS_e(e) < size(t))?IS_e(e){:}0]
\end{array}
$$

Given the locality of the transformation, instructions are handled in a similar way, we only apply IS_e and ARR-ASS when needed, as shown in Definition 21.

Definition 21: Index sanitization for instructions (IS)

$$
\begin{array}{lcl}
IS(\textbf{skip}) & = & \textbf{skip} \\
IS(x = e) & = & x = IS_e(e) \\
IS(t[e_1] = e_2) & = & t[(0 \leq IS_e(e_1) < size(t))?IS_e(e_1){:}0] = IS_e(e_2) \\
IS(s_1; s_2) & = & IS(s_1); IS(s_2) \\
IS(\textbf{if } x \textbf{ then } s_1 \textbf{ else } s_2) & = & \textbf{if } x \textbf{ then } IS(s_1) \textbf{ else } IS(s_2) \\
IS(\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } s) & = & \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } IS(s)
\end{array}
$$

The transformation, written as it is, has a big overhead. We introduce dynamic bound checks for every memory access. We will see in Section 2.3 how to reduce the number of checks, thus optimizing the transformation.

2.2.2 Semantics Preservation

The index sanitization transformation is straightforward, and so is its impact on semantics. The resulting environment is not touched, and the leaked trace is at least larger than the original by the two-fold evaluation of some expressions.

We first tackle this by looking at expressions in Lemma 5.

Lemma 5: Semantics preservation of index sanitization for expressions

Let σ be an environment, e an expression, t a trace, and v a value. If e evaluates to v, then $IS_e(e)$ also evaluates to v in the according environment and leaks a trace t' bigger than t, re-leaking parts of t. This can be written as

$$(e, \sigma) \downarrow^t v \implies (IS_e(e), \sigma) \downarrow^{t'} v$$

Proof Outline. By induction over e. Immediate for basic cases (x, c) and non-array cases (by induction hypothesis). In the case of $t[e]$, we know that e evaluates to a v_e within the bounds of t. Hence, $IS_e(e)$ does so too, by induction, and the transformed expression is safe, and has the same evaluation. □

We can then prove the theorem for instructions, stated in Theorem 5.

Theorem 5: Semantics preservation of index sanitization

Let σ, σ' be two environments, t a trace and s a statement, then if the evaluation of s within σ yields σ' while leaking t, the evaluation of its transformation $IS(s)$ within σ also yields σ', while leaking a t' greater than t, that is:

$$(s, \sigma) \downarrow^t \sigma' \implies (IS(s), \sigma) \downarrow^{t'} \sigma'$$

Proof Outline. By induction over s. Immediate for **skip**, as well as assignments (by Lemma 16). The induction hypothesis is enough to handle most cases. The last case is the array write. Because $t[e_1] = e_2$ can be evaluated, we know that e_1 evaluates to a v_1 within the bounds of t. By Lemma 16, $IS_e(e_1)$ does the same, and $IS_e(e_2)$ evaluates to the same value as e_2, and the behavior of the array write is preserved. □

We also show in Theorem 6 an interesting property related to index sanitization: after applying the transformation, a statement can be evaluated from any environment. This will allow us to prove in Section 2.6 that an incorrect branch of a conditional can still be evaluated if we apply index sanitization first.

Theorem 6: Safety of index sanitization

Let s be a statement, and σ be an environment. There, there exists a σ' and a t such that the transformation of s by index sanitization evaluated in σ yields σ', that is

$$(IS(s), \sigma) \downarrow^t \sigma'$$

Proof Outline. Firstly, we prove a similar lemma on expressions by induction over e. For most cases, the evaluation cannot fail, and for the memory read, the transformation ensures that $t[(0 \leq i < size(t))?i{:}0$ is always within the bounds of t. By induction over s, and similarly as the expressions version. The only complex case is the memory write, which is solved thanks to the introduction of the dynamic bound check. □

2.3 Scope-Increase

In all previous cases, the leaky memory access is within the scope of the condition h and therefore a delayed if-conversion is sufficient to remove the leaky access. If we take a look at Program P12, the current transformation is not enough because the leaky memory access $t[x]$ occurs after the condition h.

Program P12: Program with leak outside of scope

```
if h then x = l1 else x = l2;
t[l3] = l4;
y = t[x];
```

Indeed, applying delayed if-conversion would yield Program P13, which is unsecure because the predicated statement is inserted at the end of the conditionals, directly introducing secrets into x, which are then leaked by the last memory access.

Program P13: Transformed version of P 12

```
x_t = x; x_e = x;
x_t = l1; x_e = l2;
x = h?x_t:x_e;
t[l3] = l4;
y = t[x];
```

A naive solution would be to perform code motion and duplicate the offending code in both branches of the conditional. However, in our example, this has the adverse effect of moving the harmless statement $t[l_3 = l_4]$, as shown in Program P14. The semantics is still preserved, and the memory access won't be made on a secret x anymore. Unfortunately, by applying delayed if-conversion on this resulting program, we introduce a direct flow into x by predicating the assignment to $t[l_3]$. Such a direct flow could render subsequent operations using t unsecure.

Program P14: Trying a naive solution on P12

```
if (h) then
    x = l1
    t[l3] = l4;
    y = t[x];
else
    x = l2;
    t[l3] = l4;
    y = t[x];
```

Our solution is instead to mark the offending code and delay the final step of if-conversion until after that code. For example, marking the offending code in Program P12 in blue and *italicized* yields Program P15, and applying our proposed transformation would yield Program P16.

Program P15: P12 with marked code

```
if (h) then x = l1 else x = l2;
t[l3] = l4;
y = t[x];
```

The transformation treats differently y = t[x], which is marked and renamed as if within the conditional, and $t[l_3 = l_4]$, the harmless statement, left untouched by the transformation. It is not within the scope of this section, dedicated solely to the motion of problematic code, but will be addressed in the following Section 2.4.

Program P16: Trying our solution on P15

```
x_t = x; x_e = x;
x_t = l1; x_e = l2;
t[l3] = l4;
y_e = t[x_e];
y_t = t[x_t];
x = h?x_t:x_e;
y = h?y_t:y_e;
```

However, just applying this marking and renaming naively is not enough. If the problematic conditional and the marked statements are not within the same scope *i.e.*, the same block-level, as proposed in Program P17, we can't properly transform away. Indeed, keeping this code as it is would need the y=t[x] to be predicated both by h and cond.

Program P17: Leaking from another scope

```
if (cond) then
    if (h) then x = l1 else x = l2;
else
    skip;
t[l3] = l4;
y = t[x];
```

Instead, we first perform a code motion into the harmless conditional with the cond guard, yielding Program P18. Note that to ensure the preservation of the behavior of the original program, the moved code is actually duplicated in the **else** branch of the harmless conditional. Because the code in the **else** branch cannot be tainted by the problematic conditional, the moved statements are only marked in the **then** branch.

Program P18: Moving code before transformation

```
if (cond) then
    if (h) then x = l1 else x = l2;
    t[l3] = l4;
    y = t[x];
else
    skip;
    t[l3] = l4;
    y = t[x];
```

Finally, we can apply the transformation by renaming the marked code within the harmless conditional to get Program 19. This resulting program is safe and does not introduce any spurious leakage.

Program P19: Renaming after code motion

```
if (cond) then
    x_t = x; x_e = x;
    x_t = l_1; x_e = l_2;
    t[l_3] = l_4;
    y_t = t[x_t];
    y_e = t[x_e];
    x = h?x_t:x_e;
    y = h?y_t:y_e;
else
    skip;
    t[l_3] = l_4;
    y = t[x];
```

Unfortunately, such a code motion is not safe in the case of a **for** loop. If we look at Program P20 representing a simple search through an array, code motion as presented above would yield Program P21.

Program P20: A secret value search through an array

```
index = -1;
for i from 0 to 32 do
    h = t[i] == secret;
    if (h) then
        index = i;
t2[index] = 1;
```

Note that we took extra care to add a dynamic index check, to execute the moved code only in the last iteration of the loop.

Program P21: Trying our code motion solution on P20

```
index = −1;
for i from 0 to 32 do
    h = (t[i] == secret);
    if (h) then
        index = i;
    if i == 31 then
        t2[index] = 1;
```

This program, even though it looks secure at first glance, introduces a major security flaw: the value of *index* is tainted by the value of *secret* at each iteration. Indeed, trying to remove the conditional would yield Program P22, which obviously introduces a direct flow onto *index*.

Program P22: Trying to apply delayed if-conversion on P21

```
index = −1;
for i from 0 to 32 do
    h = (t[i] == secret);
    index_t = index;
    index_e = index;
    index_t = i;
    if i == 32 then
        t2[index_t] = h?1:t2[index_t];
    index = h?index_t:index_e;
```

Hence, we do not perform code motion inside **for** loops, and will instead rely on other methods we will describe in Section 2.5.

2.3.1 Prerequisites

If we want to set a formal definition of the transformation we just glimpsed, we need (i) to work out how we can identify instructions to be *marked*, and (ii) to define what it means for an instruction to be *marked*.

Program Annotations

As stated above, we aim to mark instructions deemed as *problematic*. We say that an instruction is problematic w.r.t a condition if it leaks a value depending on said condition. To keep track of leaks at a language level, we introduce a new construction to our language: annotations. To answer this need and store more information about our instructions, we annotate our programs in three different ways: (i) to identify uniquely each condition, (ii) to keep track of conditions within instructions, and (iii) to keep track of leaks linked to conditions.

The first kind of annotation, to identify uniquely each condition, requires us to define a set of program points. We denote this set $\mathbb{P}$. We can then annotate each condition with a $p \in \mathbb{P}$. A condition is now written as $\textbf{if}^p\ h\ \textbf{then}\ s_1\ \textbf{else}\ s_2$, and a program is *well-formed* if all conditions have a different identifier. We note $cond(s)$ the set of conditionals within a statement s *i.e.*, $cond(s) = \{p | \textbf{if}^p\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \in s\}$.

To keep track both of conditions and leaks within instructions, we introduce two annotations added to every instruction: *high* and *leak*. The function $high(s)$ yields the set of program points of the problematic, or secrets, conditions within s, while $leak(s)$ yields the set of program points of conditions leaked by s. These annotations only contain program points for *secret* conditions *i.e.*, ones that would cause a security issue. We will discuss how we determine these secret conditions, and annotations, in Chapter 3. For example, if we look at Program P12, and annotate the main condition with p, we have that $high(P12) = \{p\}$, $leak(P12) = \{p\}$, while $high(t[l_3] = l_4) = \emptyset$, $leak(t[l_3] = l_4) = \emptyset$ and $leak(y = t[x]) = \{p\}$.

Reminder: Program P12

```
if h then x = l1 else x = l2;
t[l3] = l4;
y = t[x];
```

Indeed, P12 contains only the condition p, while the subsequent instructions are atomic, therefore not containing any conditional. Furthermore, $y = t[x]$ leaks the value of x, which is directly tied to the p condition, while $t[l_3] = l_4$ leaks only "public" values. These annotations allow us to identify secret conditionals: a condition $\textbf{if}^p\ h\ \textbf{then}\ s_1\ \textbf{else}\ s_2$ is secret if and only if $p \in high(\textbf{if}^p\ h\ \textbf{then}\ s_1\ \textbf{else}\ s_2)$. To store these values directly into

the language, we actually annotate the instructions in the following way:

$$(s)^r_g \text{ where } g = high(s) \text{ and } r = leak(s)$$

A new updated version of the language can be found in Definition 22 at the end of this section.

Next construction

The annotations we just defined will allow us to identify which instructions are to be marked. However, we still need to present a way to mark any problematic code. By noting that our transformation is actually moving code into the scope of the condition, just in a smarter way, we introduce a new construction: **next**. This syntax comes as an add-on to the **if** syntax: **if** h **then** s_1 **else** s_2 **next** s_3. The **next** block is an extension of the scope of the conditional, containing a part of its continuation such that delayed if-conversion will only apply its final step after it. Semantically, a condition with a **next** block is evaluated just as (**if** h **then** s_1 **else** s_2); s_3. We can move all code preceding the last instruction to be marked into the **next** block to keep track of all marked instructions. Non-marked statements will be identified and handled differently during the transformation. For example, rewriting Program P15 (the colored version of P12) with this construction would yield Program P23.

Program P23: Rewriting of P15 with next

```
if (h) then x = l1 else x = l2 next (t[l3] = l4; y = t[x];)
```

For the sake of simplifying the presentation of our code, we write **if** h **then** s_1 **else** s_2 to avoid writing **if** h **then** s_1 **else** s_2 **next skip**.

Moreover, since we now have a way to know which memory accesses are gonna be affected by our introduction on predicated code, namely the ones within one of the branches of an unsecure conditions (may the branch be **then**, **else** or even **next**), we can restrict the application of index sanitization to theses branches only. This allows us to drastically reduce the overhead of the transformation. We call IS_p the restricted transformation. We have, for a given program point p,

$$IS_p(\textbf{if}^p\ h\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3) = \textbf{if}^p\ h\ \textbf{then}\ IS(s_1)\ \textbf{else}\ IS(s_2)\ \textbf{next}\ IS(s_3)$$

For all other statements, the IS_p function is called recursively, ensuring that we only transform memory accesses within the p conditional.

Updated Definitions

If we combine both new additions presented above, we can redefine L to be the one defined in Definition 22. We introduce a new intermediate representation of non-annotated statements to allow for a more comprehensible definition. Given that **next** is an intermediary construction, destined to be removed, we add a new syntactic constraint to input programs: **next** blocks are to be equal to **skip**. Moreover, we now say that a program point p is within a statement s (noted $p \in s$) if either s is the p conditional, or a parent instruction of the p conditional.

Definition 22: New syntax of L

Expressions:

$$expr \ni e \quad ::= \quad x \mid c \mid e_1 \oplus e_2 \mid e_1?e_2{:}e_3 \mid t[e]$$

Instructions:

$$\begin{aligned} stmt' \ni s' \quad ::= \quad & \textbf{skip} \\ & \mid x = e \\ & \mid t[e_1] = e_2 \\ & \mid s_1; s_2 \\ & \mid \textbf{if}^p\ x\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3 \\ & \mid \textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ s \end{aligned}$$

Annotated instructions:

$$stmt \ni s \quad ::= \quad (s')^r_g$$

Along with the definition of the updated language, we update the definition of our semantics to be the ones presented in Definition 23. Note that semantics are presented for the intermediate representation of our statements, without annotation. The new semantic rule for **if** is computed by chaining the **if** rule with the sequence rule, as stated previously.

Definition 23: New leaky semantics for instructions

$$\frac{}{(\textbf{skip}, \sigma) \downarrow^{\epsilon} \sigma} \qquad \frac{(e, \sigma) \downarrow^{t} v}{(x = e, \sigma) \downarrow^{t} \sigma[x \mapsto v]} \qquad \frac{(s_1, \sigma_1) \downarrow^{t_1} \sigma_2 \quad (s_2, \sigma_2) \downarrow^{t_2} \sigma_3}{(s_1; s_2, \sigma_1) \downarrow^{t_1 \cdot t_2} \sigma_3}$$

$$\frac{(e_1, \sigma) \downarrow^{t_1} i \quad 0 \leq i < size(t) \quad (e_2, \sigma) \downarrow^{t_2} v}{(t[e_1] = e_2, \sigma) \downarrow^{t_1 \cdot t_2 \cdot i} \sigma[t \mapsto \sigma[t][i \mapsto v]]}$$

$$\frac{(x, \sigma) \downarrow^{\epsilon} b \quad (s_b, \sigma) \downarrow^{t} \sigma' \quad (s, \sigma') \downarrow^{t'} \sigma''}{(\textbf{if}^{p}\ x\ \textbf{then}\ s_{true}\ \textbf{else}\ s_{false}\ \textbf{next}\ s, \sigma) \downarrow^{b \cdot t \cdot t'} \sigma''}$$

$$\frac{\forall_{i \in [c_1; c_2]} (x = i; s, \sigma_i) \downarrow^{t_i} \sigma_{i+1} \quad t = t_{c_1} \cdot \dots \cdot t_{c_2}}{(\textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ s, \sigma_{c_1}) \downarrow^{t} \sigma_{c_2+1}}$$

2.3.2 Formal Description

The purpose of the *scope increase* algorithm is to identify a conditional branching, say identified by the program point p, and to confine inside the **next** statement of the conditional all the memory accesses that are indirectly leaking the guard of p.

Condition Selection

As observed at the start of this section, duplicating code in both branches of a conditional may introduce spurious information leaks. To avoid this issue, we select the *outermost*, *rightmost* secret conditional *i.e.*, a secret conditional which is not within the syntactic scope of another secret conditional and is the last in the textual order. Consider for example the Program P24, where the statement s indirectly leaks both h_1 and h_2 *i.e.*, $p_0, p_1 \in high(s)$.

Program P24: Example of a misselection

```
if@p0 (h0) then
    if@p1 (h1) then x = 0;
    else x = 1;
else skip;
s
```

If we select the innermost conditional p_1 to first do our code motion, it would yield Program P25, where the statement s is duplicated and crosses the boundaries of h_1. This actually bypasses the entire role of the **next** block for condition p_0, and we get the same issue as presented above.

Program P25: Wrong selection in Program P24

```
if@p0 (h0) then
    if@p1 (h1) then x = 0;
    else x = 1;
    next s;
else s;
```

On the contrary, if we select the outermost conditional p_0, the code motion yields Program P26, which is secure. After full transformation, and removal, of p_0, we will be able to select and transform p_1.

Program P26: Right selection in Program P24

```
if@p0 (h0) then
    if@p1 (h1) then x = 0;
    else x = 1;
else skip;
next s;
```

We formalize this in Definition 24 by creating a predicate $RO_p(s)$ which ensures that p is the rightmost, outermost secret conditional of statement s.

Definition 24: Predicate RO_p

$$\frac{p \in g}{RO_p((\textbf{if}^p\ h\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3)^r_g)} \qquad \frac{RO_p((s_2)^{r_2}_{g_2})}{RO_p((s_1)^{r_1}_{g_1}; (s_2)^{r_2}_{g_2})} \qquad \frac{RO_p((s_1)^{r_1}_{g_1})}{RO_p((s_1)^{r_1}_{g_1}; (s_2)^{r_2}_{\emptyset})}$$

$$\frac{p' \notin g \quad RO_p(s_3)}{RO_p((\textbf{if}^{p'}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3)^r_g)} \qquad \frac{p' \notin g \quad high(s_3) = \emptyset \quad RO_p(s_2)}{RO_p((\textbf{if}^{p'}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3)^r_g)}$$

$$\frac{p' \notin g \quad high(s_3) = \emptyset \quad high(s_2) = \emptyset \quad RO_p(s_1)}{RO_p((\textbf{if}^{p'}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3)^r_g)} \qquad \frac{RO_p(s)}{RO_p(\textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ s)}$$

If s is simply a conditional containing itself as a secret conditional, it is the rightmost outermost one. Otherwise, we check if the **next** block, the **else** block, or the **then** block contains any secret conditional, in this order, and pass it as our current rightmost outermost. If we face a sequence of statements $s_1; s_2$, we first check s_2 and then s_1 for a rightmost outermost conditional. Finally, in the case of a **for** loop, we recursively look at the body of the loop.

Introduction of next

To perform code motion, and once a condition p has been identified as the rightmost, outermost one, we need (i) to identify the continuation of p in the program, and (ii) to identify the problematic part of the continuation. The first step is directly computed by syntax analysis, and the second requires the use of the annotation we defined earlier. For this purpose, we define a function $sep_p(s)$ which, given a program point p and a continuation s, compute the set of all pairs (s_l, s_s) (resp. *leaking* and *safe*) such that s_s does not leak p, and s is semantically equivalent to the sequence of s_l and s_s, that is:

$$(s_l, s_s) \in sep_p(s) \iff p \notin leak(s_s) \wedge s_l; s_s \simeq s$$

Such a function is defined in Definition 25.

Definition 25: Separation function

$$\frac{(s_l, s_s) \in sep_p(s) \quad p \notin leak(t)}{(s_l, s_s; t_s) \in sep_p(s; t)} \qquad \frac{(t_l, t_s) \in sep_p(t)}{(s; t_l, t_s) \in sep_p(s; t)}$$

$$\frac{p \in leak(s)}{(s, \mathbf{skip}) \in sep_p(s)} \qquad \frac{p \notin leak(s)}{(\mathbf{skip}, s) \in sep_p(s)}$$

If our continuation is a sequence $(s; t)$ and t is not leaking p, we can use the leaking part of s as the leaking part of $(s; t)$, and the safe components of s and t as the safe component of $(s; t)$ for the purpose of the *sep* function. Otherwise, if t is leaking p, only the safe section of t is kept as the safe section of the sequence, and the leaking one is the sequence of s and the leaking component of t. On the other hand, if the statement is not a sequence, it is either leaking or safe, and its counterpart is **skip**.

Given a condition of program point p, thanks to RO_p, and with the help of the separation function *sep* to isolate problematic statements, we can define the scope-increase algorithm $SI_p : stmt \times stmt \to stmt$. It takes two parameters: a statement s_1 such that $RO_p(s_1)$ and another statement s which is the continuation of s_1 *i.e.*, the statement to be executed after s_1. The algorithm performs code motion until inserting the motion code within the **next** statement of the p conditional. In the case of a **for** loop, code motion is performed as much as possible, without compromising the integrity of the loop. To run the algorithm on a whole program P, we find p such that $RO_p(P)$, and call $SI_p(P)$. The rules for the scope-increase algorithm are defined in Definition 26. Note that the scope-increase algorithm is always called on programs where all **next** blocks are empty, or **skip**. This is true for the input program as a syntactic constraint and is preserved by the whole transformation, as it will be described in Section 2.6. As a result, following a call to the scope-increase algorithm, the program contains only one **next** block, within the p condition.

Definition 26: Scope Increase Algorithm

$$
\begin{array}{lcl}
SI_p(\textbf{if}^p\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2, s) & = & \textbf{if}^p\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s \\
SI_p(s_1; s_2, s) & = & s_1; SI_p(s_2, s) \quad \textit{if } p \in s_2 \\
SI_p(s_1; s_2, s) & = & SI_p(s_1, s_2; s) \quad \textit{if } p \notin s_2 \wedge s \neq \textbf{skip} \\
SI_p(s_1; s_2, \textbf{skip}) & = & \textit{let } s_l, s_s \in sep_p(s_2) \textit{ in } SI_p(s_1, s_l); s_s \\
 & & \quad \textit{if } p \notin high(s_2) \\
SI_p(\textbf{if}^{p'}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2, s) & = & \textbf{if}^{p'}\ c\ \textbf{then}\ (s_1; s)\ \textbf{else}\ SI_p(s_2, s) \\
 & & \quad \textit{if } p \in s_2 \wedge p \neq p' \\
SI_p(\textbf{if}^{p'}\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2, s) & = & \textbf{if}^{p'}\ c\ \textbf{then}\ SI_p(s_1, s)\ \textbf{else}\ (s_2; s) \\
 & & \quad \textit{if } p \in s_1 \wedge p \neq p' \\
SI_p(\textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ s_1, s) & = & \textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ SI_p(s_1, \textbf{skip}); s
\end{array}
$$

If the statement s_1 is the conditional we are looking for, we insert the continuation s into the **next** statement. If the statement is a sequence of the form $s_1; s_2$, there are two cases depending on whether p is within the conditions of s_1 or s_2. If p is within s_2 *i.e.*, $p \in high(s_2)$, the statement s_1 is kept unchanged and SI_p is recursively called over s_2. Otherwise, the continuation of s is augmented by s_2 and SI_p is recursively called over s_1, $s_2; s$ being the continuation of s_1. However, if the continuation is **skip**, we can optimize and split s_2 into a pair of statements s_l and s_s, such that $(s_l, s_s) \in sep_p(s_2)$. Hence, s_s does not leak p, and SI_p is recursively called over s_1 with a reduced continuation s_l which contains all the statements of s_2 that may leak p. This is the step that actually does the selection of the code to be moved. We left in s_r all public, non-problematic code, and s_l contains all statements up to the last potentially dangerous one. If the statement is instead another conditional with annotation $p' \neq p$, there are two symmetric cases depending on whether the p conditional is located in the **then** branch (*i.e.*, $p \in high(s_1)$) or in the **else** branch (*i.e.*, $p \in high(s_2)$). Without loss of generality, consider $p \in s_2$. In that case, the continuation s is appended to the statement s_1 of the **then** branch and we call recursively SI_p over the statement s_2 of the **else** branch. For a **for**, as said earlier, we prevent any code motion inside the loop. As a result, $scope - increase$ is recursively called over the loop body s_1 with the continuation **skip**.

2.3.3 Semantics Preservation

The sole purpose of scope-increase is first to identify a problematic conditional and its continuation, and second to perform code motion to ensure that any problematic statements w.r.t to this conditional are within its scope. Thus, we first prove in Lemma 6 that given a statement s, using the function *sep* to identify the leaking part of s does not affect its semantics.

Lemma 6: Semantics preservation of *sep*

Given a statement s, two environments σ, σ' and a trace t such that $(\sigma, s) \downarrow^t \sigma'$, the sequence s_l, s_s with $(s_l, s_s) \in sep(s)$ has the same semantics as s, that is $(\sigma, s_l; s_s) \downarrow^t \sigma'$

Proof Outline. By induction over s. If s is not a sequence, either s_l or s_s is a **skip**, while the other is s, and we conclude by the semantics of a sequence. Otherwise, by induction, and the sequence rule. □

Once the continuation of a conditional has been found using the *sep* function, we prove in Theorem 7 that performing the code motion is harmless, and have no effect either on the semantics of the program. The intuition of why it is true is because the **next** block evaluates just as a sequence.

Theorem 7: Semantics preservation of SI_p

Given two statements s, s' and a program point p such that $RO_p(s)$, the transformed version $SI_p(s, s')$ has the same semantics as its initial counterpart $s; s'$. That means for any two environments σ, σ' and trace t such that $(\sigma, s; s') \downarrow^t \sigma'$, we have $(\sigma, SI_p(s, s')) \downarrow^t \sigma'$.

Proof Outline. By induction over s. If s is the conditional we are looking for, by definition of **next**. Otherwise, if s is a sequence, by induction, and Lemma 6. □

2.4 Updated Delayed If-Conversion

Because we changed the language in the previous Section, we need to revisit the previously defined transformation. If the index sanitization one is defined locally and is not impacted by this change (including its preservation proof), the delayed if-conversion is invalidated both by the presence of annotations and the introduction of the **next** statement.

Firstly, because we now have a way to precisely identify conditions, the transformation becomes local instead of global, and we only transform one condition at a time: the one identified by RO_p in the scope-increase pass. Secondly, as shown in the previous Section, the **next** statement needs particular care in its renamings. In fact, the block contains both problematic and innocuous statements. We need to keep the harmless ones as is and rename the problematic ones using renaming maps from both branches. This also means a change to the final step of the transformation, the merging of the two maps.

The way we rename instructions within the **next** block is by duplicating and renaming the statement on the fly. When renaming, we use a fresh variables policy akin to SSA (Single Static Assignment) to generate new renamings and avoid variable name clashes. For example, if we take a look at Programs P23 and P16, the transformation renamed y into two fresh variables y_t and y_e as to duplicate the memory write, and ignored the harmless statement.

Reminder: Program P23

```
if (h) then x = l1 else x = l2 next (t[l3] = l4; y = t[x];)
```

Reminder: Program P16

```
xt = x; xe = x;
xt = l1; xe = l2;
t[l3] = l4;
ye = t[xe];
yt = t[xt];
x = h?xt:xe;
y = h?yt:ye;
```

If we look at the more complicated Program P27, the value of y is overwritten by a harmless one, and the transformation should not duplicate the first memory write, the one on y.

Program P27: Program containing a change of security on a variable

```
if (h) then
    x = l1;
    y = l2;
else
    x = l3;
    y = l4;
next
    y = 5;
    w = t[y];
    z = t[x];
```

Hence, transforming this one requires *ignoring* the $y = 5$ instruction, that is not duplicating it, and instead proposing a new, unique, renaming for the y variable. This transformation is shown in Program 28.

Program P28: Program containing a change of security on a variable

```
x_t = x;  x_e = x;
y_t = y;  y_e = y;
x_t = l1;  y_t = l2;
x_e = l3;  y_e = l4;
y_n = 5;
w_n = t[y_n];
z_t = t[x_t];  z_e = t[x_e];
x = h?x_t:x_e
y = y_n;  w = w_n;
z = h?z_t:z_e;
```

Note that we also introduced new names for the z and w variables, following that SSA-like policy, to avoid any potential clashes and allow code duplication. The final step has also been changed to account for any new renaming, especially *unique* ones *i.e.*, w_n and y_n.

2.4.1 Prerequisites

Because the transformation changes the renaming maps dynamically, we need to introduce new operations on them. In particular, we need to be able to join them and create a statement merging them.

Join

Joining two renaming maps, noted $\rho_1 \bowtie \rho_2$, is akin to returning the renaming if both maps agree, or a fresh variable if they disagree. This is formalized in Definition 27.

Definition 27: Joining two renaming maps

Let ρ_1 and ρ_2 be two renaming maps. We say that $\rho_1 \bowtie \rho_2$ is the join of ρ_1 and ρ_2 if its renamings are equivalent to these of both maps xor to a fresh variable, that is:

$$\forall x, \rho_1 \bowtie \rho_2(x) = \begin{cases} \rho_1(x) & \text{if } \rho_1(x) = \rho_2(x) \\ x' \text{ where } x' \text{ is a fresh variable} & \text{otherwise} \end{cases}$$

Phi-merging

Because we create new bifurcation in the simultaneous execution tree, we also need to be able to merge renamings along the way. This is done using ϕ-merging, akin to the ϕ nodes in SSA. We define what a ϕ-merging is in Definition 28.

Definition 28: ϕ-merging

Let ρ_1 and ρ_2 be two renaming maps. We say that $\phi(\rho_1, \rho_2)$ is a ϕ-merging of ρ_1 and ρ_2 if it is a statement assigning to all renaming by ρ_1 the renaming of the renamed variable by ρ_2, that is

$$\phi(\rho_1, \rho_2) = seq(\{\rho_1(x) = \rho_2(x) | x \in \mathcal{V}^{\rho_1} \wedge \rho_1(x) \neq \rho_2(x)\})$$

After such a statement, we effectively renamed all ρ_2 renamings to ρ_1 ones.

2.4.2 Formal Description

Because of all these additions, we have to redefine the DIC^{V_P} function, although we can use most of our already-defined tools. In particular, because of the now local

characteristic of the transformation, Definition 14 needs to be updated. Similarly, we need to update Definition 17 to take into account the peculiar renaming of the **next** block. Finally, the merge step has to consider the new renamings introduced by the **next** renaming.

The change in the definition of DIC^{V_P} is actually quite straightforward: we add a rule to handle differently the condition we seek to remove, and other instruction are treated as they were in Section 2.1. An updated definition can be found in Definition 29. The function is now parameterized by p, the transformation being local to this condition. If the condition we encounter is not the one we seek to transform, we indeed pass the transformation along to the two branches. Note that because we know that only the p condition has a non-skip **next** branch, we ignore the third block in this transformation for most cases.

Definition 29: New formal definition of Delayed If-Conversion

$$
\begin{array}{lcl}
DIC_p^{V_P}(\textbf{skip}) & = & \textbf{skip} \\
DIC_p^{V_P}(x = e) & = & x = e \\
DIC_p^{V_P}(t[e_1] = e_2) & = & t[e_1] = e_2 \\
DIC_p^{V_P}(s_1; s_2) & = & DIC_p^{V_P}(s_1); DIC_p^{V_P}(s_2) \\
DIC_p^{V_P}(\textbf{if}^p\ h\ \textbf{then}\ s_t\ \textbf{else}\ s_e\ \textbf{next}\ s_n) & = & DICIF_p^{V_P}(\textbf{if}^p\ h\ \textbf{then}\ s_t\ \textbf{else}\ s_e\ \textbf{next}\ s_n) \\
DIC_p^{V_P}(\textbf{if}^{p'}\ h\ \textbf{then}\ s_t\ \textbf{else}\ s_e) & = & \textbf{if}^{p'}\ h\ \textbf{then}\ DIC_p^{V_P}(s_t)\ \textbf{else}\ DIC_p^{V_P}(s_e) \\
DIC_p^{V_P}(\textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ s) & = & \textbf{for}\ x\ \textbf{from}\ c_1\ \textbf{to}\ c_2\ \textbf{do}\ DIC_p^{V_P}(s)
\end{array}
$$

where $p \neq p'$

Simultaneous Renaming of next

As stated, the $DICIF^{V_P}$ function is actually quite close to its $DICIF^{V_P}$ counterpart. The sole changes are in the addition of a *Nxt* function to rename specifically the **next** block, and an upgrade to the crafting of the *post* statement.

The function *Nxt* is parameterized by the guard h of the condition, and by the two renaming maps for each branch, ρ_t and ρ_e. This is written as $Nxt_h^{\rho_t,\rho_e} : stmt \to rmap \times rmap \times stmt$, a function taking as argument a statement corresponding to a **next** block, and yielding the updated renaming maps and the transformed statement, after renaming. Because the transformation needs to be simultaneous for both branches, the expressions of

the function *Nxt* are arguably more complex than its DIC^{V_P} counterpart. This definition is divided into two parts: one in Definition 30 containing the renaming of the atomic instructions, and one in Definition 31, containing sequences, conditionals, and loops.

Definition 30: Renaming a next block - Assignments

We define $\mathcal{V}_{fresh}$ to be an arbitrary big set of variables fresh from all variables used in the initial program, and in the current renamings.

$$Nxt_h^{\rho_t,\rho_e}(x = v) = \begin{cases} (\rho_t[x \mapsto x'], \rho_e[x \mapsto x'], x' = \rho_e(v)) \\ \quad \text{if } \rho_t(v) = \rho_e(v) \\ (\rho_t[x \mapsto x_t], \rho_e[x \mapsto x_e], x_t = \rho_t(v); x_e = \rho_e(v)) \\ \quad \text{otherwise} \end{cases}$$

where x_t, x_e and x' are sampled from $\mathcal{V}_{fresh}$

$$Nxt_h^{\rho_t,\rho_e}(t[e_1] = e_2) = \begin{cases} (\rho_t, \rho_e, t[\rho_t(e_1)] = \rho_t(e_2)) \\ \quad \text{if } \bigwedge \begin{array}{l} \rho_t(e_1) = \rho_e(e_1) \\ \rho_t(e_2) = \rho_e(e_2) \end{array} \\ \left(\rho_t, \rho_e, \left(\begin{array}{l} t[\rho_t(e_1)] = h?\rho_t(e_2):t[\rho_t(e_1)] \\ t[\rho_e(e_1)] = !h?\rho_e(e_2):t[\rho_e(e_1)] \end{array} \right)\right) \\ \quad \text{otherwise} \end{cases}$$

We use the renamed values as a way to track the *secrecy* of a variable. If the renaming maps yield the same renaming: the to-be-assigned value is not dependent on the guard, and both executions would yield the same. Otherwise, the resulting value would be directly linked to the branch executed, it is potentially leaking, and the renamings are distinct. As such, when trying to rename a simple assignment $x = v$, the behavior of the *Nxt* function changes whether or not the renamings are equal. If they are, we introduce a new unique renaming x', update both maps ρ_t and ρ_e to use this name from now on, and make the assignment on it, without the need to duplicate the statement. Otherwise, we pick two new names, duplicate the statement, and virtually create two different, simultaneous executions: the **then** branch with x_t, and the **else** branch with x_e. Note that this mechanism works whether or not the variable was duplicated before. This allows us to merge previously problematic variables that became harmless, or secure the ones that are not innocuous anymore. The same reasoning is applied to an array update, where, instead of generating one or two new names, the choice is made on whether or not

we predicate the statement. If the update poses a security threat, we duplicate it and update both simultaneous executions by writing them as a **cmove** such that the update is dummy unless the branch is executed. Because a memory write does not overwrite any variables, we do not need to update the renaming maps.

Definition 31: Renaming a next block - Complex statements

$$
\begin{aligned}
Nxt_h^{\rho_t,\rho_e}(s_1; s_2) \quad &= \quad let\ (\rho'_t, \rho'_e, s'_1) = Nxt_h^{\rho_t,\rho_e}(s_1)\ in \\
&\quad\ \ let\ (\rho''_t, \rho''_e, s'_2) = Nxt_h^{\rho_t,\rho_e}(s_2)\ in \\
&\quad\ \ (\rho''_t, \rho''_e, s'_1; s'_2) \\
Nxt_h^{\rho_t,\rho_e}(\mathbf{if}^{p'}\ h'\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2) \quad &= \quad let\ (\rho^1_t, \rho^1_e, s'_1) = Nxt_h^{\rho_t,\rho_e}(s_1)\ in \\
&\quad\ \ let\ (\rho^2_t, \rho^2_e, s'_2) = Nxt_h^{\rho_t,\rho_e}(s_2)\ in \\
&\quad\ \ let\ \dot{\rho}_t = \rho^1_t \bowtie \rho^2_t\ and\ \dot{\rho}_e = \rho^1_e \bowtie \rho^2_e\ in \\
&\quad\ \ let\ s_{\phi^1_t} = \phi(\dot{\rho}_t, \rho^1_t)\ and\ s_{\phi^1_e} = \phi(\dot{\rho}_e, \rho^1_e)\ in \\
&\quad\ \ let\ s_{\phi^2_t} = \phi(\dot{\rho}_t, \rho^2_t)\ and\ s_{\phi^2_e} = \phi(\dot{\rho}_e, \rho^2_e)\ in \\
&\quad\ \ let\ r_{then} = (s'_1; s_{\phi^1_t}; s_{\phi^1_e})\ in \\
&\quad\ \ let\ r_{else} = (s'_2; s_{\phi^2_t}; s_{\phi^2_e})\ in \\
&\quad\ \ (\dot{\rho}_t, \dot{\rho}_e, \mathbf{if}^{p'}\ \rho_t(h')\ \mathbf{then}\ r_{then}\ \mathbf{else}\ r_{else}) \\
Nxt_h^{\rho_t,\rho_e}(\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s) \quad &= \quad let\ \rho'_t = \rho_t[x \mapsto x' \mid x' \in \mathcal{V}_{fresh} \wedge x \in mod(s)]\ in \\
&\quad\ \ let\ \rho'_e = \rho_e[x \mapsto x' \mid x' \in \mathcal{V}_{fresh} \wedge x \in mod(s)]\ in \\
&\quad\ \ let\ (\rho''_t, \rho''_e, s') = Nxt_h^{\rho'_t,\rho'_e}(s)\ in \\
&\quad\ \ let\ s'_t = \phi(\rho'_t, \rho_t)\ and\ s'_e = \phi(\rho'_e, \rho_e)\ in \\
&\quad\ \ let\ s''_t = \phi(\rho'_t, \rho''_t)\ and\ s''_e = \phi(\rho'_e, \rho''_e)\ in \\
&\quad\ \ let\ s_b = (s'; s''_t; s''_e)\ in \\
&\quad\ \ (\rho'_t, \rho'_e, s'_t; s'_e; \mathbf{for}\ \rho_t(i)\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s_b)
\end{aligned}
$$

For the sequence, both statements are renamed and the renamings are threaded along. For the conditional, by construction, we have the guarantee that there is no **next** statement. This is because the only **next** in the program has just been introduced by the SI_p transformation and we are currently processing the generated **next** statement. Both branches are recursively renamed using the same initial renaming maps ρ_t and ρ_e. At the end of the conditional, to reconcile the renaming ρ^1_t and ρ^2_t (resp. ρ^1_e and ρ^2_e) we join the renaming maps $\dot{\rho}_t = \rho^1_t \bowtie \rho^2_t$ (resp. $\dot{\rho}_e = \rho^1_e \bowtie \rho^2_e$). To synchronize the program variables with the renaming maps $\dot{\rho}_t$ and $\dot{\rho}_e$, we append to each of the branches a sequence of

assignments using the ϕ-merging. For the **for** loop, before renaming the loop body, we update the initial renaming maps ρ_t and ρ_e so that each variable of the loop body is given a fresh variable. The loop body s is renamed using the obtained renaming maps ρ'_t and ρ'_e. In order to synchronize the renaming maps with the program variables, we insert ϕ-merging before and after the renaming of the loop body s'. This is needed to ensure that the variable names are coherent for the next loop iteration.

Finalization

In our first version of delayed if-conversion, the final merging was done by a simple sequence of statements *post*. This statement was assigning to each variable renamed a conditional choice between both renamings to ensure the correct semantics of the transformation. However, we now have two possible cases: a variable has either two renamings or a unique one. We thus need to produce a more complex *post* statement, doing a straight assignment for the safe variables, with only one renaming, and a predicated assignment for the problematic ones. This leads to a definition of *post* as

$$post = seq(\{x = \left\{ \begin{array}{ll} \rho_t(x) & \text{if } \rho_t(x) = \rho_e(x) \\ h?\rho_t(x){:}\rho_e(x) & \text{otherwise} \end{array} \right. \Bigg| \; x \in \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}\})$$

We can then define what the delayed if-conversion of a problematic condition p yields. By first creating the two renaming maps ρ_t and ρ_e, initializing their renamed names, renaming both the **then** and the **next** branched, renaming simultaneously the **next** branch, and finally merging back the renamings together with the *post* statement, we are able to remove the condition. The definition of this exact step is shown in Definition 32.

Definition 32: Delayed if-conversion of a condition with a next

$$\frac{\begin{array}{c} \rho_t = init(mod(s_t), V_p) \quad \rho_e = init(mod(s_e), V_p \cup \mathcal{V}_{\rho_t}) \\ pre_t = seq(\{\rho_t(x) = x \mid x \in \mathcal{V}^{\rho_t}\}) \quad pre_e = seq(\{\rho_e(x) = x \mid x \in \mathcal{V}^{\rho_e}\}) \\ Nxt_h^{\rho_t,\rho_e}(s_n) = (\dot{\rho}_t, \dot{\rho}_e, s'_n) \\ post = seq(\{x = \left\{ \begin{array}{ll} \rho_t(x) & \text{if } \rho_t(x) = \rho_e(x) \\ h?\rho_t(x){:}\rho_e(x) & \text{otherwise} \end{array} \right. \Bigg| \; x \in \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}\}) \end{array}}{DICIF_p^{V_P}(\textbf{if}^p\ h\ \textbf{then}\ s_t\ \textbf{else}\ s_e\ \textbf{next}\ s_n) = pre_t; pre_e; Rn_h^{\rho_t}(s_t); Rn_{\neg h}^{\rho_e}(s_e); s'; post}$$

2.4.3 Semantics Preservation

Because the definitions of ρ_t, ρ_e, pre_t, and pre_e are left unchanged, all lemmas proved in Section 2.1 are still valid. And, to handle the **next** transformation, we suppose that the statement s_n evaluates both from two different environments. One being the result of evaluating the correct branch, the second being the result of evaluating the incorrect one, albeit first transformed by index sanitization. Then, if we have a combination of both renamed environments, the **next** renaming can be evaluated and results in the updated renaming of both environments, as shown in Lemma 7

Lemma 7: Semantics Preservation of next renaming

Given a statement s, two renaming maps ρ_t, ρ_e, a guard h, two traces t_t, t_e, and some environments σ_t, σ'_t, σ_e and σ'_e such that s evaluated in σ_t yields σ'_t (*i.e.*, $(s, \sigma_t) \downarrow^{t_t} \sigma'_t$), and s evaluated in σ_e yields σ'_e. Then, the **next** renaming of s can be evaluated within the renaming of σ by ρ_t and ρ_e, that is :

$$(\rho'_t, \rho'_e, s') = Nxt_h^{\rho_t,\rho_e}(s) \implies (\rho_t(\rho_e(\sigma_i, \sigma_e), \sigma_t), s') \downarrow^{t'} \rho'_t(\rho'_e(\sigma'_i, \sigma'_e), \sigma'_t)$$

where t' is larger than $t_t \cdot t_e$ due to the multiple evaluations of memory accesses, and σ'_i is σ_i where the arrays are updated from values in σ'_t if h is true, and values from σ'_e otherwise.

Proof Outline. First, note that the statement defined as $\phi(\rho_1, \rho_2)$ harmonizes the environment $\rho_2(\sigma)$ to $\rho_1(\sigma)$. Then, by induction over s. For the assignment, the new assignment is just a renaming, and preserves the value, in both cases. For the memory write, if h is true, the value of t is updated the same way as in σ'_t. Otherwise, it is updated as in σ'_e. The other cases are by induction hypothesis, and using the property about ϕ. □

This lemma about the **next** transformation means that we simultaneously continue both executions, those of the correct and incorrect branches. At the end, we merge both executions using the *post* statement. The effect of this merging statement is that we only keep the environment of the correct branch, discarding all incorrect renamings, as stated in Lemma 8.

Lemma 8: Semantics of the merging statement

Let ρ_t, ρ_e be two renamings, and $\sigma_i, \sigma_t, \sigma_e$ be three environments. Then, the merging of ρ_t and ρ_e evaluated in the renaming of σ_t and σ_e yields σ_t if h is true, and σ_e otherwise.

Proof Outline. The proof is by induction over the set of renamed variables. For each renamed variable, we assign to it the renaming from the correct branch (if both renaming are equals, we can prove that their renamings come from the **next** renaming and that their values are equal). Thus, the resulting environment is that of the correct branch. □

Finally, we can state the semantics preservation for the whole $DIC_p^{V_P}$ transformation. Given a conditional p transformed by index sanitization *i.e.*, where all branches can be evaluated whatever the environment is, the transformation of the conditional p by the $DICIF_p^{V_P}$ has the same behavior as the branch, only without leaking the guard. This theorem is immensely simplified by the assumption that we only look at *interesting* variables in environments. See Chapter 4 for more detailed proofs using the same kind of reasoning on typing environments.

Theorem 8: Semantics preservation of $DICIF_p^{V_P}$

Let $s = \textbf{if}^p\ h\ \textbf{then}\ s_t\ \textbf{else}\ s_e\ \textbf{next}\ s_n$ be the p conditional, previously transformed by index sanitization. If there is σ, σ', h, such that $(s, \sigma) \downarrow^t \sigma'$, then we have

$$(DICIF_p^{V_P}(s), \sigma) \downarrow^{t'} \sigma'$$

where, if we note $t = h \cdot t_s$, we have t' greater than t_s due to some double evaluations of memory access, but without $h \in t_s$.

Proof Outline. By Lemma 2, we know that the *pre*s statements rename σ by ρ_t and ρ_e. By Lemma 6, we know that we can evaluate the incorrect branch, and propagate the renamings using Lemma 4. And, we continue both executions by Lemma 7, using here too Lemma 6 to ensure that the **next** branch can be evaluated in both branches. We conclude by Lemma 8 to merge back both renamings to the correct evaluation. The guard h is removed from the trace by the removal of the conditional. □

Per the definition of $DIC_p^{V_P}$, we can use Theorem 8 to prove that the whole transformation is safe and preserves the semantics.

2.5 Handling direct leaks

There are some leaks that our current transformation is unable to mitigate. In particular, because we forbid any code motion within a loop, any memory access outside a loop depending on a condition within it would still be problematic even after the delayed if-conversion. To allow a full transformation, and a program clean of any problematic statement, we apply the naive transformation presented in Section 1.4 to delete those unsecure accesses. We do not have to worry about conditions: they should be handled by the transformation anyway.

One way to remove those accesses would be to wait until after our transformation is done, and then remove any remaining leaking access. However, let's say that we have a way to identify from the get-go such accesses, we could then use a preprocessing pass to apply this naive, costly transformation. We will see how we are effectively able to mark these leaking statements using a type system in Chapter 3. For now, we assume that we have a clean-up pass that applies this transformation to any remaining issue.

2.5.1 Semantics Preservation

For this clean-up pass to be effective, it needs to preserve the behavior of the program. Thus, we prove in Theorem 9 that the array traversal transformation preserves the semantics of the transformed statement.

Theorem 9: Semantics preservation of array traversal

Let t be an array and x, y be two variables. Then, if there exist σ, t, σ' such that $(\sigma, y = t[x]) \downarrow^t \sigma'$, then $(\sigma, \textbf{for } i \textbf{ from } 0 \textbf{ to } size(t) - 1 \textbf{ do } y = (x == i)?t[i]{:}y) \downarrow^{t'} \sigma'$ where $t' = 0 \cdot 1 \cdots size(t)$.

Similarly, if $(t[x] = y], \sigma) \downarrow^t \sigma'$, then $(\sigma, \textbf{for } i \textbf{ from } 0 \textbf{ to } size(t) - 1 \textbf{ do } t[i] = (x == i)?y{:}t[i]) \downarrow^{t'} \sigma'$ where $t' = 0 \cdot 1 \cdots size(t)$.

Proof outline. The assignment $y = (x == i)?t[i]{:}y$ is always safe as $0 \leq i < size(t)$. Moreover, if $i == x$, it is equivalent to $y = t[i]$, and $y = y$ otherwise. Because the initial array access is safe, there is a single $i' \in [0, size(t) - 1]$ such that $x = i'$, and the for loop amounts to a single assignment $y = t[i']$ *i.e.*, $y = t[x]$. Thus, the semantics is preserved. And, because a memory access is done for each i, the trace leaked is the concatenation of all those i *i.e.*, $0 \cdot 1 \cdots size(t)$.

The reasoning is the same for the transformation of memory write. □

2.6 Overall transformation

Our constant-time transformation consists in iterating the previous transformations *i.e.*, SI_p, IS_p, and $DIC_p^{V_P}$, on s until we can't find a p such that $RO_p(s)$. The main idea is that scope-increase will introduce a **next** block on a condition, index sanitization will secure any access within that condition, and delayed if-conversion finally removes this condition. Because there are only a finite number of conditions within a program, and each iteration effectively removes one of them, the transformation always terminates. An overview of the whole transformation can be found in Figure 2.1 as an UML state diagram while the constant-time transformation itself is formalized in Definition 33. Given an initial program P, the transformation consists of first finding a program point p such that RO_pP. If no p satisfies the right-most out-most constraint, we apply the clean-up pass described in the previous section to ensure that no secret memory access remains. However, if such a p exists, we apply scope-increase on the conditional p, then index sanitization on the same conditional before finishing by applying the delayed if-conversion transformation. Once delayed if-conversion is applied, the p conditional has been removed, and we can find a new p' such that $RO_{p'}(P)$. We repeat this operation while we can find such a p'.

Definition 33: Constant-Time Transformation

Let T be the function removing (if it exists) the rightmost outermost problematic conditional of a program P.

$$T(P) = \begin{cases} DIC_p^{V_P}(IS(SI_p(P))) & \textbf{if } RO_p(P) \text{ for some } p \text{ and } V_P = var(P) \\ P & \textbf{otherwise} \end{cases}$$

For a program P, the Constant-Time Transformation $CTT(P)$ iterates the function T until there is no unsecure conditional left.

$$CTT(P) = \begin{cases} P & \textbf{if } \ T(P) = P \\ CTT(T(P)) & \textbf{otherwise} \end{cases}$$

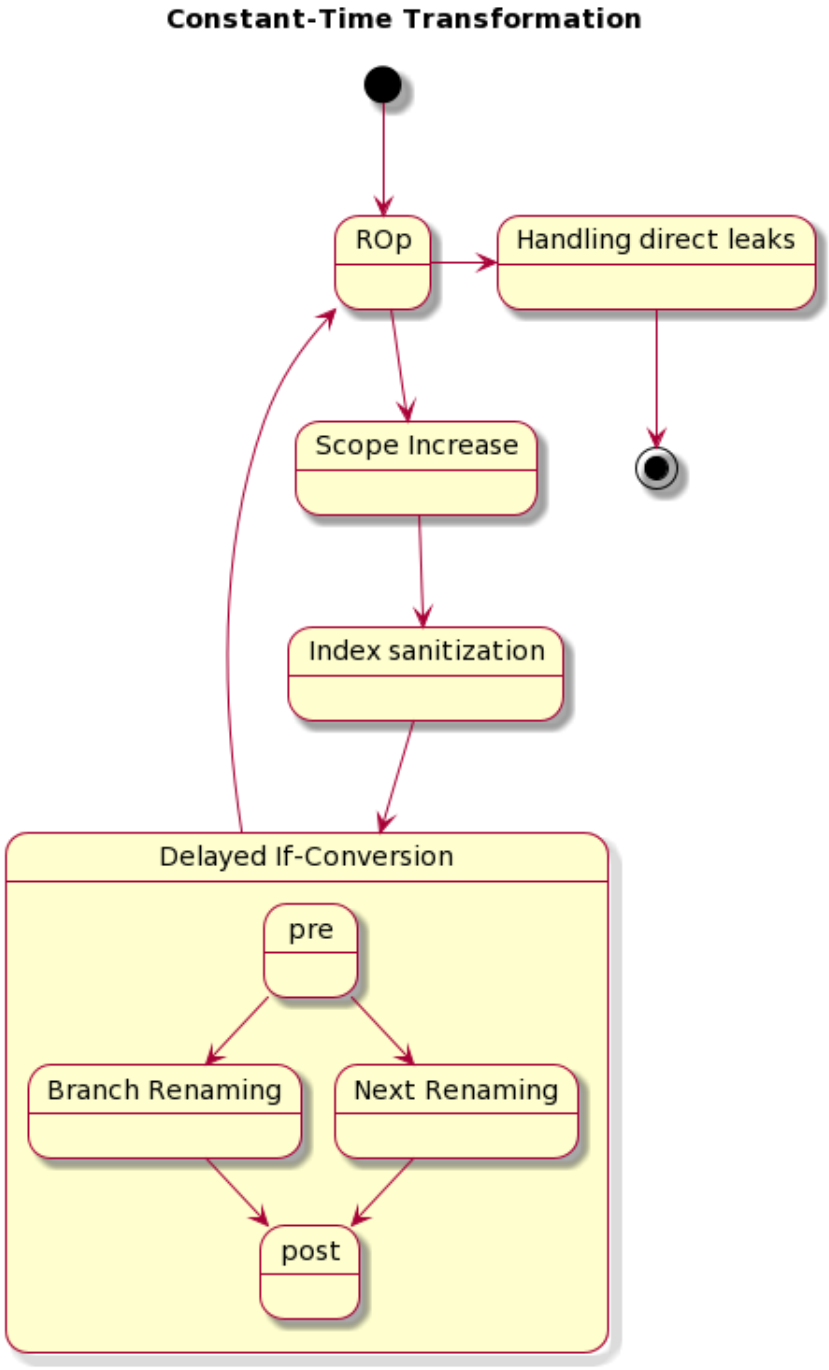


Figure 2.1 – Overview of the Constant-Time Transformation

2.6.1 Semantics Preservation

By defining the Constant-Time Transformation to be an iteration of a function composed of the transformations described above, we have that this transformation preserves the behavior of the program, and removes all guards from the leakage trace. We state this more formally in Theorem 10.

Theorem 10: Semantics preservation of the Constant-Time Transformation

Given a program P, two typing environments σ, σ', and a trace t such that $(\sigma, P) \downarrow^{t} \sigma'$, then the constant-time transformation of P evaluates the same way, and we have $(\sigma, CTT(P)) \downarrow^{t'} \sigma'$ where t' is t stripped of all guards of problematic conditionals within P, and augmented by some double evaluations of array accesses. In particular, t' only contains secrets leaked by array accesses.

Proof Outline. We start by proving that the T function preserves the semantics, and reduces the number of guards in the trace by 1. This is done by chaining Theorems 7, 5, and 8. The preservation of semantics immediately follows, and the decrease in number of guards in the trace is a direct implication of Theorem 8. We then prove that CTT ends because there is only a finite number of guards in a given program, and by induction, the semantics is preserved. □

We can then prove that by applying our clean-up pass, we could remove all secret leakage from the program. However, we will look more into the security of the transformation in Chapter 4.

Chapter 3

Type System

In this chapter, we present our type designed to answer the needs noted in Chapter 2 notably by proposing a method to compute and check annotations. We will first describe our types and their associated lattice before formalizing the annotation shown in Section 2.3 and presenting our flow-sensitive information flow type system which distinguishes between direct and indirect flows. This chapter contains some assumptions; these will be proven in Chapter 4.

3.1 Types

As we pointed out in Section 1.3, the main issue with the classical if-conversion is that it transforms indirect flows into direct ones which are then responsible for leaking secret values. Our proposed solution, delayed if-conversion, detailed in Section 2.1, aims to delay the introduction of direct flow until all possible indirect assignments have been executed. Intrinsically, such a transformation would not need an elaborate type system: we just need to be able to identify secret conditions, and the usual C-T type system does so well. However, to allow us to transform more complex programs, the scope-increase transformation (Section 2.3) performs code motion, targeting the continuation of a conditional containing all corresponding indirect flow. To do so, we need to be able to track indirect flow within the program and differentiate it from its usual, direct, counterpart. Hence, we extend the usual two-point $\{\mathbf{H}, \mathbf{L}\}$ lattice with information flow types of form $\mathbf{I}(l)$ where $l \subseteq \mathbb{P}$ is a subset of program points. The type $\mathbf{I}(l)$ is given to variables that are secret because of an indirect secret flow arising from a conditional labeled with one of the labels in $\mathbb{P}$. We keep the type $\mathbf{H}$ which now means "secret because of a direct or indirect flow". The definition of our lattice can be found in Definition 34.

Definition 34: Type lattice

$$IFType = \mathcal{P}(\mathbb{P}) \cup \{\mathbf{H}\}.$$

In this lattice, $\mathbf{H}$ is the greatest element and $\mathbf{I}(l_1) \sqsubseteq \mathbf{I}(l_2)$ if $l_1 \subseteq l_2$ because it is safe to over-approximate the set of conditionals that caused an indirect flow. The element $\mathbf{I}(\varnothing)$ intuitively means "does not depend on secrets" and types values with only public informations. We will use $\mathbf{L}$ as an abbreviation for the type $\mathbf{I}(\emptyset)$. We also say that a program point p is within a typing environment Γ (noted $p \in \Gamma$) if there exists a variable x such that $\Gamma(x) = \mathbf{I}(l)$ with $p \in l$. Throughout this document, we say that a type τ is simple if $\tau \in \{\mathbf{L}, \mathbf{H}\}$. The same goes for an environment such that, for all x, $\Gamma(x)$ is simple.

This lattice is actually a generalization of the Constant-Time type system, and we define in Definition 35 a function `simplify` taking as an argument a type within our lattice, and yielding the corresponding type in the Constant-Time lattice.

Definition 35: Conversion from lattice to lattice

Assuming that $\{\mathbf{L}_{CT}, \mathbf{H}_{CT}\}$ is the Constant-Time lattice, we have

$$\begin{array}{lcll} \texttt{simplify}(\mathbf{L}) & = & \mathbf{L}_{CT} & \\ \texttt{simplify}(\mathbf{I}(s)) & = & \mathbf{H}_{CT} & if\, s \neq \emptyset \\ \texttt{simplify}(\mathbf{H}) & = & \mathbf{H}_{CT} & \end{array}$$

And, the `simplify` preserves the partial order. We state this property of the function in Theorem 11.

Theorem 11: Order preservation

Given two types t and t' such that $t \sqsubseteq t'$. The conversion of t and t' to the Constant-Time lattice preserves the same ordering, that is

$$\texttt{simplify}(t) \sqsubseteq \texttt{simplify}(t')$$

Proof Outline. By case analysis. Immediate if $t = t'$, or $t, t' \in \{\mathbf{L}, \mathbf{H}\}$. Otherwise, $\texttt{simplify}(t) = \texttt{simplify}(t') = \mathbf{H}_{CT}$, and $\mathbf{H}_{CT} \sqsubseteq \mathbf{H}_{CT}$. □

3.1.1 Classifying

The main tool of our transformation is the predication of statements. This actually transforms an indirect flow into a direct one. Even though we use them sparsely and cleverly, we still need to track their impact on the typing. To do so, we define a *classifying* operator. Declassification is often used in programming: allowing a secret value that we trust to be considered as a public one. Our classifying operator formally defined in Definition 36 does the contrary: it assigns to some variables a higher security level. Indeed, it sets any variables indirectly secret because of a given program point to the **H** security level. This actually simulates a predicated statement, where the dependence is made direct. Formally, given a set of program point l, we set to **H** any variable of type **H** or $\mathbf{I}(l')$ if l and l' are not disjoint. To do so, Definition 36 first defines the operator on a type, before extending it to a typing environment.

Definition 36: Classifying

$$\uparrow_l(\tau) = \begin{cases} \tau & \textbf{if } \tau = \mathbf{I}(l') \wedge l \cap l' = \emptyset \\ \mathbf{H} & \textbf{otherwise} \end{cases}$$

$$(\uparrow_l \Gamma)(x) = \uparrow_l (\Gamma(x))$$

Because the transformation only goes one way, it preserves the type order, and the environment order. This is stated in Lemma 9.

Lemma 9: Monotony of classify

Let τ and τ' be two types such that $\tau \sqsubseteq \tau'$, and l be a set of program point. We have

$$\uparrow_l \tau \sqsubseteq \uparrow_l \tau'$$

Similarly, if we have Γ and Γ' two typing environments such that $\Gamma \sqsubseteq \Gamma'$, we have

$$\uparrow_l \Gamma \sqsubseteq \uparrow_l \Gamma'$$

Proof Outline. We prove the first property by case analysis. If $\tau' = \mathbf{H}$, $\uparrow_l \tau \sqsubseteq \tau'$ for all values of l and τ. Otherwise, $\tau' = \mathbf{I}(()l_2)$, and $\tau = \mathbf{I}(()l_1)$ with $l_1 \subseteq l_2$. Thus, if $l \cap l_1 \neq \emptyset$, we have that $l \cap l_2 \neq \mathit{emptyset}$, and the two types are classified. Finally, if τ is not classified by l, either τ' isn't, and the hypothesis concludes, or τ' is, and $\tau \sqsubseteq \mathbf{H}$.

The second property is simply by definition of the order on environments, and by the first property. □

3.2 Program annotations for Constant-Time Transformation

We presented back in Section 2.3 a concept of program annotations *high* and *leak*, corresponding to the set of program points that respectively were within a statement s, or were indirectly leaked by a statement s. This leaking is most often induced by a memory access on an indirectly secret index. Formally, if a statement s is annotated by two sets g and r (noted $(s)^r_g$), the set g is an over-approximation of the set of conditionals with non-low guards within s, while the set r is an upper bound of the security levels of the indices that have been used to access an array within s. This way of defining r is acceptable because all secret indexes would be of type $\mathbf{I}(p)$ where p is the set of program points of conditionals that caused an indirect flow. Hence, the union of these sets (which can be transformed into the upper bound of the security levels) is equal to the set of conditionals indirectly leaked by s. We will see in a subsequent section how we adapt our type system to compute these annotations.

3.3 Type System for Constant-Time Transformation

To adapt the usual Constant-Time type system to our transformation *i.e.*, allow it to accept non-constant-time programs that we are able to transform into constant-time, we have to loosen some of its rules. Indeed, rules for memory accesses should not enforce that indices are of security level $\mathbf{L}$, but simply that they are not $\mathbf{H}$. By ensuring that all memory accesses are done on indirect flow, the delayed if-conversion coupled with scope-increase should be enough to mitigate the security issues. Similarly, we don't have to impose a restriction on the guard's security level within a conditional: we remove them anyway. However, allowing secret conditionals and tracking indirect flows requires us to keep track of the *context* in which we are evaluating our statement. We model the context by a type: $\mathbf{L}$ at top-level, when there is no context for the statement, and $\mathbf{I}(p)$ when inside conditionals, with p containing all program points of encapsulating conditionals.

This results in a type judgment of the form

$$\Delta, \kappa \vdash \Gamma\{s\}\Gamma'$$

where κ is said context, Δ is the typing environment for array variables, and Γ and Γ' are the typing environments for scalar variables before and after running the annotated statement s.

To compute this context, we use the *safe selection* (Definition 37) to increment it if needed. Indeed, the $\ltimes$ operator yields the value v if τ is not $\mathbf{L}$, and the default value d otherwise. By taking the type of the guard as τ, we are able to increment the context in secret conditionals only. On the other hand, just as in the Constant-Time type system, the typing of arrays is *global* (*i.e.*, not flow-sensitive), and *simple* (*i.e.*, all arrays have a type $\tau \in \{\mathbf{L}, \mathbf{H}\}$). The first is a consequence of our *weak update* modeling, and the second comes directly from the first: it would make not sense to have an array dependent on the conditional p before encountering it. There are no such constraints on the typing environments Γ and Γ'.

Definition 37: Safe selection

$$\tau \ltimes_d v = \begin{cases} d & \textbf{if } \tau = \mathbf{L} \\ v & \textbf{otherwise} \end{cases}$$

3.3.1 Rules for expressions

For expressions, the typing judgment is of the form

$$\Delta, \Gamma \vdash e : \tau, l$$

where τ is the security type of the result and l is the upper bound of security levels of array indices used to compute the value of e. This set l is used to compute the *leak* annotation of statements.

The typing rules of expressions, detailed in Definition 38 are similar to those of the Constant-Time type system, and essentially join the type of the sub-expressions to compute the type of the whole expression. For array accesses, an additional hypothesis enforces that the type of the index is $\mathbf{I}(l')$ *i.e.*,strictly below $\mathbf{H}$. If it is $\mathbf{L}$, there is no leakage and the expression is well typed. Otherwise, there is a leakage of secrets due to indi-

rect flows but the expression is still well-typed because the leakage will be erased by our program transformation.

Definition 38: Typing rules for expressions

$$\frac{}{\Delta, \Gamma \vdash x : \Gamma(x), \varnothing} \qquad \frac{}{\Delta, \Gamma \vdash i : \mathbf{L}, \varnothing} \qquad \frac{\Delta, \Gamma \vdash e : \tau, l \quad \tau = \mathbf{I}(l')}{\Delta, \Gamma \vdash t[e] : \Delta(t) \sqcup \tau, l \cup l'}$$

$$\frac{\Delta, \Gamma \vdash e_i : \tau_i, l_i \quad i \in \{1, 2\}}{\Delta, \Gamma \vdash e_1 \oplus e_2 : \bigsqcup_i \tau_i, \bigcup_i l_i} \qquad \frac{\Delta, \Gamma \vdash e_i : \tau_i, l_i \quad i \in \{1, 2, 3\}}{\Delta, \Gamma \vdash e_1 ? e_2 : e_3 : \bigsqcup_i \tau_i, \bigcup_i l_i}$$

3.3.2 Rules for instructions

For statements, the rules can be found in Definition 39. The ones for **skip** and sequence are standard for a flow-sensitive type system. Note that we take care of updating the annotations of the sequence by joining the annotations of both underlying statements. For an assignment $x = e$, the type for x is updated to be the least upper bound of the type τ of the expression e and the type of the security context κ. This ensures that indirect flows are correctly reflected in the typing of the assigned variable. The rule for array update is flow insensitive. It checks that the type τ_1 of the index is not $\mathbf{H}$. It also checks that the type obtained from joining the type of the index τ_1 and the type of the written value τ_2 with an upgraded security context $\kappa \ltimes_{\mathbf{L}} \mathbf{H}$ is below the type $\Delta(t)$ of the array. Therefore, if the security context κ is $\mathbf{L}$, we have $\tau_1 \sqcup \tau_2 \sqsubseteq \Delta(t)$. Otherwise, if the array update is performed under a security context $\kappa \neq \mathbf{L}$, the safe selection $\kappa \ltimes_{\mathbf{L}} \mathbf{H}$ yields $\mathbf{H}$, and typing constraints entail that $\Delta(t) = \mathbf{H}$. The r annotation is updated to reflect that an array access with an index of type τ_1 has been made.

Definition 39: Typing rules for Instructions

$$\frac{}{\Delta, \kappa \vdash \Gamma\{\mathbf{skip}^{\varnothing}_{\varnothing}\}\Gamma} \qquad \frac{\Delta, \kappa \vdash \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1 \quad \Delta, \kappa \vdash \Gamma_1\{(s_2)^{r_2}_{g_2}\}\Gamma' \quad r = r_1 \cup r_2 \quad g = g_1 \cup g_2}{\Delta, \kappa \vdash \Gamma\{(s_1; s_2)^{r}_{g}\}\Gamma'}$$

$$\frac{\Delta, \Gamma \vdash e : \tau, r}{\Delta, \kappa \vdash \Gamma\{(x = e)^{r}_{\varnothing}\}\Gamma[x \mapsto \tau \sqcup \kappa]} \qquad \frac{\Delta, \Gamma \vdash e_1 : \tau_1, l_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2, l_2 \quad \tau_1 = \mathbf{I}(l'_1) \quad \tau_1 \sqcup \tau_2 \sqcup (\kappa \ltimes_{\mathbf{L}} \mathbf{H}) \sqsubseteq \Delta(t) \quad r = l_1 \cup l'_1 \cup l_2}{\Delta, \kappa \vdash \Gamma\{(t[e_1] = e_2)^{r}_{\varnothing}\}\Gamma}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash c : \tau, r_c \quad \kappa' = \kappa \sqcup (\tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p\})) \\ \Delta, \kappa' \vdash \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1 \quad \Delta, \kappa' \vdash \Gamma\{(s_2)^{r_2}_{g_2}\}\Gamma_2 \\ \Delta, \kappa \vdash \Gamma_1 \sqcup \Gamma_2\{(s_3)^{r_3}_{g_3}\}\Gamma' \\ r = r_1 \cup r_2 \cup r_3 \cup r_c \\ g = g_1 \cup g_2 \cup g_3 \cup \tau \ltimes_{\emptyset} \{p\}\end{array}}{\Delta, \kappa \vdash \Gamma\{(\mathbf{if}^{p}\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3)^{r}_{g}\}\Gamma'}$$

$$\frac{\begin{array}{c}\Gamma \sqsubseteq \Gamma' \quad \Gamma_1 \sqsubseteq \Gamma' \quad \uparrow_{high(s)} \Gamma' = \Gamma' \\ \Delta, \kappa \vdash \Gamma'[i \mapsto \kappa]\{(s)^{r}_{g}\}\Gamma_1\end{array}}{\Delta, \kappa \vdash \Gamma\{(\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s)^{r}_{g}\}\Gamma'}$$

The rule for conditions computes the security level τ of the condition c. If τ is different from $\mathbf{L}$, c contains secret information and the security context κ' is updated with $\mathbf{I}(\{p\})$, recording that execution in the branches takes place under a secret condition located at program point p. This information is also added to the annotation g that is the set of labels of the secret conditions in the statements that it annotates. Simultaneously, the annotation r is updated to be the joining of the annotations for all inner branches, and the leaking of the evaluation of the condition c.

The typing rule for **for** loops checks that Γ' is an invariant typing environment for the loop, by checking the body s can be type checked in the slightly more constraining typing environment $\Gamma'[i \mapsto \kappa]$. In this environment, the iteration variable i gets the type of the security context κ. The rule also enforces that Γ' does not contain any dependency to conditions within s by ensuring that $\uparrow_{high(s)} \Gamma' = \Gamma'$. This equality means that there

are no variables of type $\mathbf{I}(l)$ in Γ, with l containing at least one program point p of a condition within s. This means that we keep track of indirect flows within a loop only, and do not propagate them outside of its containing loop.

3.3.3 Constant-Time property

Because our transformation aims to transform a program into a constant-time part, our type system not only accepts constant-time programs but can, with a set of constraints, accept *only* constant-time programs. Indeed, if we only consider a type derivation in the empty security context ($\kappa = \mathbf{L}$) and a program P with empty annotations ($P^{\emptyset}_{\emptyset}$), our flow tracking type system enforces the constant-time property of Definition 5. This is more formally stated in Theorem 12. This will allow us to formulate more easily our security property: a transformed program shall type with a $\mathbf{L}$ context and without annotations.

Theorem 12: Constant-Time Enforcement

If a program P is well-typed in our flow tracking type system with empty annotations, low context, and a *simple* typing environment Γ *i.e.*,

$$\Delta, \mathbf{L} \vdash \Gamma\{(P)^{\emptyset}_{\emptyset}\}\Gamma'$$

then P is constant-time. More precisely, the predicate $CT(P, L)$ holds for any set of variables satisfying

$$\{x|\Gamma(x) \neq \mathbf{H}\} \cup \{t|\Delta(t) \neq \mathbf{H}\} \subseteq L$$

Proof Outline. Given a type derivation $\Delta, \mathbf{L} \vdash \Gamma\{(P)^{\emptyset}_{\emptyset}\}\Gamma'$, we can exhibit a type derivation $\Delta \vdash^{ct} \Gamma\{P\} \downarrow \Gamma'$ where $\downarrow \Gamma'$ is obtained by mapping all the indirect flow types *i.e.*, $\mathbf{I}(l)$ for some l, to $\mathbf{L}$. By Theorem 1, we conclude the proof. □

3.3.4 Computing annotations

Even though the type system is written as if to ensure that annotations are well-written, we can actually use it to compute these annotations. Consider Program P0, by following the typing rules, we would annotate each memory access $(y = t[x])^{\{p\}}_{\emptyset}$ because of x typing $\mathbf{I}(\{p\})$ due to the indirect flow. Each branch would then be annotated $(s_t)^{\{p\}}_{\emptyset}$ (respectively $(s_e)^{\{p\}}_{\emptyset}$), the assignment to x being benign. By the **if** rule, we can finally compute the annotations of P0 : $(P0)^{\{p\}}_{\{p\}}$.

Reminder 1: Program P0

```
if@p h then x=l1; y=t[x] else x=l2; y=t[x]
```

3.4 Adapting the transformation

Before even trying to transform any program, we pass it through our type system. Two cases arise: it is either rejected or accepted.

If a program is rejected, it is necessarily because of **H** memory accesses, those being the only hard constraint of the system. In the previous chapter, we assumed the existence of a post-transformation pass to mitigate all non-transformable array accesses. All those accesses were actually those made on **H** values. By replacing this hypothesis pass with a preprocessing one in which we apply the array-traversal transformation presented in Section 2.5 on unsecure array accesses until the program is accepted, we can guarantee that any program, or a semantically equivalent version, is accepted by the type system. For example, take the manually curated Program P29 with an initial typing environment $\{h \mapsto \mathbf{H}\}$.

Program P29: Manually curated program with H indexes

```
for i from 0 to 31 do
    if@p h then
        x = 0;
    else
        x = 1;
    t[h] = 5;
y = t[x];
```

Given an initial typing environment $\{h \mapsto \mathbf{H}\}$, our type system would reject the program while typing the **for** loop because of the secret memory write $t[h]$ within its block. Hence, we can apply the naive array traversal transformation on this statement to obtain the Program 30.

Program P30: First iteration of array traversal on Program P29

```
for i from 0 to 31 do
    if@p h then
        x = 0;
    else
        x = 1;
    for j from 0 to size(t) do
        t[j] = (j==h)?5:t[j];
y = t[x];
```

In this instance, the type system accepts the **for** loop. The resulting typing environment after the memory access but still within the loop is $\{h \mapsto \mathbf{H}; x \mapsto \mathbf{I}(\{p\})\}$. Thus, after the loop, our typing system yields $\{h \mapsto \mathbf{H}; x \mapsto \uparrow_{\{p\}} \mathbf{I}(\{p\}) = \mathbf{H}\}$. Because x is typed to **H**, the last memory access is rejected by the program. Similarly, we can apply the array traversal transformation to this statement, yielding Program 31. This program is now fully typable, and accepted by our type system.

Program P31: Second and final iteration of array traversal on Program P29

```
for i from 0 to 31 do
    if@p h then
        x = 0;
    else
        x = 1;
    for j from 0 to size(t) do
        t[j] = (j==h)?5:t[j];
for k from 0 to size(t) do
    y = (k==x)?t[k]:y;
```

Once, or if, a program is accepted by our type system, along with the correct annotations, we can apply our constant-time transformation, as stated in Section 2.6. This gives us the new UML diagram for the transformation shown in Figure 3.1 . We name

$FCTT$, the transformation following this diagram *i.e.*, our whole transformation. To reuse Definition 33, for a given program P and a typing environment Γ, $FCTT_{\Gamma,\Delta}(P) = CTT(AT_\Gamma(P))$, where AT_Γ is the function applying array-traversal until P is typable with Γ. The transformation works as follows: given a program P, we try typing it using our type system. If the typing system rejects the program, it is necessarily because of a memory access, and we apply the array traversal on it. We then try again to type it. When we finally have a program accepted by the typing system, we do one iteration just as in the transformation described in Section 2.6. That is, we find a p such as RO_pP, we apply scope-increase, index sanitization, and then delayed if-conversion. Once this iteration of transformation has been applied, we retype the program. This allows us to get rid of the $\mathbf{I}(\{p\})$ type now that the p conditional has been removed, and to look for a new p' such that $RO_{p'}(P)$. When there is no more such p', the transformation is complete, and the resulting program is constant-time, as proven in the following Chapter 4.

3.5 Auxiliary Type System

Intuitively, after running $SI_p(s_1, s)$, all indirect leaks due to a given p conditional are localized within one of the branches of the conditional. This translates into having all statements annotated with p localized within the conditional. To formalize this using our type system, we devise a strengthened type system which, given a label p, prevents indirect flows from escaping the conditional labeled p. Because the system is parameterized with p, the typing judgment is of the form $\Delta, \kappa \vdash^{@p} \Gamma\{s\}\Gamma'$.

Definition 40: Localized Implicit Flows Typing Rule

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash c : \tau, r_c \quad \kappa' = \kappa \sqcup (\tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p'\})) \\ \Delta, \kappa' \vdash^{@p} \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1 \quad \Delta, \kappa' \vdash^{@p} \Gamma\{(s_2)^{r_2}_{g_2}\}\Gamma_2 \\ \Delta, \kappa \vdash^{@p} \Gamma_1 \sqcup \Gamma_2\{(s_3)^{r_3}_{g_3}\}\Gamma' \\ r = r_1 \cup r_2 \cup r_3 \cup r_c \quad g = g_1 \cup g_2 \cup g_3 \cup \tau \ltimes_\emptyset \{p'\} \\ \Gamma'' = \begin{cases} \uparrow_{\{p\}} \Gamma' \ \textit{if} \ \tau \neq \mathbf{L} \wedge p = p' \\ \Gamma' \ \textit{otherwise} \end{cases}\end{array}}{\Delta, \kappa \vdash^{@p} \Gamma\{(\mathbf{if}^{p'} \ c \ \mathbf{then} \ s_1 \ \mathbf{else} \ s_2 \ \mathbf{next} \ s_3)^r_g\}\Gamma''}$$

It is obtained from our type system of Definitions 38,39 by keeping all the typing

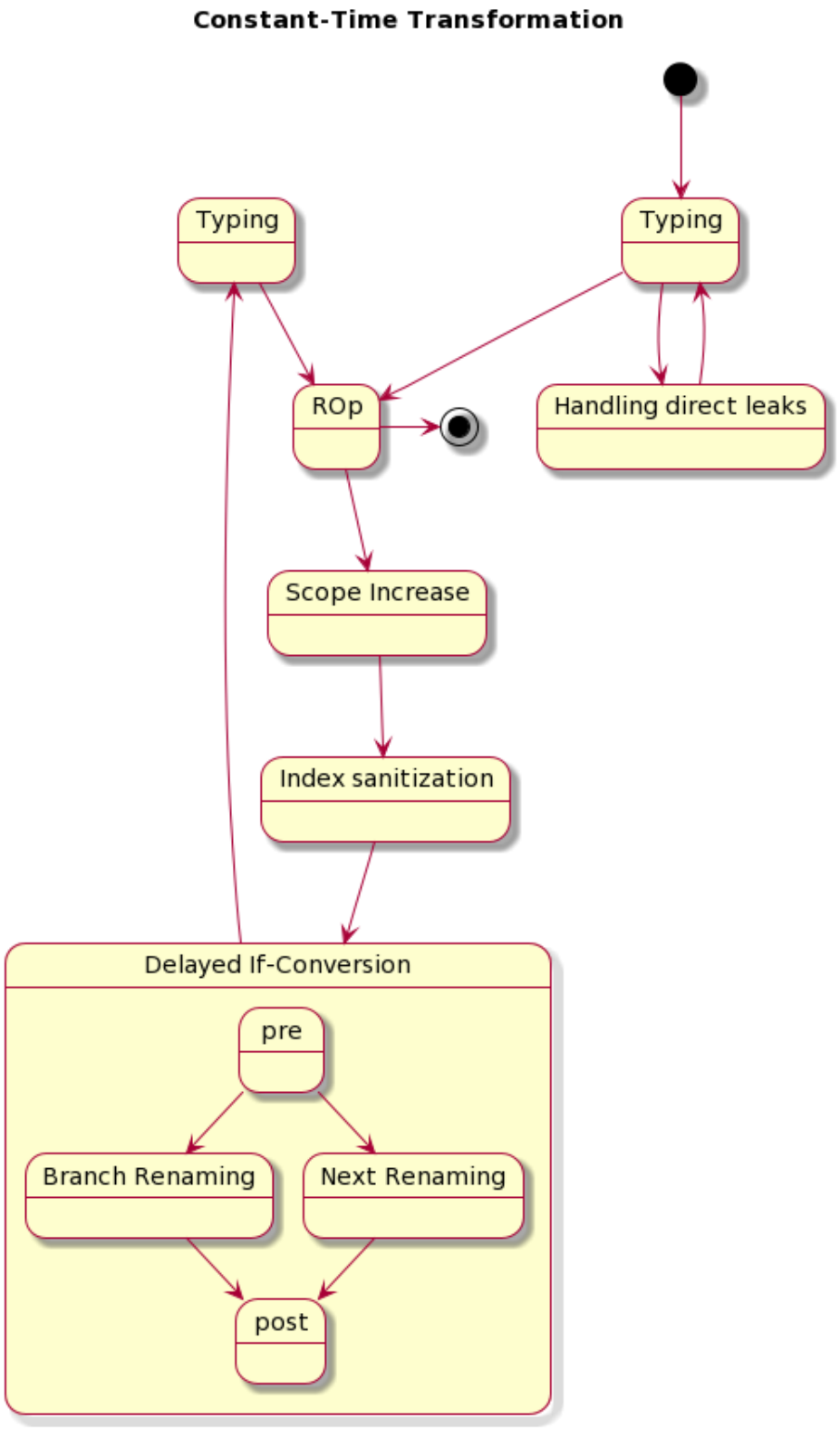


Figure 3.1 – Overview of the Constant-Time Transformation - with typing

rules except the typing rule for the conditional that is replaced by the typing rule of Definition 40. The typing rule of Definition 40 is very similar to the original typing rule. Actually, the typing judgments only differ when the label p' of the condition is p and when the typing of the condition c is not $\mathbf{L}$. In that case, instead of Γ', the final typing environment is $\Gamma'' = \uparrow_{\{p\}} \Gamma'$ which classifies the indirect flows due to the current conditional annotated by p. This ensures that no indirect flow could subsist after the conditional. This auxiliary system will mostly be used when trying to prove the security of our transformation in Chapter 4.

Chapter 4

Security of the transformation

This chapter aims to show that the transformations shown above ensure the constant-property of its result. This property is stated in Theorem 13, and will be our goal for this chapter.

Theorem 13: Security of our transformation

Given a program P and two typing environments Γ and Δ, by defining L the set of public variables in Γ and Δ, then the program resulting of our transformation on P is constant-time w.r.t L, that is :

$$L = \{x | \Gamma(x) \sqcup \Delta(x) = \mathbf{L}\} \implies CT(FCTT_{\Gamma,\Delta}(P), L)$$

We do so by using the type system described in Chapter 2.

The first step of our proof strategy is to show that any program expressed in our language can be typed after a first preprocessing pass (*i.e.*, applying array traversal). Then, we aim to prove that each iteration of our CCT transformation removes a secret conditional of a program and decreases the size of annotations. Because our annotations are of finite size, the transformation ends, and we aim to conclude by Theorem 12.

We present here proofs *on paper*, omitting some details to keep an appropriate level of abstraction. Some of these proofs have been mechanized in Coq and can be found here [1].

4.1 Preprocessing

4.1.1 Typing Constraints

As stated above, the goal of the preprocessing pass is to ensure that any program accepted by our language is accepted by our type system, for a given typing environment.

1. https://github.com/frosqh/ctt_proofs

By default, this isn't true: given a typing environment where h is mapped to $\mathbf{H}$, the memory access $x = t[h]$ is adequately rejected by the type system. Note that the only valid reason for a program to be rejected is because of a secret access, or because of a misconstructed annotation. This is trivial by looking at the typing rules: an expression is rejected only if a memory access is done on an index with a type $\tau \neq \mathbf{I}(l')$ for any l', *i.e.*, if τ is $\mathbf{H}$. For expressions, aside from annotations that introduce a lot of constraints on our type system, the only true limitation is in the rule for $t[e_1] = e_2$, we reject any program where the type of e_1 τ_1 is not a $\mathbf{I}(l'_1)$ for any l'_1, *i.e.*, where the index is typed $\mathbf{H}$. Another way to look at it is to look at programs without any accesses: those programs should type independently of Γ, as shown in Lemma 10.

Lemma 10: Typing without arrays

Given a program P not containing a single array, two typing environments Γ and Δ, and a context κ, then there exists a typing environment Γ' such that P is typable and yields Γ', that is:

$$\Delta, \kappa \vdash \Gamma\{P\}\Gamma'$$

Proof Outline. First, we prove a similar lemma for the expressions, by induction: there are four non-array rules, and all accept any expression.

Then, we proceed by induction on P. Aside from the $t[e_1] = e_2$ rule, which we cannot encounter because of our syntactic constraints, there are two base cases: **skip**, which is trivial, and $x = e$, which cannot fail either. By induction, the sequence and the conditional are immediate. For the **for** loop, we can choose Γ' to be the $\mathbf{H}$ environment, *i.e.*, $\Gamma'(x) = \mathbf{H}$ for all x, and conclude by induction. □

4.1.2 Array Traversal

By Lemma 10, we have shown that the only way for a program to be rejected by our type system is because of unsecure memory accesses. To alleviate this issue, we use the array traversal transformation. Because the result of this transformation only uses memory accesses on loop indexes, and loop indexes are always public, those results should always be typable. This is stated in Lemmas 11, 12. Hence, at the potential cost of classifying an array, as shown in Lemma 12, we can apply our array traversal transformation to any blocking memory access, and obtain a typable program.

Lemma 11: Typing an array traversal on memory read

Given an array t, two variables y, x, a context $\kappa \neq \mathbf{H}$, and two typing environments Γ, Δ, then the array traversal transformation of $y = t[x]$ is typable for Δ, κ, and Γ, that is, there exists a Γ' such that :

$$\Delta, \kappa \vdash \Gamma\{\textbf{for } i \textbf{ from } 0 \textbf{ to } \mathit{size}(t){-}1 \textbf{ do } y = (x == i)?t[i]{:}y\}\Gamma'$$

Proof Outline. By defining $\tau = \Gamma(y) \sqcup \Gamma(x) \sqcup \kappa \sqcup \Delta(t)$, and $\Gamma' = \Gamma[i \mapsto \mathbf{H}][y \mapsto \tau]$, we have $\Gamma \sqsubseteq \Gamma'$ (because $\Gamma(y) \sqsubseteq \Gamma(y) \sqcup \Gamma(x) \sqcup \kappa \sqcup \Delta(t)$ and $\Gamma(i) \sqsubseteq \mathbf{H}$).

Moreover, per the rule of **ctselect** and assignment,

$$\Delta, \kappa \vdash \Gamma'[i \mapsto \kappa]\{y = (x == i)?t[i]{:}y\}\Gamma'[i \mapsto \kappa][y \mapsto \Gamma(x) \sqcup \kappa \sqcup \Delta(t) \sqcup \tau]$$

Note that $\Gamma(x) \sqcup \kappa \sqcup \Delta(t) \sqcup \tau = \tau$, and

$$\Gamma'[i \mapsto \kappa][y \mapsto \tau] = \Gamma[i \mapsto \mathbf{H}][y \mapsto \tau][i \mapsto \kappa][y \mapsto \tau] = \Gamma[i \mapsto \kappa][y \mapsto \tau]$$

And, $\Gamma[i \mapsto \kappa][y \mapsto \tau] \sqsubseteq \Gamma'$ (because $\kappa \sqsubseteq \mathbf{H}$), and we can conclude by the **for** rule. □

Lemma 12: Typing an array traversal on memory write

Given an array t, two variables y, x, a context $\kappa \neq \mathbf{H}$ and two typing environments Γ, Δ such that $\Delta(t) = \mathbf{H}$, then the array traversal transformation of $t[x] = y$ is typable for Δ, κ and Γ, that is, there exists a Γ' such that :

$$\Delta, \kappa \vdash \Gamma\{\textbf{for } i \textbf{ from } 0 \textbf{ to } \mathit{size}(t){-}1 \textbf{ do } t[i] = (x == i)?y{:}t[i]\}\Gamma'$$

Proof Outline. By taking $\Gamma' = \Gamma[i \mapsto \mathbf{H}]$, we have that $\Gamma \sqsubseteq \Gamma'$.

More over, per the rule of **ctselect** and memory write, and because $\Delta(t) = \mathbf{H}$,

$$\Delta, \kappa \vdash \Gamma'[i \mapsto \kappa]\{t[i] = (x == i)?y{:}t[i]\}\Gamma'[i \mapsto \kappa]$$

And, $\Gamma'[i \mapsto \kappa] = \Gamma[i \mapsto \mathbf{H}][i \mapsto \kappa] = \Gamma[i \mapsto \kappa] \sqsubseteq \Gamma'$ (because $\kappa \sqsubseteq \mathbf{H}$), and we can conclude by the **for** rule. □

4.2 Scope-Increase

To follow the proof strategy described at the start of this chapter, we aim to show that any iteration scope-increase into index sanitization into delayed if-conversion removes a secret conditional of the program. This results in showing that for any such iteration, if the *high* annotation of our program is of cardinality n, the resulting partially transformed program is typable with a *high* annotation of cardinality $n-1$. The first step of this proof is to show that, given an annotated P and its appropriate rightmost condition p such that $RO_p(P)$, we can decompose P as a sequence $c; s$ where c and s are two annotated statements such that $RO_p(c)$ and $high(s) = \emptyset$, and type the scope-increase transformation $SI_p(c, s)$ with the strengthened type system shown in 40. This would mean that we are able to identify the whole continuation of the conditional p as s and that the scope-increase transformation virtually restrains all leaks relative to p within its own scope, as intuited in Section 3.5. Thus, our goal for this section is to prove the following Theorem 14.

Theorem 14: Security of *Scope Increase*

Let p be the rightmost (high) condition of c *i.e.*,$RO_p(c)$ and s be a program without any high condition *i.e.*,$high(s) = \emptyset$. Suppose that $c; s$ is well-typed *i.e.*,

$$\Delta, \mathbf{L} \vdash \Gamma\{c; s\}\Gamma'$$

and $\forall x, p \notin \Gamma(x)$.

We have that $SI_p(c, s)$ is well-typed with respect to the strengthened type system. More precisely,

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c, s)\} \uparrow_{\{p\}} \Gamma'$$

4.2.1 Intermediate Results for SI

To prove this theorem, multiple intermediate results are needed. First, we have to state that the upgrade from our type system to the one strengthened w.r.t to the p conditional is free on any statement not containing p. Indeed, if a statement s does not contain the p conditional, adding a constraint on said conditional is trivial, and has no impact on the typing. This is stated and proved in Lemma 13. This will allow us to cross over to the strengthened type system in base cases.

Lemma 13: Free upgrade

Let s be a program, p a program point such that $p \notin \mathit{high}(s)$, and s is well-typed, *i.e.*,

$$\Delta, \kappa \vdash \Gamma\{s\}\Gamma'$$

Then, s is also well-typed using the strengthened type system for $\vdash^{@p}$, *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{s\}\Gamma'$$

Proof. The proof is by induction over the height of the type derivation.

— For $s \neq \textbf{if}^p\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3$, the judgments $\vdash$ and $\vdash^{@p}$ are identical. As a result, we can reconstruct an isomorphic typing derivation by simply replacing $\vdash$ with $\vdash^{@p}$.

— For $s = \textbf{if}^p\ c\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{next}\ s_3$, we have $\mathit{high}(s) = \mathit{high}(s_1) \cup \mathit{high}(s_2) \cup \mathit{high}(s_3) \cup \tau \ltimes_\emptyset \{p\}$ for τ the type of the condition c *i.e.*, $\Delta, \Gamma \vdash c : \tau$. We perform a case analysis over the type τ.

  — For $\tau = \mathbf{L}$, the derivation of $\vdash$ and $\vdash^{@p}$ are still the same and therefore the property holds.

  — For $\tau \neq \mathbf{L}$, we have $\mathit{high}(s) = \mathit{high}(s_1) \cup \mathit{high}(s_2) \cup \mathit{high}(s_3) \cup \mathbf{I}(\{p\})$. Therefore, we have $p \in \mathit{high}(s)$ and the property follows by contradiction.

□

Because the strengthened type system introduces a classifying operator on the resulting environment in certain cases (*i.e.*, if the typed conditional is the *to-be-secured* one), Theorem 14 introduces such an operator in all cases. In particular, we have to be able to show that after a classifying, the continuation is still typable. We first do so by showing in Lemma 14 that if an expression not leaking p is typable for a given typing environment, it is also typable if this environment is classified on p.

Lemma 14: Preservation of typing by classifying

Let e be an expression, r a set of program points, and p a program point such that $p \notin r$, and e is well-typed leaking r, *i.e.*,

$$\Delta, \Gamma \vdash e : \tau, r$$

Then, e is also well-typed for the classified environment $\uparrow_{\{p\}} \Gamma$, *i.e.*,

$$\Delta, \uparrow_{\{p\}} \Gamma \vdash e :\uparrow_{\{p\}} \tau, r$$

Proof. The proof is by induction over the height of the type derivation.

— For $e = x$, we have that $r = \emptyset$, and we need to prove that $\Delta, \uparrow_{\{p\}} \Gamma \vdash x :\uparrow_{\{p\}} (\Gamma(x)), \emptyset$. This is immediate per the definition of the classifying operator on typing environments: $\uparrow_{\{p\}} \Gamma(x) =\uparrow_{\{p\}} (\Gamma(x))$ and the typing rule for variables.

— For $e = c$, because $\uparrow_{\{p\}} \mathbf{L} = \mathbf{L}$, we can directly conclude by the typing rule for constants.

— For $e = e_1 \oplus e_2$ and $e = e_1?e_2{:}e_3$, we conclude by induction hypothesis.

— For $e = t[e_1]$, we have that $r = l \cup l'$ and $\Delta, \Gamma \vdash t[e_1] : \Delta(t) \sqcup \mathbf{I}(l'), r$, and we need to prove that $\Delta, \uparrow_{\{p\}} \Gamma \vdash t[e_1] :\uparrow_{\{p\}} (\Delta(t) \sqcup \mathbf{I}(l')), \emptyset$. Because $\Delta(t)$ is simple, we have that $\uparrow_{\{p\}} \Delta(t) = \Delta(t)$. Similarly, because $p \notin r$, we know that $p \notin l'$, and $\uparrow_{\{p\}} \mathbf{I}(l') = \mathbf{I}(l')$. We can then conclude by the memory read rule and by induction hypothesis.

□

We generalize it in Lemma 15 by stating that for any s well-typed not containing the p conditional, the statement would still be typable even if both starting and resulting typing environments are classified on p. To prove this, we rely heavily on the monotony of $\uparrow_{\{p\}}$ (Lemma 9) and Lemma 14 for the base cases. For other cases, the proof relies on the induction hypothesis and arithmetic operations around the classify operator.

Lemma 15: Free classify

Let s be a well-typed statement in a $\mathbf{L}$ security context with an empty set of $\mathbf{H}$ conditionals (*i.e.*, $high(s) = \emptyset$) and such that $p \notin leak(s)$ *i.e.*,

$$\Delta, \mathbf{L} \vdash \Gamma\{s\}\Gamma'$$

Then, s is well-typed for the classified environment $\uparrow_{\{p\}} \Gamma$, *i.e.*,

$$\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma)\{s\}(\uparrow_{\{p\}} \Gamma')$$

Proof. The proof is by induction over the height of the typing derivation.

— For $s = \mathbf{skip}$, we have $\Gamma = \Gamma'$ and need to prove $\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma)\{\mathbf{skip}^r_\emptyset\}(\uparrow_{\{p\}} \Gamma)$. This is done by the typing rule of **skip**.

— For $s = (x = e)$, we have the hypotheses $H_1 : \Delta, \Gamma \vdash e : \tau, r$, $H_2 : \Gamma[x \mapsto \tau] \sqsubseteq \Gamma'$ and need to prove $\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma)\{(x = e)^r_\emptyset\}(\uparrow_{\{p\}} \Gamma')$. By Lemma 14, using H_1, we have $\Delta, (\uparrow_{\{p\}} \Gamma) \vdash e :\uparrow_{\{p\}} \tau, r$. By the typing rule for assignment, it remains to prove $(\uparrow_{\{p\}} \Gamma)[x \mapsto \uparrow_{\{p\}} \tau]) \sqsubseteq (\uparrow_{\{p\}} \Gamma')$. We conclude by monotony of $\uparrow_{\{p\}}$ over H_2 and because $\uparrow_{\{p\}} (\Gamma[x \mapsto \tau]) = (\uparrow_{\{p\}} \Gamma)[x \mapsto \uparrow_{\{p\}} \tau]$.

— For $s = (t[e_1] = e_2)$. Let τ_1 be the type of e_1 and τ_2 be the type of e_2. The proof is similar to the case of assignment and uses the monotony of $\uparrow_{\{p\}}$ to retype the expressions e_1 and e_2 in a classified environment. By the typing rule for array write, it remains to prove that $\tau_1 \sqcup \tau_2 \sqsubseteq \Delta(t) \Rightarrow (\uparrow_{\{p\}} \tau_1) \sqcup (\uparrow_{\{p\}} \tau_2) \sqsubseteq \Delta(t)$. By case analysis over $\Delta(t)$

  — If $\Delta(t) = \mathbf{L}$, we have $\tau_i = (\uparrow_{\{p\}} \tau_i) = \mathbf{L}$ and the property holds.

  — If $\Delta(t) = \mathbf{H}$, then $\tau \sqsubseteq \Delta(t)$ for any type τ and the property also holds.

— For $s = (s_1; s_2)$, the proof follows by using the induction hypothesis over s_1 and s_2 and by monotony of $\uparrow_{\{p\}}$.

— For $s = \mathbf{if}^p\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3$, we have

$$\begin{gathered} Hc : \Delta, \Gamma \vdash c : \tau, rc \\ H_1 : \Delta, (\tau \ltimes_\emptyset \{p\}) \vdash \Gamma\{s_1\}\Gamma_1 \\ H_2 : \Delta, (\tau \ltimes_\emptyset \{p\}) \vdash \Gamma\{s_2\}\Gamma_2 \\ H_2 : \Delta, \mathbf{L} \vdash \Gamma_1 \sqcup \Gamma_2\{s_3\}\Gamma' \end{gathered}$$

and we have to prove

$$\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma)\{\mathbf{if}^p\ c\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{next}\ s_3\}(\uparrow_{\{p\}} \Gamma').$$

Because there are no high conditionals, we conclude that $\tau = \mathbf{L}$ and therefore $\tau \ltimes_\emptyset \{p\} = \emptyset$. The proof follows by using the induction hypothesis over H_1, H_2 and H_3 using the property that $(\uparrow_{\{p\}} \Gamma_1) \sqcup (\uparrow_{\{p\}} \Gamma_2) = \uparrow_{\{p\}} (\Gamma_1 \sqcup \Gamma_2)$.

— For $s = \mathbf{for}\ x\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ b$, we have

$$H_1 : \Delta, \mathbf{L} \vdash \Gamma'[x \mapsto \mathbf{L}]\{b\}\Gamma_1$$
$$H_2 : \uparrow_{C(b)} \Gamma' = \Gamma', H_3 : \Gamma_1 \sqsubseteq \Gamma', H_4 : \Gamma \sqsubseteq \Gamma'$$

and need to prove

$$\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma)\{s\}(\uparrow_{\{p\}} \Gamma').$$

The proof follows by induction hypothesis over H_1 using that $((\uparrow_{\{p\}} \Gamma')[x \mapsto \mathbf{L}]) = \uparrow_{\{p\}} (\Gamma'[x \mapsto \mathbf{L}])$ and $(\uparrow_{C(b)} \uparrow_{\{p\}} \Gamma') = (\uparrow_{\{p\}} \uparrow_{C(b)} \Gamma') = \uparrow_{\{p\}} \Gamma'$.

□

4.2.2 SI Security Theorem

With the help of those three lemmas, we aim to prove Theorem 14 stated at the start of this section. To do so, we work by induction over the statement c. In most cases, we deconstruct the typing derivation, apply both Lemma 13 and the induction hypothesis, and reconstruct the correct one in the strengthened type system. This is not enough for the sequence case $s_1; s_2$ where the conditional is in s_1, because of the classifying, thus needing Lemma 15. The other complex case is the **for** loop, where the subtyping characteristics of the rule may cause some introduction of spurious leaks. To circumvent this issue, we construct a stricter, safer, stronger typing environment on which we call the induction hypothesis.

Reminder: Theorem 14

Let p be the rightmost (high) condition of c *i.e.*, $RO_p(c)$ and s be a program without any high condition *i.e.*, $high(s) = \emptyset$. Suppose that $c; s$ is well-typed *i.e.*,

$$\Delta, \mathbf{L} \vdash \Gamma\{c; s\}\Gamma'$$

and $\forall x, p \notin \Gamma(x)$.

We have that $SI_p(c, s)$ is well-typed with respect to the strengthened type system. More precisely,

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c, s)\} \uparrow_{\{p\}} \Gamma'$$

Proof. By the typing rule of the sequence, we have that

$$\Delta, \mathbf{L} \vdash \Gamma\{c\}\Gamma_c \text{ and } \Delta, \mathbf{L} \vdash \Gamma_c\{s\}\Gamma'$$

The proof is by induction over the typing derivation of program c.

— The base cases $s = \textbf{skip}$, $s = (x = e)$ and $c = (t[e_1] = e_2)$ do not contain a condition. As a result, p cannot be the rightmost condition of c and therefore the property is vacuously true.

— For $c = c_1; c_2$, we have $\Delta, \mathbf{L} \vdash \Gamma\{c_1\}\Gamma_1$ and $\Delta, \mathbf{L} \vdash \Gamma_1\{c_2\}\Gamma_c$. We perform a case analysis depending on whether $p \in c_1$ or $p \in c_2$.

— Suppose that $p \in c_1$. We have $(s_1, s_2) \in sep_p(c_2; s)$ and because sep_p preserves typing we have $\Delta, \mathbf{L} \vdash \Gamma_1\{s_1\}\Gamma_1'$ and $\Delta, \mathbf{L} \vdash \Gamma_1'\{s_2\}\Gamma'$ and need to prove

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c_1, s_1); s_2\} \uparrow_{\{p\}} \Gamma'.$$

By induction hypothesis, we get $\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c_1, s_1)\} \uparrow_{\{p\}} \Gamma_1'$. By Lemma 15, we get $\Delta, \mathbf{L} \vdash (\uparrow_{\{p\}} \Gamma_1')\{s_2\}(\uparrow_{\{p\}} \Gamma')$ and conclude by Lemma 13.

— Suppose that $p \in c_2$. In that case, we need to prove

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{c_1; SI_p(c_2, s)\} \uparrow_{\{p\}} \Gamma'.$$

Because $p \notin c_1$ and $p \notin \Gamma$, we can exhibit alternative typing derivation for c_1 and c_2

$$\Delta, \mathbf{L} \vdash \Gamma\{c_1\}\Gamma_1' \text{ and } \Delta, \mathbf{L} \vdash \Gamma_1'\{c_2\}\Gamma_c$$

such that $p \notin \Gamma'_1$ and $\Gamma'_1 \sqsubseteq \Gamma_1$. The property follows by using Lemma 13 on the typing derivation of c_1 and applying the induction hypothesis on the typing derivation of c_2.

— For $c = \mathbf{if}^{p'}\ x\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$, we perform a case analysis over p'.

— For $p = p'$, we have $SI_p(c, s) = \mathbf{if}^{p}\ c\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{next}\ s$ and we therefore need to prove

$$\Delta, \mathbf{L} \vdash \Gamma\{\mathbf{if}^{p}\ x\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{next}\ s\} \uparrow_{\{p\}} \Gamma'$$

We cannot directly apply Lemma 13 over the typing derivation of c because we expect that $p \in high(c)$. Yet, $p \notin c_1$ and $p \notin c_2$ and therefore $p \notin high(c_1)$ and $p \notin high(c_2)$. Hence, we can apply Lemma 13 over the derivations for c_1, c_2, and s. For c, we can reconstruct a typing derivation,

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{c\}\Gamma''$$

where $\Gamma'' = \Gamma'$ if $\Gamma(x) = \mathbf{L}$ or $\Gamma'' = \uparrow_{\{p\}} \Gamma'$ if $\Gamma(x) \neq \mathbf{L}$. In either case, $\Gamma'' \sqsubseteq \uparrow_{\{p\}} \Gamma'$ and the property holds.

— For $p \neq p'$, we have that $\Gamma(x) = \mathbf{L}$. Without loss of generality, suppose that $p \in c_1$. As a result, given

$$\begin{gathered} H_1 : \Delta, \mathbf{L} \vdash \Gamma\{c_1\}\Gamma_1 \\ H_2 : \Delta, \mathbf{L} \vdash \Gamma\{c_2\}\Gamma_2 \\ H_3 : \Gamma_1 \sqcup \Gamma_2 \sqsubseteq \Gamma_c \\ H_4 : \Delta, \mathbf{L} \vdash \Gamma_c\{s\}\Gamma' \end{gathered}$$

we have to prove

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{\mathbf{if}^{\}}\ x\ \mathbf{then}\ SI_p(c_1, s)\ \mathbf{else}\ c_2; s\ \mathbf{next}\ p' \uparrow_{\{p\}} \Gamma'.$$

Because $\Gamma_2 \sqsubseteq \Gamma_c$ and $\Gamma_1 \sqsubseteq \Gamma_c$, we can exhibit a typing derivation for s starting from either Γ_1 or Γ_2. Therefore, from H_2 and H_4, we have $\Delta, \mathbf{L} \vdash \Gamma\{c_2; s\}\Gamma'$. Moreover, by Lemma 13, we can lift the derivation to $\vdash^{@p}$ and weaken Γ' to $\uparrow_{\{p\}} \Gamma'$. By induction hypothesis, we also get

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{SI_p(c_1, s)\} \uparrow_{\{p\}} \Gamma'.$$

The proof follows by applying the typing rule for the condition.

— For $c = \textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } b$, by the typing rule of the for loop, we have

$$H_1 : \Delta, \mathbf{L} \vdash \Gamma_c[x \to \mathbf{L}]\{b\}\Gamma'_c$$
$$H_2 : \uparrow_{high(b)} \Gamma_c = \Gamma_c$$
$$H_3 : \Gamma'_c \sqsubseteq \Gamma_c$$
$$H_4 : \Gamma \sqsubseteq \Gamma_c$$
$$H_5 : \Delta, \mathbf{L} \vdash \Gamma_c\{s\}\Gamma'$$

and we need to prove

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{\textbf{for } x \textbf{ from } c_1 \textbf{ to } c_2 \textbf{ do } SI_p(b, \textbf{skip}); s\} \uparrow_{\{p\}} \Gamma'.$$

A difficulty is that the induction hypothesis directly cannot be applied over H_1 because we do not have the guarantee that $p \notin \Gamma_c$. The solution is to exhibit an alternative typing derivation for b starting from an environment $\downarrow \Gamma_c$ that is obtained from Γ_c by erasing all the spurious indirect flows that are present in Γ_c but absent from Γ. As a result, we get a stronger typing derivation that is still a fixpoint

$$\Delta, \mathbf{L} \vdash (\downarrow \Gamma_c)\{b\}(\downarrow \Gamma_c)$$

By induction hypothesis, we get

$$\Delta, \mathbf{L} \vdash^{@p} (\downarrow \Gamma_c)[x \mapsto \mathbf{L}]\{SI_p(b, \textbf{skip})\} \uparrow_{\{p\}} \downarrow \Gamma_c$$

However, by H_2 and because $p \in C(b)$, we have $\uparrow_{\{p\}} \downarrow \Gamma_c = \downarrow \Gamma_c$. The proof follows by using the typing rules for the for loop and the sequence.

□

4.3 Security of IS

The second step of our transformation of a typed program, as shown in Figure 3.1, is the sanitization of all problematic memory accesses. To ensure any property on the following steps of the transformation, we show here that applying the index sanitization pass to a given conditional p does not affect the typing of said conditional. We first state a

similar property on expression *i.e.*, that any typable expression is still typable if we apply to it the index sanitization transformation. This is stated in Lemma 16, and proved by induction over the expression. In most cases, the induction hypothesis is enough. However, the main argument for the preservation of the typing is the public status of the size of the array, allowing us to introduce these bounds at no cost. Note that both this proof and the following are general: we do not account for the syntax restriction on *simple* memory accesses.

Lemma 16: Security of index instrumentation of expression

Let e be a well-typed expression, *i.e.*,

$$\Delta, \Gamma \vdash e : \tau, l$$

for some Δ, Γ, τ and l.

Then, the instrumented version $IS_e(e)$ is also well-typed, and we have

$$\Delta, \Gamma \vdash IS_e(e) : \tau, l$$

Proof. The proof is by induction over the structure of the expression e.

Case 1 ($e = x$ **or** $e = c$) As $IS_e(e) = e$, the property holds.

Case 2 ($e = e_1 \oplus e_2$) We have $IS_e(e_1 \oplus e_2) = IS_e(e_1) \oplus IS_e(e_2)$. The property follows by induction hypothesis using the typing rule for $\oplus$.

Case 3 ($e = e_1?e_2{:}e_3$) We have that $IS_e(e) = IS_e(e_1)?IS_e(e_2){:}IS_e(e_3)$. Like Case 2, the property follows by induction hypothesis using the typing rule for the conditional expression.

Case 4 ($e = t[e_2]$) We have $\Delta, \Gamma \vdash t[e_2] : \tau, l$ and need to prove $\Delta, \Gamma \vdash t[0 \leq IS_e(e_2) \leq size(t)?IS_e(e_2){:}0] : \tau, l$. By the typing rule for array access, we have $\Delta, \Gamma \vdash e_2 : \mathbf{I}(l_1), l_2$ for some l_1 and l_2, $\tau = \Delta(t) \sqcup \mathbf{I}(l_1)$ and $l = l_2 \cup l_1$. By induction hypothesis, we have $\Delta, \Gamma \vdash IS_e(e_2) : \mathbf{I}(l_1), l_2$. Because 0 and $size(t)$ are public values, and using the typing rule for conditional expressions, we get that the instrumented index has the typing as e_2 *i.e.*,

$$\Delta, \Gamma \vdash 0 \leq IS_e(e_2) \leq size(t)?IS_e(e_2){:}0 : \mathbf{I}(l_1), l_2$$

By the array access rule, the property holds. □

However, the bulk of the transformation is done on instructions, not expressions. Thus, we state for those a similar property: a typable program is still typable after applying the index sanitization transformation. The main argument is the same as for expressions: the transformation is very local, and only uses public values (mainly the array size). We first show in Lemma 17 the consequence of applying the IS transformation (implicitly to the already selected blocks of the p conditional), before finally stating in Theorem 15 the property for the whole IS_p one.

Lemma 17: Security of local index instrumentation

Let P a program typable for our strengthened type system *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{P\}\Gamma'$$

for some κ, Γ, Δ, Γ' and let P' be the program P where all the array accesses are instrumented according to the rules ARR-SAN and ARR-ASS *i.e.*, $P' = IS(P)$.

We have that the program P' is still typable for the same environments *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{P'\}\Gamma'$$

Proof. By induction over the syntax of P.

Case 1 **($P =$ skip)** By definition of IS, we have that $IS(P) = P$.
Hence, $\Delta, \kappa \vdash^{@p} \Gamma\{P'\}\Gamma'$

Case 2 **($P = (x = e)^r_g$)** By definition of IS, we have that $IS(P) = (x = IS_e(e))$.
Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{(x = e)^r_g\}\Gamma'$ gives us that there exists τ such that

$$\Delta, \Gamma \vdash e : \tau, r$$

By Lemma 16, we have that $\Delta, \Gamma \vdash IS(e) : \tau, r$
Hence, $\Delta, \kappa \vdash^{@p} \Gamma\{(x = IS(e))^r_g\}\Gamma'$

Case 3 **($P = (t[e_1] = e_2)^r_g$)** By definition of IS, we have that

$$IS(P) = t[0 \leq IS_e(e_1) \leq size(t)?IS_e(e_1):0] = IS_e(e_2)$$

Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{(t[e_1] = e_2])^r_g\}\Gamma'$ gives us that there exists τ_1, τ_2, l_1, l_2 such that

$$\Delta, \Gamma \vdash e_1 : \tau_1, l_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2, l_2$$

$$\tau_1 = \mathbf{I}(l_1')$$

$$\tau_1 \sqcup \tau_2 \sqcup (\kappa \ltimes_{\mathbf{L}} \mathbf{H}) \sqsubseteq \Delta(t)$$

$$r = l_1 \sqcup l_2 \sqcup l_1'$$

By Lemma 16, we have that

$$\Delta, \Gamma \vdash IS_e(e_1) : \tau_1, l_1 \quad \Delta, \Gamma \vdash IS_e(e_2) : \tau_2, l_2$$

Furthermore, 0 and $size(t)$ are public values, so

$$\Delta, \Gamma \vdash 0 \leq IS_e(e_1) \leq size(t)?IS_e(e_1):0 : \tau_1, l_1$$

We conclude with the memory write rule.

Case 4 (Other cases) By induction, Lemma 16 and the suitable rule, similarly to Case 2. □

Theorem 15: Security of index instrumentation

Let P a program typable for our strengthened type system *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{P\}\Gamma'$$

for some κ, Γ, Δ and Δ', and P' be the program P where all the array accesses within the p conditional are instrumented according to the rules ARR-SAN and ARR-ASS *i.e.*, $P' = IS_p(P)$.

We have that the program P' is still typable for the same environments *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{P'\}\Gamma'$$

Proof Outline. By induction over the type derivation of P. Until the instruction is the p conditional, the IS_p transformation is the identity function, so the result is immediate. In the case of the p conditional, we can conclude by using Lemma 17. □

4.4 Security of if-conversion

The last step of a single iteration of our transformation is the delayed if-conversion, which should remove the p conditional. Removing a conditional should have an impact on the typability of the program by allowing the *high* annotation of the program to be smaller by one *i.e.*, without the p program point. Unlike previous transformations, this one uses renaming maps and is composed of multiple distinct steps. Thus, we formulate a lemma for each of those steps, as well as multiple intermediate results on renaming maps, just like we did in Chapter 2 when we proved the semantics preservation. For the different steps of the transformation, the main idea of the proof is that the *pre* step will introduce new typing environments taking into account the renaming maps, while the Rn_h^ρ pass will pass along these typing environments. Finally, the Nxt_h transformation ensures that a typing is preserved even with new renaming maps, and the *post* statement merges back the new typing environments into one that would be suitable for the continuation of the transformation.

4.4.1 Intermediate Results on Renaming Maps

Because we aim to provide guarantees on the renaming operations, we will come to work with renamed typing environments *i.e.*, typing environments where all the renamings of a renaming map are typed with the type of the renamed variable. This allows us to create *updated* typing environments after assigning to each renaming its renamed variable. However, future operations are done on these environments, and we need to keep track of both the previous and the current environments. Contrary to the original *execution*, the renamed branch should be typed as if there were no secret conditional as context. To do so, we devise in Definition 41 an operator which removes a dependence to a program point p.

Definition 41: Declassifying

$$\downarrow_{\{p\}} \Gamma(x) = \begin{cases} \mathbf{I}(l \setminus \{p\}) & \text{if } \Gamma(x) = \mathbf{I}(l) \\ \Gamma(x) & \text{otherwise} \end{cases}$$

To keep track of renamings on typing environments, we note $\rho^p(\Gamma, \Gamma')$ the renaming of an initial typing environment Γ by a renaming ρ, updated to Γ', and ignoring the

program point p, and we define it formally in Definition 42. In particular, note that $\rho(\Gamma, \Gamma')(\rho(x)) = \Gamma(x)$ and that if $x \notin \mathcal{V}_\rho, \rho(\Gamma, \Gamma')(x) = \Gamma(x)$.

Definition 42: Renaming a typing environment

$$\rho^p(\Gamma_i, \Gamma)(x') = \begin{cases} \downarrow_{\{p\}} \Gamma(\rho^{-1}(x)) & \text{if } x \in \mathcal{V}_\rho \\ \Gamma_i(x) & \text{otherwise} \end{cases}$$

We can also update a renaming :

$$\rho^p(\Gamma_i, \Gamma)[x \mapsto \downarrow_{\{p\}} \tau] = \begin{cases} \rho^p(\Gamma_i, \Gamma[\rho^{-1}(x) \mapsto \tau]) & \text{if } x \in \mathcal{V}_\rho \\ \rho^p(\Gamma_i[x \mapsto \downarrow_{\{p\}} \tau], \Gamma) & \text{otherwise} \end{cases}$$

An update can also be taken in a less literal way: if $x \notin \mathcal{V}^\rho$, we can choose a fresh x' such that

$$\rho^p(\Gamma_i, \Gamma)[x \mapsto \downarrow_{\{p\}} \tau] = \rho[x' \mapsto x]^p(\Gamma_i, \Gamma[x' \mapsto \tau]) \cup \Gamma_u$$

where Γ_u is the reminder of the renaming *i.e.*, $\Gamma_u = \Gamma_{id}[\rho(x') \mapsto \Gamma(x')]$. Note that when $\rho(x') = x$, we can forego Γ_u.

The main intuition of this definition is that we keep track of non-renamed variables in the first environment Γ, and the renamed variables are updated in Γ'. Along with the definition of renaming for an environment, we also define what it means for such a map to be updated. The intuition is that if we update a renamed environment on a renamed variable, we can instead update the renaming map. Unfortunately, updating the renaming map means that we *forget* about the previous renaming. We introduce the environment Γ_u to compensate for that. Still, to ensure the preservation of information on the original variables, the renamed environment should not differ from the original on non-renamed variables. At this end, we define the well-formation of a couple of environments w.r.t to a map ρ in Definition 43.

Definition 43: Well formation of environments for renaming

We say that Γ_i and Γ are well-formed with respect to a renaming map ρ (noted $\Gamma_i \equiv_\rho \Gamma$) if for any non-renamed variable x, Γ and Γ_i type x the same, that is

$$x \notin \mathcal{V}^\rho \implies \Gamma_i(x) = \Gamma(x)$$

In particular, if $\rho(e) = e$, then e types the same way in Γ_i and Γ.

However, because the renaming is done on two branches at the same time, we end up working with two renaming maps at the same time, and thus on an environment twice renamed. Fortunately, given certain restrictions on the renaming maps, the composition of renamings on an environment is commutative. One of these restrictions is for the renaming maps to be *negligibly interfering*, that is, if both maps have a renaming into a variable x, both maps rename the same variable into x. This is explicated in Definition 44.

Definition 44: Negligible Interference

Let ρ_1 and ρ_2 be two renaming maps. We say that ρ_1 and ρ_2 are negligibly interfering if for any x such that $x \in \mathcal{V}_{\rho_1} \wedge x \in \mathcal{V}_{\rho_2}$, we have $\rho_1^{-1}(x) = \rho_2^{-1}(x)$.

We state in Lemma 18 what composition means for renamings. By assigning a *current* environment to each renaming, we can ensure that combining two maps, or more, is done without loss of information. The constraint of negligible interference ensures that the order does not matter, while the constraints on $\mathcal{V}^{\rho_{1,2}}$ and $\mathcal{V}_{\rho_{1,2}}$ guarantee that a renaming cannot be renamed by the other. We say that two renamings following these last two constraints are *compatible*.

Lemma 18: Composition of renamings on a typing environment

Let ρ_1 and ρ_2 be two negligibly interfering maps such that $\mathcal{V}^{\rho_1} \cap \mathcal{V}_{\rho_2} = \emptyset$ and $\mathcal{V}^{\rho_2} \cap \mathcal{V}_{\rho_1} = \emptyset$, and Γ_i, Γ_1 and Γ_2 be three typing environments. We have, for any p,

$$\rho_1^p(\rho_2^p(\Gamma_i, \Gamma_2), \Gamma_1) = \rho_2^p(\rho_1^p(\Gamma_i, \Gamma_1), \Gamma_2)$$

Proof Outline. The proof is by case analysis on $x \in \mathcal{V}^{\rho_i}, i \in \{1, 2\}$ in the definition of the renaming of an environment. In some cases, the negligible interference ensures that both

renamings are equal and that the order of call does not matter. This, along with the fact that the constraints guarantee that there are no *intermediate* renamings, applying the definition in Definition 42 is enough.

For example, if there exists x_1 such that $\rho_1(x_1) = x$ but not such x_2 for ρ_2, we know thanks to our constraints that there are no x_2' such that $\rho_2(x_2') = x_1$, and thus, by definition, we have

$$\rho_1^p(\rho_2^p(\Gamma, \Gamma_2), \Gamma_1)(x) =_{\downarrow_{\{p\}}} \Gamma_1(x_1)$$

$$\rho_2^p(\rho_1^p(\Gamma, \Gamma_1), \Gamma_2))(x) = \rho_1^p(\Gamma, \Gamma_1)(x) =_{\downarrow_{\{p\}}} \Gamma_1(x_1)$$

□

Furthermore, we can show that if two renaming maps are created following the method described in Section 2.1, they follow the constraints detailed above. This is formalized in Lemma 19.

Lemma 19: Characteristics of initialized maps

Let v, v_t and v_e be three sets of variables such that $v_e \subseteq v$ and $v_t \subseteq v$, and ρ_t and ρ_e be two renaming maps such that

$$\rho_t = init(v_t, v) \qquad \rho_e = init(v_e, v \cup \mathcal{V}_{\rho_t})$$

We have that ρ_t and ρ_e are non-interfering and compatible.

Proof Outline. By definition of *init*, the maps are non-interfering. And, $\mathcal{V}^{\rho_t} = v_t \subseteq v$, and by definition of *init*, $\mathcal{V}_{\rho_e} \cap v = \emptyset$, hence $\mathcal{V}^{\rho_t} \cap \mathcal{V}_{\rho_e} = \emptyset$ □

4.4.2 Intermediate Results on the Initializing Statements

The first step of the delayed if-conversion transformation is the initialization of the renaming maps used further along the way. As stated in Lemma 19, the maps initialized by and for the transformation follow the constraints of Lemma 18, and it will be usable for the transformation. Moreover, we show in Lemma 20 that such maps are *sequenceable* using the *seq* operator into a typable statement, and that the resulting typing environment is the renaming of the initial environment. The intuition of the proof is that each single assignment is independent of others (because of the fresh nature of renamings), and thus can be chained.

Lemma 20: Security of pre renaming

Let ρ be a fresh renaming map from a set of variable V within a larger set V_p *i.e.*, $V \subseteq V_p \wedge \rho = init(V, V_p)$ and s a statement such that $s = seq(\{\rho(x) = x \mid x \in \mathcal{V}^\rho\})$. Then, we have that, for any $p \notin \Gamma$

$$\Delta, \mathbf{L} \vdash \Gamma\{s\}\rho^p(\Gamma, \Gamma)$$

for some Δ, Γ.

Proof. We start by noticing that by definition of *init*, $\mathcal{V}^\rho = V$. Then, by induction over V.

Case 5 $(V = \varnothing)$ We then have $\rho = id$ and $pre = seq(\varnothing) = \mathbf{skip}$, and, because $p \notin \Gamma$,

$$\Delta, \kappa \vdash \Gamma\{pre\}\Gamma$$

Case 6 $(V = V' \cup \{y\})$ Because *seq* defines an arbitrarily chosen sequence, we can choose to define $seq(\{\rho(x) = x | x \in V\})$ as

$$seq(\{\rho(x) = x | x \in V'\}); (\rho(y) = y)$$

We can also define a map ρ' such that $\rho = \rho'[y \mapsto \rho(y)]$.
By induction, we have that

$$\Delta, \kappa \vdash \Gamma\{seq(\{\rho(x) = x | x \in V'\})\}\rho'^p(\Gamma, \Gamma)$$

By the sequence rule,

$$\Delta, \kappa \vdash \Gamma\{pre\}\rho'^p(\Gamma, \Gamma)[\rho(y) \mapsto \Gamma(y)]$$

Because $p \notin \Gamma$, we have $\Gamma(y) =\downarrow_{\{p\}} \Gamma(y)$, and, by definition, $\rho'^p(\Gamma, \Gamma)[\rho(y) \mapsto \Gamma(y)] = \rho'[y \mapsto \rho(y)]^p(\Gamma, \Gamma[y \mapsto \Gamma(y)])$, thus

$$\Delta, \kappa \vdash \Gamma\{pre\}\rho^p(\Gamma, \Gamma)$$

□

Because we are able to transform one such *pre* statement, we can look at what happens when we sequence two of them, as we do in the transformation. We state in Lemma 21 that typing the sequence of two initializing statements for two statements within the same program yields the renaming of the typing environment by both maps. Because there is no real added value to keeping twice the original environment, we use Lemma 18 to keep track of only one.

Lemma 21: Sequence of initializing statements

Let s_t and s_e be two branches of the same conditional, that is $mod(s_t) \subseteq v$ and $mod(s_e) \subseteq v$ where v is the overall set of variables of the conditional/program. If we define ρ_t and ρ_e such that $\rho_t = init(mod(s_t), v)$ and $\rho_e = init(mod(s_e), v \cup \mathcal{V}_{\rho_t})$, we can define two statements pre_t and pre following $pre_i = seq(\{\rho_i(x) = x \mid x \in \mathcal{V}^{\rho_i}\})$.

Then, for any starting typing environment Γ such that $p \notin \Gamma$, the sequence of $pre_t; pre_e$ is typable, and we have

$$\Delta, \mathbf{L} \vdash \Gamma\{pre_t; pre_e\}\rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma)$$

Proof. By Lemma 19, we know that ρ_t and ρ_e are non-interfering and compatible. Thus, they are negligibly interfering as well. By Lemma 20, we have

$$\Delta, \mathbf{L} \vdash \Gamma\{pre_t\}\rho_t^p(\Gamma, \Gamma)$$

$$\Delta, \mathbf{L} \vdash \rho_t(\Gamma, \Gamma)\{pre_e\}\rho_e^p(\rho_t^p(\Gamma, \Gamma), \rho_t^p(\Gamma, \Gamma))$$

Furthermore, because ρ_t and ρ_e are compatible, we can simplify $\rho_e^p(\rho_t^p(\Gamma, \Gamma), \rho_t^p(\Gamma, \Gamma))$ to $\rho_e^p(\rho_t^p(\Gamma, \Gamma), \Gamma)$. Indeed, if a variable is a renaming by ρ_e, the renamed variable cannot be a renaming by ρ_t, and the second $\rho_t^p(\Gamma, \Gamma)(x)$ would always yield $\Gamma(x)$. Thus, by the sequence rule and Lemma 18,

$$\Delta, \mathbf{L} \vdash \Gamma\{pre_t; pre_e\}\rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma)$$

□

4.4.3 Intermediate Results on Branch Renaming

The first pass aimed to introduce the renamings that we use throughout the transformation. On the other hand, the Branch Renaming transformation is very local and impacts only a specific branch of the p conditional. Firstly, we show in Lemma 22 that given a renaming map over a well-typed statement, any well-typed expressions within this statement is still typable after renaming. This is nearly immediate from the definition of a renaming of an environment.

Lemma 22: Expression renaming

Let ρ be a renaming map of a statement containing e, a well-typed expression, *i.e.*,

$$var(e) \cap \mathcal{V}_\rho = \emptyset \quad \Delta, \Gamma \vdash e : \tau, l$$

for some τ, l

Then, $\rho(e)$ is also well-typed, and we have, for two well-formed environments Γ_i and Γ w.r.t ρ such that $p \notin \Gamma_i$,

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e) :\downarrow_{\{p\}} \tau, l$$

Proof. By induction over the syntax of e

Case 7 $(e = x)$ If $x \in \mathcal{V}^\rho, \rho(e) = \rho(x)$. By definition, $\rho^p(\Gamma_i, \Gamma)(\rho(x)) =\downarrow_{\{p\}} \Gamma(x)$, hence

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e) :\downarrow_{\{p\}} \tau, l$$

Otherwise, $\rho(e) = x$, and there are no renamings to x, so $\rho^p(\Gamma_i, \Gamma)(x) = \Gamma_i(x) = \Gamma(x)$ by well-formation. And, $p \notin \Gamma_i$, so $p \notin \Gamma(x)$, and $\Gamma(x) =\downarrow_{\{p\}} \Gamma(x)$, and

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e) :\downarrow_{\{p\}} \tau, l$$

Case 8 $(e = c)$ By definition, $\rho(c) = c$, and $\downarrow_{\{p\}} \mathbf{L} = \mathbf{L}$, hence

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e) :\downarrow_{\{p\}} \tau, l$$

Case 9 $(e = e_1 \oplus e_2)$ By definition, $\rho(e) = \rho(e_1) \oplus \rho(e_2)$.
We conclude by induction and the $\oplus$ rule.

Case 10 $(e = e_1?e_2{:}e_3)$ By definition, $\rho(e) = \rho(e_1)?\rho(e_2){:}\rho(e_3)$.
We conclude by induction and rule for the conditional expression

Case 11 $(e = t[e_1])$ By definition, $\rho(e) = t[\rho(e_1)]$.
We conclude by induction and the memory read rule.

□

For statements, we show in Lemma 23 that if a branch s is typable using Γ and Γ', its renaming by a renaming map ρ is still typable using $\rho^p(\Gamma_i, \Gamma)$ and $\rho^p(\Gamma_i, \Gamma')$ if Γ_i is not dependent on any conditional of s, and that if the couple (Γ_i, Γ) is well-formed for a renaming, the same goes for (Γ_i, Γ'). This is done by induction over the statement s. Most cases are trivial by induction hypothesis. But, the memory write relies on the intuition that because the write was being done within a high conditional, the array is **H** anyway, and augmenting its type should not have any impact.

Lemma 23: Security of branch renaming

Let ρ be a renaming map on the set of variables of a well-typed s statement *i.e.*,

$$\mathcal{V}^\rho = mod(s) \quad \Delta, \kappa \vdash^{@p} \Gamma\{s\}\Gamma'$$

with some Δ, κ, Γ and Γ'.

If $p \in \kappa$, $p \notin s$, $\rho(h) = h$, $p \notin \Gamma_i$, $\forall p' \in high(s), p' \notin \Gamma_i$, and $\Gamma_i \equiv_\rho \Gamma$, we define $\kappa_p = \downarrow_{\{p\}} \kappa$, and we have :

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{Rn_h^\rho(s)\}\rho^p(\Gamma_i, \Gamma') \qquad \Gamma_i \equiv_\rho \Gamma'$$

Proof. By induction over the type derivation.

Case 12 $(P = \textbf{skip})$ By definition of Rn_h^ρ, we have that $P' = Rn_h^\rho(P) = P$.
Furthermore, by the **skip** rule, $\Delta, \kappa_p \vdash \Gamma\{\textbf{skip}\}\Gamma$, and $\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{\textbf{skip}\}\rho^p(\Gamma_i, \Gamma)$

Case 13 $(P = (x = e)_g^r)$ By definition of Rn_h^ρ, we have that $P' = Rn_h^\rho(P) = (\rho(x) = \rho(e))$.

Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{(x = e)^r_g\}\Gamma'$ gives us that there exists τ such that

$$\Delta, \Gamma \vdash e : \tau, r \quad \Gamma' = \Gamma[x \mapsto \tau \sqcup \kappa]$$

By Lemma 22, we have that

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e) :\downarrow_{\{p\}} \tau, r$$

Hence, by the assignment rule, we have

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{\rho(x) = \rho(e)\}\rho^p(\Gamma_i, \Gamma)[\rho(x) \mapsto (\downarrow_{\{p\}} \tau) \sqcup \kappa_p]$$

The definition of κ_p gives us $(\downarrow_{\{p\}} \tau) \sqcup \kappa_p =\downarrow_{\{p\}} (\tau \sqcup \kappa)$, and by definition of $\rho^p(\Gamma_i, \Gamma)$, we can rewrite this as

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{\rho(x) = \rho(e)\}\rho^p(\Gamma_i, \Gamma[x \mapsto \tau \sqcup \kappa])$$

And,

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{\rho(x) = \rho(e)\}\rho^p(\Gamma_i, \Gamma')$$

Furthermore, the only difference between Γ' and Γ is on a single variable $x \in \mathcal{V}^\rho$, so Γ_i and Γ' are well-formed w.r.t to ρ.

Case 14 $(P = (t[e_1] = e_2)^r_g)$ By definition of Rn^ρ_h, we have that $P' = Rn^\rho_h(P) = (t[\rho(e_1)] = h?\rho(e_2){:}t[\rho(e_1)])$

Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{(t[e_1] = e_2)^r_g\}\Gamma'$ gives us that there exists τ_1, τ_2, l_1, l_2 such that

$$\Delta, \Gamma \vdash e_1 : \tau_1, l_1 \quad \Delta, \Gamma \vdash e_2 : \tau_2, l_2$$

$$\tau_1 \sqcup \tau_2 \sqcup (\kappa \ltimes_{\mathbf{L}} \mathbf{H}) \sqsubseteq \Delta(t)$$

$$\tau_1 = \mathbf{I}(l'_1) \qquad r = l_1 \cup l'_1 \cup l_2$$

$$\Gamma' = \Gamma$$

Because $\kappa \neq \mathbf{L}$, we have $\kappa \ltimes_{\mathbf{L}} \mathbf{H} = \mathbf{H}$, hence

$$\mathbf{H} \sqsubseteq \Delta(t)$$

By Lemma 22, we have that

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e_1) :\downarrow_{\{p\}} \tau_1, l_1$$

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(e_2) :\downarrow_{\{p\}} \tau_2, l_2$$

And, by the rule for a conditional expression,

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash h?\rho(e_2){:}t[\rho(e_1)] : \Gamma(h) \sqcup \downarrow_{\{p\}} \tau_1 \sqcup \downarrow_{\{p\}} \tau_2, l_1 \cup l_2$$

By definition of $\sqsubseteq$, $\Gamma(h) \sqcup \downarrow_{\{p\}} \tau_1 \sqcup \downarrow_{\{p\}} \tau_2 \sqsubseteq \mathbf{H}$, thus

$$\Gamma(h) \sqcup \downarrow_{\{p\}} \tau_1 \sqcup \downarrow_{\{p\}} \tau_2 \sqsubseteq \Delta(t)$$

By the array assignment rule,

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{(t[\rho(e_1)] = h?\rho(e_2){:}t[\rho(e_1)])^l_g\}\rho^p(\Gamma_i, \Gamma')$$

And, $\Gamma' = \Gamma$, so the well-formation is preserved

Case 15 $(P = (s_1; s_2)^r_g)$ By definition of Rn^ρ_h, we have that $P' = Rn^\rho_h(P) = Rn^\rho_h(s_1); Rn^\rho_h(s_2)$. Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{(s_1; s_2)^r_g\}\Gamma'$ gives us that there exists $\Gamma_1, g_1, g_2, r_1, r_2$ such that

$$\Delta, \kappa \vdash^{@p} \Gamma\{(s_1)^{r_1}_{g_1}\}\Gamma_1$$

$$\Delta, \kappa \vdash^{@p} \Gamma_1\{(s_2)^{r_2}_{g_2}\}\Gamma'$$

$$r = r_1 \cup r_2 \wedge g = g_1 \cup g_2$$

By induction, we have that

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma)\{(Rn^\rho_h s_1)^{r_1}_{g_1}\}\rho^p(\Gamma_i, \Gamma_1)$$

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma_1)\{(Rn^\rho_h s_1)^{r_1}_{g_1}\}\rho^p(\Gamma_i, \Gamma')$$

We conclude by the sequence rule.

Case 16 $(P = \mathbf{if}^{p'}\ h'\ \mathbf{then}\ s_t\ \mathbf{else}\ s_e\ \mathbf{with}\ p' \neq p)$ By definition of Rn^ρ_h, we have that $P' = Rn^\rho_h(P) = \mathbf{if}^{p'}\ \rho(h')\ \mathbf{then}\ Rn^\rho_h(s_t)\ \mathbf{else}\ Rn^\rho_h(s_e)$

Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{\mathbf{if}^{p'}\ h'\ \mathbf{then}\ s_t\ \mathbf{else}\ s_e\}\Gamma'$ give us that there exists $\tau_c, r_c, \kappa', \Gamma_3, r_t, g_t, r_e, g_e$ such that

$$\Delta, \Gamma \vdash c : \tau, r_c$$

$$\kappa' = \kappa \sqcup (\tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p'\}))$$

$$\Delta, \kappa \vdash^{@p} \Gamma\{(s_t)^{r_t}_{g_t}\}\Gamma_1 \quad \Delta, \kappa \vdash^{@p} \Gamma\{(s_e)^{r_e}_{g_e}\}\Gamma_2$$

$$\Delta, \kappa \vdash^{@p} \Gamma_1 \sqcup \Gamma_2\{\mathbf{skip}\}\Gamma'$$

$$r = r_t \cup r_e \cup r_c$$

$$g = g_t \cup g_e \cup \tau \ltimes_{\mathbf{L}} \{p'\}$$

By induction,

$$\Delta, \kappa'_p \vdash \rho^p(\Gamma_i, \Gamma)\{(Rn^\rho_h(s_t))^{r_t}_{g_t}\}\rho(\Gamma_i, \Gamma_1)$$

$$\Delta, \kappa'_p \vdash \rho^p(\Gamma_i, \Gamma)\{(Rn^\rho_h(s_t))^{r_t}_{g_t}\}\rho(\Gamma_i, \Gamma_2)$$

By Lemma 22,

$$\Delta, \rho^p(\Gamma_i, \Gamma) \vdash \rho(c) :\downarrow_{\{p\}} \tau, r$$

By the skip rule, we have that $\Gamma_1 \sqcup \Gamma_2 = \Gamma'$, and $\rho^p(\Gamma_i, \Gamma_1) \sqcup \rho^p(\Gamma_i, \Gamma_2) = \rho^p(\Gamma_i, \Gamma_1 \sqcup \Gamma_2)$. And, because both couples (Γ_i, Γ_1) and (Γ_i, Γ_2) are well-formed, the union is well-formed.

Note that because we use $\downarrow_{\{p\}} \tau$ instead of τ, we need to type $Rn^\rho_h(s)$ with a $\kappa''_p \sqsubseteq \kappa'_p$. By the same reasoning, the *high* annotation of P may be stricter than its original one. Nonetheless, we are able to conclude by the if rule.

Case 17 ($P = (\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s)^r_g$) By definition of Rn^ρ_h, we have that $P' = Rn^\rho_h(P) = \mathbf{for}\ \rho(i)\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ Rn^\rho_h(s)$

Moreover, $\Delta, \kappa \vdash^{@p} \Gamma\{\mathbf{for}\ i\ \mathbf{from}\ c_1\ \mathbf{to}\ c_2\ \mathbf{do}\ s\}\Gamma'$ gives us that there exists Γ_1 such that

$$\Delta, \kappa \vdash^{@p} \Gamma'[i \mapsto \kappa]\{(s)^r_g\}\Gamma_1$$

$$\Gamma \sqsubseteq \Gamma_1 \quad \Gamma_1 \sqsubseteq \Gamma' \quad \uparrow_{high(s)} \Gamma' = \Gamma'$$

By definition, $\rho^p(\Gamma_i, \Gamma')[i \mapsto \kappa_p] = \rho^p(\Gamma_i, \Gamma'[i \mapsto \kappa])$, and, by induction,

$$\Delta, \kappa_p \vdash \rho^p(\Gamma_i, \Gamma'[i \mapsto \kappa])\{Rn^\rho_h(s)\}\rho^p(\Gamma', \Gamma_1)$$

And, $\rho^p(\Gamma_i, \Gamma) \sqsubseteq \rho^p(\Gamma_i, \Gamma'), \rho^p(\Gamma_i, \Gamma_1) \sqsubseteq \rho^p(\Gamma_i, \Gamma')$ by monotony of $\rho^p()$, and $\uparrow_{high(s)}$

$\rho^p(\Gamma_i, \Gamma') = \rho^p(\Gamma_i, \Gamma')$ because $\uparrow_{high(s)} \Gamma' = \Gamma'$, and $\forall p' \in high(s), p' \notin \Gamma_i$.

We conclude by the for rule.

□

In the final transformation, we transform the two branches **then** and **else**. Thus, we need to be able to type the sequence of the two renamed statements. We state exactly this in Lemma 24, and prove it by using twice Lemma 23, and Lemma 18 to switch the order of the maps.

Lemma 24: Sequence of branch renamings

Let s_t, s_e be two statements within a program containing all variables in v, and ρ_t, ρ_e be two renaming maps such that $\rho_t = init(mod(s_t), v)$ and $\rho_e = init(mod(s_e), v \cup \mathcal{V}_{\rho_t})$. For any variable h, if s_t and s_e are typable for a typing environment Γ to environments Γ_1, Γ_2, the sequence of renamings $Rn_h^{\rho_t}(s_t); Rn_{\neg h}^{\rho_e}(s_e)$ is typable by the renaming of Γ by ρ_t and ρ_e, that is

$$\Delta, \kappa \vdash \rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma)\{Rn_h^{\rho_t}(s_t); Rn_{\neg h}^{\rho_e}(s_e)\}\rho_t^p(\rho_e^p(\Gamma, \Gamma'), \Gamma')$$

Proof. By Lemma 23, we have that $\Delta, \kappa \vdash \rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma)\{Rn_h^{\rho_t}(s_t)\}\rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma_1)$

By Lemma 19, ρ_t and ρ_e are compatible and negligibly interfering, so we apply Lemma 18 to get $\rho_t^p(\rho_e^p(\Gamma, \Gamma), \Gamma_1) = \rho_e^p(\rho_t^p(\Gamma, \Gamma_1), \Gamma)$

By Lemma 23, we have $\Delta, \kappa \vdash \rho_e^p(\rho_t^p(\Gamma, \Gamma_1), \Gamma)\{Rn_h^{\rho_e}(s_e\}\rho_e^p(\rho_t^p(\Gamma, \Gamma_1), \Gamma_2)$

We conclude by Lemma 19, and the sequence rule. □

4.4.4 Intermediate results on Nxt_h

The Nxt_h transformation is the trickier one of them. Indeed, we track two renaming maps to simulate the execution of both branches. However, as we stated in Section 2.4, we associate the relation between the two maps, and the typing environment. We define in Definition 45 a property binding two renaming maps, a typing environment, and a program point p so that the maps agree on a variable if and only if this variable is not dependent on x in Γ. This well-formation property will serve as a guide throughout the proof.

Definition 45: Well-formation of environment for renamings

A typing environment Γ is well-formed with respect to a program point p and two renaming maps ρ_1 and ρ_2 (written $\rho_1 \equiv^p_\Gamma \rho_2$)

if the following holds

$$\forall x, \rho_1(x) = \rho_2(x) \leftrightarrow p \notin \Gamma(x)$$

The renaming of the **next** branch uses two main operations on maps to allow the use of two simultaneous renaming maps: the phi-merging ϕ, and the join $\bowtie$. The phi-merging aims to readjust a renaming to another one. We state in Lemma 25 that for two well-defined maps w.r.t to an initial environment, the phi-merging of those maps does exactly that. Because ϕ uses the *seq* operator, the proof is similar to the one of Lemma 20, by applying the definition of an update on each assignment.

Lemma 25: ϕ-typing

Let ρ_1 and ρ_2 be two renaming maps, such that Γ_i and Γ are well-formed with respect to them, such that $\mathcal{V}^{\rho_2} \subseteq \mathcal{V}^{\rho_1}$. The sequence of statements represented by $\phi(\rho_1, \rho_2)$ is typable, that is

$$\Gamma_i \equiv_{\rho_1} \Gamma \wedge \Gamma_i \equiv_{\rho_2} \Gamma \implies \Delta, \kappa \vdash \rho_2^p(\Gamma_i, \Gamma)\{\phi(\rho_1, \rho_2)\}\rho_1^p(\Gamma_i, \Gamma) \cup \Gamma_u$$

for some Δ, κ, and Γ, and where Γ_u is a memory of retyped renamings of ρ_2 over Γ.

Proof. The proof is by induction over the set $\mathcal{V}^{\rho_1}$.

If $\mathcal{V}^{\rho_1}$, then $\rho_1 = \rho_2 = id$, and $\phi(\rho_1, \rho_2) = \textbf{skip}$. Hence, $\Delta, \kappa \vdash id(\Gamma_i, \Gamma)\{\textbf{skip}\}id(\Gamma_i, \Gamma)$.

Otherwise, there is a x, x_1, and ρ_1' such that $\rho_1 = \rho_1'[x \mapsto x_1]$. There are then two cases: either x is renamed by ρ_2 or not. If $x \in \mathcal{V}^{\rho_2}$, there exists x_2 and ρ_2' such that $\rho_2 = \rho_2'[x \mapsto x_2]$. And, we can choose how ϕ works such that $\phi(\rho_1, \rho_2) = \phi(\rho_1', \rho_2'); (\rho_1(x) = \rho_2(x))$. We conclude by induction hypothesis and the sequence rule. However, if x is not renamed by y, we have $\phi(\rho_1, \rho_2) = \phi(\rho_1', \rho_2); (\rho_1(x) = x)$. Similarly, we conclude by the induction hypothesis and the sequence rule (thanks to $\Gamma_i \equiv_{\rho_2} \Gamma$). In both cases, the definition of an update to a renaming of an environment is what introduces the reminder environment Γ_u. □

The second operation ($\bowtie$) is more of a union between two maps: it keeps common

renamings, and generates fresh ones to avoid conflicts. Thus, Lemma 26 guarantees that any interesting property shared by two maps, the join of them possesses it too. The proof is immediate by developing the definition of $\bowtie$ within the definition of the property.

Lemma 26: Property preservation by $\bowtie$

Let ρ_1 and ρ_2 be two renaming maps. If any of the following properties is shared by ρ_1 and ρ_2, it affects $\rho_1 \bowtie \rho_2$ too :

— Negligible interference with another renaming

— Compatibility with another renaming

— Well-formation w.r.t one environment, one renaming, and one program point

Proof Outline. The proof for each property is done by developing the definition of $\bowtie$. The introduction of fresh variables ensures that no renaming of those names are already present within the system.

For example, if we look at the negligible interference, we would define a map ρ_3 such that ρ_1 and ρ_3, as well as ρ_2 and ρ_3 are negligibly interfering. Then, if we take a $x \in \mathcal{V}^{\rho_1 \bowtie \rho_2}$, either $x \in \mathcal{V}^{\rho_1}$, $x \in \mathcal{V}^{\rho_2}$ or $x \notin \mathcal{V}^{\rho_3}$. The last case is immediate, while the others use the negligible interference of ρ_1 (or ρ_2) and ρ_3 to conclude. □

As we have seen earlier when defining the update to a renaming of an environment, updating a map renaming an environment *hides* the previous renaming. To ensure that the types of these renamings are not forgotten, we usually add a reminder environment Γ_u. By its definition, this Γ_u should not type any name used by the program or its renamings, and we say that the environment is harmless. We define precisely what harmlessness means in Definition 46.

Definition 46: Harmless environment

We say that an environment Γ is harmless to a renaming map ρ if, for any x, we have $\Gamma(x) = \mathbf{L}$ or $x \notin \mathcal{V}^{\rho} \cup \mathcal{V}_{\rho}$.

Similarly, an environment Γ is harmless to a statement s if, for any x, we have $\Gamma(x) = \mathbf{L}$ or $x \notin var(s)$.

By tracking two maps at once, and especially by using a typing environment twice renamed, it is hard to apply updates uniformly. Indeed, updating a twice-renamed environment usually only affects the surface-level map, not the *hidden* one. The main reason

is that these updates have no effects on the second map: the updated value is always caught by the first. Hence, we state in Lemmas 27 and 28 that we can, if we wish, apply those updates for free to the second map, given certain restrictions, mainly that the updated variable is a renaming of or is renamed by the first map.

Lemma 27: Hidden Renaming

Let ρ_1 and ρ_2 be two compatible renamings. Then,

$$\forall x, x', x \in \mathcal{V}_{\rho_1} \implies \rho_1^p(\rho_2^p(\Gamma_i, \Gamma), \Gamma) = \rho_1^p(\rho_2[x' \mapsto x]^p(\Gamma_i, \Gamma[x' \mapsto \Gamma(x')]), \Gamma)$$

Proof Outline. By case analysis. If we look at a renaming of ρ_1, then by compatibility, the renamed variable is not a renaming of ρ_2 nor $\rho_2[x' \mapsto x]$ (the renamed variable cannot be equal to x, which is a renaming). Thus, the two expressions are equivalent. Otherwise, the variable is not a renaming of ρ_1, so not x, and the update on ρ_2 can be ignored. □

Lemma 28: Hidden Retyping

Let ρ_1 and ρ_2 be two compatible renamings. Then,

$$\forall x, \tau, x \in \mathcal{V}^{\rho_1} \implies \rho_1^p(\rho_2^p(\Gamma_i, \Gamma), \Gamma') = \rho_1^p(\rho_2^p(\Gamma_i, \Gamma[x' \mapsto \tau]), \Gamma')$$

Proof Outline. Similarly to Lemma 27, thanks to the compatibility of the renamings, we never access $\Gamma[x' \mapsto \tau](x')$. □

Finally, we can express the security of the next transformation: given two maps and a statement, the renaming of this statement by those two maps is typable. In practice, the lemma requires and ensures some properties on the maps or the typing environments, but the lemma really boils down to this: the renaming is typable. To prove it, we reason by induction over the statement. The base cases are solved by clever use of the definition of a renaming, and Lemma 22 (on expression renaming). The complex cases make use of the Lemmas 25 and 26 to ensure that the created maps are well-defined.

Lemma 29: Security of the next transformation

Let ρ_t, ρ_e be two negligibly interfering compatible renaming maps, P be a typable **next** branch, and Γ_i an environment *i.e.*,

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{P\}\Gamma' \qquad high(P) = \emptyset$$

for some Δ, p, Γ and Γ' such that $\rho_t \equiv^p_\Gamma \rho_e$, $\Gamma_i \equiv_{\rho_t} \Gamma$, $\Gamma_i \equiv_{\rho_e} \Gamma$ and $Nxt^{\rho_t,\rho_e}_h(P) = (\rho'_t, \rho'_e, P')$.

For any Γ_u harmless to ρ_t, ρ_e and s, there exists a Γ'_u such that

— ρ'_t and ρ'_e are negligibly interfering and compatible

— $\Delta, \mathbf{L} \vdash \rho^p_t(\rho^p_e(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{P'\}\rho'^p_t(\rho'^p_e(\Gamma_i, \Gamma'), \Gamma') \cup \Gamma'_u$

— $\rho'_t \equiv^p_{\Gamma'} \rho'_e$

— $\Gamma_i \equiv_{\rho'_t} \Gamma'$ and $\Gamma_i \equiv_{\rho'_e} \Gamma'$

— Γ'_u is harmless to ρ'_t and ρ'_e

Proof. For simplicity's sake, we name H_0 the fact that the maps are negligibly interfering and compatible, H_1 the hypothesis $\Delta, \mathbf{L} \vdash^{@p} \Gamma\{P\}\Gamma'$, H_2 the hypothesis $\rho_t \equiv^\rho_\Gamma \rho_e$, H_3 and H_4 the well-formation of ρ_t and ρ_e w.r.t to Γ_i, and H_5 the harmlessness of Γ_u. Similarly, we name R_i the corresponding result for each of the hypotheses.

The proof is by induction over the typing derivation.

Case 18 **($P = \mathbf{skip}$)** Because $Nxt^{\rho_t,\rho_e}_h(\mathbf{skip}) = (\rho_t, \rho_e, \mathbf{skip})$, most of the results are immediate. Moreover, H_1 gives us that $\Gamma' = \Gamma$, and R_1 follows, by taking $\Gamma'_u = \Gamma_u$.

Case 19 **($P = (x = v)$)** We have two cases here :

— If $\rho_t(v) = \rho_e(v)$, then $Nxt^{\rho_t,\rho_e}_h(x = v) = (\rho_t[x \mapsto x'], \rho_e[x \mapsto x'], (x' = \rho_t(v))$.

We prove R_0 by choosing a $y \in \mathcal{V}_{\rho'_t}$. Either $y \in \mathcal{V}_{\rho_t}$ or $y = x'$. In the first case, H_0 gives us that if $y \in \mathcal{V}_{\rho_e}$, and we conclude. In the other case, we know that $\rho'_e(x') = x$. Moreover, x' is, by definition, not renamed by ρ'_t and ρ'_e, and ρ_t and ρ_e being compatible, ρ'_t and ρ'_e are too.

The typing of H_1 gives us that $\Delta, \Gamma \vdash v : \tau, l$, and $\Gamma' = \Gamma[x \mapsto \tau]$. By Lemma 22, we have $\Delta, \rho^p_t(\rho^p_e(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(v) :\downarrow_{\{p\}} \tau, l$. Hence, per the typing rule of assignment,

and because Γ_u is harmless :

$$\Delta, \kappa \vdash \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{x' = \rho_t(v)\}\rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma)[x' \mapsto \downarrow_{\{p\}} \tau] \cup \Gamma_u$$

And, by definition, $\rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma)[x' \mapsto \downarrow_{\{p\}} \tau] = \rho_t'^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma[x \mapsto \tau]) \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)]$. By Lemmas 27, 28, we can introduce the renaming of x' and retyping of x within the renaming of ρ_e, that is $\rho_t'^p(\rho_e'^p(\Gamma_i, \Gamma'), \Gamma') \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)]$. By defining $\Gamma'_u = \Gamma_u \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)]$, that proves R_1. Furthermore, if $\rho(x) = x$, we can redefine $\Gamma'_u = \Gamma_u$ without any loss of precision, and otherwise, $\rho(x)$ is not used in ρ'_t nor ρ'_e, nor in the program, proving R_5.

Because ρ'_t (resp. ρ'_e) is simply an update on ρ_t (resp. ρ'_e), we have, for any y, that $\rho'_t(y) = \rho'_e(y)$ if and only if $\rho_t(y) = \rho_e(y)$ or $y = x$. In the first case, H_2 is enough to prove R_2, by definition of Γ', while in the second, $\rho_t(v) = \rho_e(v)$ ensures that $p \notin \tau$, and, by definition of Γ', we prove R_2.

Moreover, ρ'_t renames more than ρ_t, so choosing a $y \notin \mathcal{V}^{\rho'_t}$ amounts to choosing a $y \notin \mathcal{V}^{\rho_t}$, and H_3 directly implies R_3. We do the same for ρ_e.

— Otherwise, $\rho_t(v) \neq \rho_e(v)$, and $Nxt_h^{\rho_t,\rho_e}(x = v) = (\rho_t[x \mapsto x_t], \rho_e[x \mapsto x_e], (x_t = \rho_t(v), x_e = \rho_e(v))$.

We prove R_0 by choosing two y_t, y_e in $\mathcal{V}_{\rho'_t}, \mathcal{V}_{\rho'_e}$. For each y_i, either $y \in \mathcal{V}_{\rho_i}$, or $y_i = x_i$. If one is a past renaming and the current is new, we have $y_t \neq y_e$. Otherwise, if the two are past renamings, H_0 gives us either $x_t \neq y_e$ or $\rho'_t(y_t) = \rho'_e(y_e)$. Finally, if the two are the two renamings, we have by definition of x_t and x_e, $y_t \neq y_e$. Similarly to the previous case, we show the compatibility of the two new maps by looking at the new renamings: they are fresh variables, not renamed themselves. From the same reasoning, ρ'_t and ρ_e are compatible too.

The typing of H_1 gives us that $\Delta, \Gamma \vdash v : \tau, l$, and $\Gamma' = \Gamma[x \mapsto \tau]$. By Lemma 22, we have $\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(v) :\downarrow_{\{p\}} \tau, l$. By the typing rule of assignment and by H_5,

$$\Delta, \kappa \vdash \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{x_t = \rho_t(v)\}\rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma)[x_t \mapsto \downarrow_{\{p\}} \tau] \cup \Gamma_u$$

And, $x_t = \rho'_t(x)$, thus $\rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma)[x \mapsto \tau] = \rho_t'^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma[x_t \mapsto \tau]) \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)] = \rho_t'^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma') \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)]$. We define $\Gamma'_{u_1} = \Gamma_u \cup \Gamma_{id}[\rho_t(x) \mapsto \Gamma(x)]$, which is harmless, by the same argument as in the previous case. By Lemma 18,

we have

$$\Delta, \kappa \vdash \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{x_t = \rho_t(v)\}\rho_e^p(\rho_t'^p(\Gamma_i, \Gamma'), \Gamma) \cup \Gamma'_{u_1}$$

Following the same logic, we show

$$\Delta, \kappa \vdash \rho_e^p(\rho_t'^p(\Gamma_i, \Gamma'), \Gamma) \cup \Gamma'_{u_1}\{x_e = \rho_e(v)\}\rho_t'^p(\rho_e'^p(\Gamma_i, \Gamma'), \Gamma') \cup \Gamma'_{u_1} \cup \Gamma_{id}[\rho_e(x) \mapsto \Gamma(x)]$$

We prove R_1 with the sequence rule, and R_5 by the same arguments as in the previous case *i.e.*, either $\rho_e(x) = x$ or not, and in the first case, the reminder renaming can be ignored, while in the second, the renaming is not used anywhere. Because ρ'_t (resp. ρ'_e) is simply an update on ρ_t (resp. ρ'_e), we have, for any y, that $\rho'_t(y) = \rho'_e(y)$ if and only if $\rho_t(y) = \rho_e(y)$, because $x_t \neq x_e$. Then H_2 proves R_2. Moreover, ρ'_t renames more than ρ_t, so choosing a $y \notin \mathcal{V}^{\rho'_t}$ amounts to choosing a $y \notin \mathcal{V}^{\rho_t}$, and H_3 directly implies R_3. We do the same for ρ_e to prove R_4.

Case 20 **($P = t[e_1] = e_2$)** Here too, the case is itself divided in two cases. However, in both cases, we have $\rho'_t = \rho_t$, $\rho'_e = \rho_e$, and $\Gamma' = \Gamma$. Thus, the proofs of R_0, R_2, R_3, and R_4 are free.

— If $\rho_t(e_1) = \rho_e(e_1)$ and $\rho_t(e_2) = \rho_e(t_2)$, then $P' = t[\rho_t(e_1)] = \rho_t(e_2)$.
The typing of H_1 gives us that $\Delta, \Gamma \vdash e_1 : \tau_1, r_1$, $\Delta, \Gamma \vdash e_2 : \tau_2, r_2$, and $\tau_1 \sqcup \tau_2 \sqsubseteq \Delta(t)$ (κ being $\mathbf{L}$). By Lemma 22, we have

$$\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(e_1) :\downarrow_{\{p\}} \tau_1, r_1$$

$$\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(e_2) :\downarrow_{\{p\}} \tau_2, r_2$$

We conclude by the typing rule for memory update. Thus, $\Gamma'_u = \Gamma_u$, and that proves R_5.

— Otherwise, $\rho_t(e_1) \neq \rho_t(e_1)$ or $\rho_t(e_2) \neq \rho_e(e_2)$, and $P' = \begin{array}{l} t[\rho_t(e_1)] = h?\rho_t(e_2){:}t[\rho_t(e_1)]; \\ t[\rho_e(e_1)] = !h?\rho_e(e_2){:}t[\rho_e(e_1)]; \end{array}$
Similarly to the previous case, the typing of H_1 gives us that $\Delta, \Gamma \vdash e_1 : \tau_1, r_1$, $\Delta, \Gamma \vdash e_2 : \tau_2, r_2$, $\tau_1 = \mathbf{I}(l)$ and $\tau_1 \sqcup \tau_2 \sqsubseteq \Delta(t)$ (κ being $\mathbf{L}$). By Lemma 22, we have

$$\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(e_1) :\downarrow_{\{p\}} \tau_1, r_1$$

$$\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \vdash \rho_t(e_2) :\downarrow_{\{p\}} \tau_2, r_2$$

Furthermore, the fact that the renamings yield different values either for e_1 or e_2 means that there is a variable $x \in (e_1 \cup e_2)$ such that $\rho_t(x) \neq \rho_t(x)$. By H_2, we have that $p \in \Gamma(x)$. Hence, the typing of H_1 gives us that $\mathbf{I}(\{p\}) \sqsubseteq \Delta(t)$, that is $\Delta(t) = \mathbf{H}$.

By the rule for conditional selection and H_5, we have that

$$\Delta, \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u \vdash h?\rho_t(e_2):t[\rho_t(e_1)] :\downarrow_{\{p\}} \tau, r_2 \cup r_1 \cup (l \setminus \{p\})$$

We conclude by the typing rule for memory update.

Case 21 $(P = s_1; s_2)$ The typing of H_1 gives us that there exists a Γ_1 such that

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma\{s_1\}\Gamma_1$$

$$\Delta, \mathbf{L} \vdash^{@p} \Gamma_1\{s_2\}\Gamma'$$

And we conclude by using twice the induction hypothesis. To show that the Γ'_{u_1} generated is harmless to s_2, we know that all replaced variables are renamings of ρ_t *i.e.*, fresh variables w.r.t the program, and in particular s_2. Thus, retyping those fresh variables is harmless.

Case 22 $(P = \mathbf{if}^{p'}\ h'\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2)$ The typing of H_1 gives us that $\Delta, \Gamma \vdash h' : \tau, r_c$, $\kappa' = \tau \ltimes_{\mathbf{L}} \mathbf{I}(\{p\})$, as well as

$$\Delta, \kappa' \vdash^{@p} \Gamma\{s_1\}\Gamma_1 \qquad \Delta, \kappa' \vdash^{@p} \Gamma\{s_2\}\Gamma_2$$

with Γ_1, Γ_2 such that $\Gamma_1 \sqcup \Gamma_2 = \Gamma'$.

More over, $high(P) = \emptyset$, so $\tau \ltimes_\emptyset \{p\} = \emptyset$, and $\tau = \mathbf{L}$. Therefore, $\kappa' = \mathbf{L}$, and we use the induction hypothesis on s_1 and s_2 along with the harmlessness of Γ_u to get :

$$\Delta, \mathbf{L} \vdash \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{s'_1\}\rho_t^{1p}(\rho_e^{1p}(\Gamma_i, \Gamma_1), \Gamma_1) \cup \Gamma'_{u_1}$$

$$\Delta, \mathbf{L} \vdash \rho_t^p(\rho_e^p(\Gamma_i, \Gamma), \Gamma) \cup \Gamma_u\{s'_2\}\rho_t^{2p}(\rho_e^{2p}(\Gamma_i, \Gamma_2), \Gamma_2) \cup \Gamma'_{u_2}$$

The induction gives us all the required properties on $\rho_{t,e}^{1,2}$, that we pass on to $\rho_{t,e}$ thanks to Lemma 26, thus proving R_0, R_2, R_3, and R_4.

To prove R_1, we use Lemma 25 to ensure that the renaming to Γ_1 and Γ_2 are the same in each environment. This introduces new reminder environments, that we show are harmless per the freshness of the renamings we use. And, by the sequence rule, and the if rule, we conclude.

Case 23 ($P =$ **for** x **from** c_1 **to** c_2 **do** s) To prove R_0, if a variable x is a renaming of ρ'_t and ρ'_e, it is either a renaming of ρ_t and ρ_e, or a fresh variable. In the first case, H_0 is enough. Otherwise, we cannot have the same fresh variable be a renaming of both ρ'_t and ρ'_e, by definition. Thus, ρ'_t and ρ'_e are negligibly interfering. Moreover, a fresh variable cannot be an already renamed variable, and by the same logic as above, ρ'_t and ρ'_e are compatible.

Furthermore, by using the same kind of logic about fresh variables, and noticing that ρ'_t and ρ'_e differ from their initial counterpart on variables modified by s only, we show similarly R_1, R_2, and R_3, because we can choose a type derivation such that Γ' differ from Γ on the modified variables only.

Then, by induction hypothesis, Lemma 25, the sequence rule, and the for rule, we can conclude. □

4.4.5 Merging branches

The last step of the delayed if-conversion is to merge back the two simultaneous executions by using a *post* statement. However, in this *post* statement, the choice between the two maps as to which value to keep is done by using a conditional expression on the condition guard h. Thus, we transform the indirect flow from h originally present to a direct flow. This intuition is clarified in Definition 47 by defining the quasi-classify operator, which removes p from the typing of a variable, and adds that of h instead, if the typing contains p *i.e.*, if the variable is indirectly secret from h in the original program.

Definition 47: Quasi-classify

$$\uparrow^h_{\{p\}} \Gamma(x) = \begin{cases} \Gamma(x) & \text{if } p \notin \Gamma(x) \\ \downarrow_{\{p\}} \Gamma(x) \cup \Gamma(h) & \text{otherwise} \end{cases}$$

Once this quasi-classify operator is defined, we say in Lemma 30 that given a twice renamed typing environment, the post statement yields the quasi-classifying of said environment. Indeed, we only introduce the direct flow from h if the variable is dependent on

p, just as in Definition 47. We prove this by induction over the set of renamed variables, and by verifying that updating using every single assignment of the *post* statement is equivalent to applying the typing operator.

Lemma 30: Merging renaming maps

Let ρ_t and ρ_e be two renaming maps and Γ_i, Γ be two environments such that $\rho_t \equiv^p_\Gamma \rho_e$, $\Gamma_i \equiv_{\rho_t} \Gamma$ and $\Gamma_i \equiv_{\rho_e} \Gamma$, $p \notin \Gamma_i$. Then, for any h such that $\Gamma(h) \neq \mathbf{L}$, the merging of ρ_t and ρ_e defined by :

$$post = seq(\{x = \left\{ \begin{array}{ll} \rho_t(x) & \text{if } \rho_t(x) = \rho_e(x) \\ h?\rho_t(x):\rho_e(x) & \text{otherwise} \end{array} \right. \Big| \, x \in \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}\})$$

is typable, that is

$$\Delta, \kappa \vdash \rho^p_t(\rho^p_e(\Gamma_i, \Gamma), \Gamma)\{post\} \uparrow^h_{\{p\}} \Gamma$$

Proof. By induction over the set $V = \mathcal{V}^{\rho_t} \cup \mathcal{V}^{\rho_e}$.

Case 24 $(V = \emptyset)$ If the renaming maps don't have a single renamed variable, that means that $\rho_t = \rho_e = id$. And, for any x, by well-formation of Γ_i and Γ w.r.t ρ_t, $\Gamma_i(x) = \Gamma(x)$, and $\Gamma_i = \Gamma$. Thus, $\rho^p_t(\rho^p_e(\Gamma_i, \Gamma), \Gamma) = \Gamma$. By definition, we also have $post = \mathbf{skip}$, and

$$\Delta, \kappa \vdash \Gamma\{\mathbf{skip}\}\Gamma$$

Moreover, for any x, by well-formation of Γ w.r.t ρ_t and ρ_e, we have that $x \notin \Gamma(x)$, and $\uparrow^h_{\{p\}} \Gamma = \Gamma$.

Case 25 $(V = V' \cup x)$ We have three cases, either x is a variable renamed by ρ_t or ρ_e, or both.

We first look at the case where x is renamed by both maps. We can have either $\rho_t(x) = \rho_e(x)$ or not. If the equality is true, the induction hypothesis gives us the well-typing for the merging of V', and by well-formation of Γ w.r.t ρ_t, and ρ_e, we know that $p \notin \Gamma(x)$, and we conclude by assignment rule, and Definition 47. However, if the renamings are different, we assign a conditional expression to x, which is then typed to $\Gamma(h) \sqcup \downarrow_{\Gamma(x)} \sqcup \downarrow_{\Gamma(x)} = \uparrow^h_{\{p\}} \Gamma(x)$. We conclude by the induction hypothesis, and the assignment rule.

If only one map renames the variable, the logic is the same, with the difference that we use $\Gamma_i(x)$ instead of $\Gamma(x)$. We conclude by well-formation of Γ_i.

□

4.4.6 Security of the delayed if-conversion

The delayed if-conversion is peculiar in the way that it is trivial for most cases. Only if we look at the p conditional is the transformation complex, using all the steps shown above. In any case, we show in Theorem 16 that if a program P is typable using our strengthened type system on a program point p, we can remove the p conditional, and still type using our main type system. Moreover, by removing this conditional, we effectively decrease the number of conditions (represented by the cardinality of the annotation $high$) by at least one. We prove this by using all the Lemmas proved earlier, and the sequence rule to show that we can chain the yield of each transformation.

Theorem 16: Security of the delayed if-conversion

Let P be a program, p a program point, and Γ a typing environment such that $p \notin \Gamma$, and P is well-typed using Γ *i.e.*,

$$\Delta, \kappa \vdash^{@p} \Gamma\{P\}\Gamma'$$

Then, the transformation P' of P by the delayed if-conversion is also well-typed, and we have $high(P') \subset high(P)$*i.e.*,

$$\Delta, \mathbf{L} \vdash \Gamma\{DIC_p^{V_P}(P')\}\Gamma'' \cup \Gamma_u$$

with $\Gamma'' \sqsubseteq \Gamma'$ and Γ_u a harmless reminder environment.

Proof. By induction over the type derivation. For most cases, this is either immediate or directly from the induction hypothesis. For the sequence, the argument is that because $\Gamma'' \sqsubseteq \Gamma'$, we can retype the second statement using this environment. We can also ignore Γ_u thanks to its harmlessness. For the **for** loop, we ensure that $p \notin \Gamma$ thanks to the classify within the typing rule.

The main case is for the p conditional. We prove that the renaming maps ρ_t and ρ_e follow our constraints using Lemma 19. We also prove that $pre_t; pre_e; Rn_h^{\rho_t}(s_t); Rn_{\neg h}^{\rho_e}(s_e)$ is well-typed using Lemmas 21, 24 and the sequence rule. We conclude by using Lemma 29, as well as Lemma 30 and the sequence rule, using the harmlessness of the reminder environment. Moreover, the quasi-classifying of Γ is laxer than the classifying of Γ. Due

to the removal of the conditional, the *high* annotation can be updated, and we can remove the p program point. □

4.5 Overall transformation and conclusion

Our proposed transformation consists of iterating the array-traversal transformation to ensure that any program types, to then iterate on applying all our subsequent transformations until there is no conditional left. We expressed at the start of this chapter the Theorem 13 which states that given a set of public variables L and an according environment, applying this transformation will yield a constant-time program w.r.t this set L. We prove this by using the Lemmas describing the behavior of a statement transformed by array-traversal to ensure that any program is typing. Then, we use the Theorems proved in this chapter to gradually reduce the number of secret conditionals, up until the point where we can use Theorem 12 to conclude.

Reminder 2: Theorem 13

Given a program P and two simple typing environments Γ and Δ, by defining L the set of public variables in Γ and Δ, then the program resulting of our transformation on P is constant-time w.r.t L, that is :

$$L = \{x | \Gamma(x) \sqcup \Delta(x) = \mathbf{L}\} \implies CT(FCTT_{\Gamma,\Delta}(P), L)$$

Proof Outline. We know by Lemma 10 that for any Γ, Δ, a program without array accesses is typable. Therefore, there is a minimal set of array accesses for which a program is typable (even if this set can be empty). We proved in Lemmas 11, 12 that transforming an array access with the array traversal method yields a typable statement. Hence, by replacing all memory accesses not within said set by their transformation with array traversal, we obtain a typable program $AT_{\Gamma,\Delta}$. Then, while $high(P) \neq \emptyset$, there is a $p \in high(P)$ such that $RO_p(P)$. By chaining Theorems 14, 15 and 16, we are able to show that $T(P) = DIC_p^{V_P}(IS(SI_p(P)))$ types, and that $|high(T(P))| < |high(P)|$. Thus, the cardinality of the annotation being a natural number, by iterating, we end up with a P such that $high(P) = \emptyset$, and we end the transformation. Because $high(P) = \emptyset$, it is easy to show that $leak(P) = \emptyset$, and we conclude by Theorem 12. □

In addition to the proofs in this thesis, the proofs for Theorem 12, Theorem 14 and

Theorem 15 have been mechanized in Coq and can be found here [2]

2. https://github.com/frosqh/ctt_proofs

Chapter 5

Experimentation and evaluation

We have implemented and tested our constant-time enforcement transformation as a pass in the Jasmin compiler [Alm+17; Alm+20].

5.1 Implementation

The Jasmin compiler is written in a mix of Caml and Coq, with the compilation passes in Coq, and oracles and helper functions in Caml. Thus, implementing our transformation amounts to around 4 KLOC in the Gallina language of the Coq proof assistant, along with some util functions in Caml. This development can be found at https://github.com/frosqh/ctt_jasmin.

The first step to implementing our transformation is to update the language to add the **next** construction. This is done by creating a new `instr_n` type, being a pair of `instr_info` (serving as a program point for each instruction), and `instr_r_n` (the *real* instruction). We can then define the conditional as

`Cif_n`: `pexpr` → `seq instr_n` → `seq instr_n` → `seq instr_n` → `instr_r_n`

where `pexpr` is the type for expressions.

5.1.1 Annotations

A crucial part of our transformation that the Jasmin compiler does not handle in its coq formalization is our annotation system. The language allows the programmer to insert annotations on variables, which we use to set the initial type environment for the transformation. Then, by using the `PPSet` type, which is a set of program points (represented using the `positive` type) , we can define the type of an annotation as

```
Record t := mk { leaked: PPSet.t; high : PPSet.t}.
```

Instead of augmenting every instruction with an annotation, which would require to redefine all operations on instruction, we keep track of an *annotation map*, which, for each instruction identified by a program point assigns an annotation.

5.1.2 Typing & Computing Annotations

To ensure the preservation of the constant-time property, JASMIN has a constant-time type system implemented in Caml. Our type system is a generalization of the constant-time one, so we had to implement it from scratch. The intention being to prove the security of the transformation, and our transformation being type-directed, the type system has been implemented in Coq. First, we define a type as

```
Inductive t :=
| I (s:PPSet.t) (* Indirect flow due to the conditionals *)
| H (* High value *)
.
```

On this type, we can define the same operation as in this thesis, such as classify for example :

```
Definition classify (p: positive) (ty: t):=
match ty with
| H ⇒ H
| I s ⇒ if PPSet.mem p s then H else ty
end.
```

Then, a typing environment is simply a map from variables to type. And, we can define a function $type_of_instr$ which implements our typing system, and taking into parameter a fuel `n` notably used for the typing of the **for** loop, a type `m` which is used as the *secret* type, the context `ctx`, the location of the instruction `loc` (for logging purposes), the map of annotations `ga` previously described, a map of typing environment `ge`, the current typing environment `ev`, and finally the instruction `i` :

```
Fixpoint type_of_instr (n: nat) (m: Typing.t) (ctx: Typing.t) (lc: loc) (ga: AnnotMap.t)
(ge : TEnvMap.t) (ev: TEnv.t) (i : instr_n) {struct i} :
    (* ... *).
```

The map of typing environment is a workaround to the flow-sensitivity of our type system. It allows us to access the typing environment after the evaluation of a specific instruction. The main difference between the work in this thesis and the implementation

is the computation of annotations. In Chapter 3, we supposed that the annotations were already computed, and that the role of the type system was to check them. Here, we start with an empty annotations map, and we build it as we type the program. Other than that, the implementation of the type system strictly follows our implementation. For example, the assignment case is simply typing the expression, checking the type of the assignee (in case of a memory access for example), and updating the typing environment to this new type, as well as the annotations map to the union of both leaked traces. All this update is done in the `incl_ge` function.

```
| Cassgn_n lv _ _ e ⇒
    match type_of_pexpr lc ev e with
      | Error e ⇒ Error e
      | Ok (ty, r) ⇒ match (type_of_lval m ctx lc ev lv ty) with
        | Ok (ty', r') ⇒ incl_ge ga ge (ok ty') (Annot.mk (PPSet.join r r') PPSet.empty)
        | Error e ⇒ Error e
        end
    end
```

However, we still use the constant-time type system to check *a posteriori* that our transformation has generated constant-time code.

5.1.3 Scope increase

In the formalization we presented, we suppose it is easy to fetch the p conditional and its continuation from a program. However, in our implementation, this requires a search through the entire program and can be quite taxing. Thus, we use *contexts* to keep track of this unique conditional. We define a context as

```
Inductive t : Type :=
    | Here : instr_info → pexpr → cmd_n → cmd_n → cmd_n → t
    | CElse : instr_info → pexpr → cmd_n → List → cmd_n → t
    | CThen : instr_info → pexpr → List → cmd_n → cmd_n → t
    | CFor : instr_info → var_i → range → List → t
    | CWhile1 : instr_info → align → List → pexpr → cmd_n → t
    | CWhile2 : instr_info → align → cmd_n → pexpr → List → t
    | CSeq : List → t
with
    List := CList : cmd_n → t → cmd_n → List.
```

This context can be seen as a directed search to the p conditional. Any program containing the searched conditional can be represented as a context, and it is easy to fetch any information on the conditional. This mainly works because there is at most one conditional with a non-skip **next** branch at a given time, allowing us to only have cases for `CElse` and `CThen`. Then, we can simply apply our transformations locally to the conditional identified in `Here`, and reconstruct the program afterward.

5.1.4 Renaming and fresh variables

To apply our transformation, we need to be able to generate new names. To this end, we take profit of the implementation of variables in Jasmin. In fact, a variable is represented by two values: an identifier, which is a `positive`, and a set of properties (its name, its type, its *locality* or *globality*). Moreover, when generating code, the Jasmin compiler appends the identifier to the name of any variable. Thus, instead of changing the name of variables, we can simply keep track of the maximum of the identifier used, increment it, and use this as a new identifier. This allows us to create *renamings* that are fresh from anything. Unfortunately, the access to all identifiers is done on Caml, and we do not have a formal guarantee that a fresh identifier is indeed fresh.

5.1.5 Compilation

The bulk of our transformation relies on a single operation: the conditional expression. However, Jasmin is a language for low-level, or cryptographic, applications. And, the language does have a conditional expression operation, which is compiled using the **cmove** instruction, but this operation does not bode well with complex expressions. For example, using a conditional expression within a conditional expression is not handled, and the compilation crashes. One solution would be to first decompose any complex expression we create into simpler ones. This approach would work, if not for the fact that it would require to create lots of temporary variables, used as indexes for memory access. To avoid tackling this, we instead choose to preserve the complex expressions, and stop the compilation before the generation of assembly code. To have an executable result, we use a printer to the C syntax before using a standard C compiler.

5.1.6 Limitations

Because we do not have formal support for function calls, we transform each function at a time, before the aggressive inlining pass of Jasmin. To allow for this, we over-approximate the output typing of a function from $\mathbf{I}(l)$ to $\mathbf{H}$.

5.2 Benchmark

We evaluate our constant-time enforcement transformation on simple but challenging programs that illustrate the expressiveness of our constant-time enforcement transformations. They include the following programs which are taken from the FaCTtest suite

— BranchRemoval : $\mathbf{if}^p\ h\ \mathbf{then}\ x = l_1\ \mathbf{else}\ x = l_2$

— PotentialOOB : $\mathbf{if}^p\ h\ \mathbf{then}\ t[x] = 0\ \mathbf{else\ skip}$

— ReturnDeferral : $\mathbf{if}^p\ h\ \mathbf{then}\ return\ x\ \mathbf{else\ skip}$

We also include the motivating examples taken from the previous examples, such as Program P0 and P12, as well as bubble sort. We remind below both Programs P0 and P12.

Reminder: Program P0

```
if@p h then x=l1; y=t[x] else x=l2; y=t[x]
```

Reminder: Program P12

```
if@p h then x = l1 else x = l2;
t[l3] = l4;
y = t[x];
```

Finally, we hand-craft programs to evaluate the behavior of the transformations on particular patterns. Firstly, we define Program P32 to be the `cswap` function used by the existing implementation of Curve25519 in Jasminfrom *libjc*[1] [Alm+20]. More precisely, we have rewritten the existing constant-time `cswap` to the more natural non constant-time version.

1. https://github.com/tfaoliveira/libjc

Program P32: Swap function from Curve25519

```
if@p swap then
for i from 0 to 4 do
    tmp = z2p[i]; z2p[i] = z3p[i];; z3p[i] = tmp;
    tmp = x2p[i]; x2p[i] = x3p[i];; x3p[i] = tmp;
else skip
```

To test the extent of the scope-increase transformation, we create Programs P33, P34, 35 and P36 that requires code motion to be transformed.

Program P33: Conditional access

```
(if@p h then x = l1 else x = l2;) y = t[x];
```

Program P34: Two nested conditionals

```
if@p h then
    if@p' h' then
        x = l1
    else x = l2
    y = t[x];
else y = l3
t[y] = l4;
```

Program P35: Two sequential conditionals

```
if@p h then r=0 else r=1;
if@p' r then x=1 else x=2;
y = t[x];
```

Program P36: For loop containing a memory access

```
for i from c1 to c2 do
    if@p h then x = l1 else x = l2;
    y = t[x];
```

We also experiment on the use of the naive array traversal transformation by using Program 37 in which the memory access is outside the bounds of a **for** loop.

Program P37: For loop followed by a memory access

```
for i from c1 to c2 do
    if@p h then x = l1 else x = l2;
y = t[x];
```

Unfortunately, programs like AES would be pointless to transform with our transformation, given that only the array traversal would apply. Thus, we restrict ourselves to experiment on *meaningful* programs.

Metrics

We evaluate each program, and its transformation, on the following set of metrics. These metrics are computed directly within the Jasmin compiler.

a) Constant-Time: We check whether the program is successfully transformed and if the transformed version is indeed Constant-Time according to the type checker of Jasmin.

b) Code Size Overhead: We provide the size of the initial code and its size after the transformation in terms of number of statements.

c) Number of Variables: We provide the number of variables used by the program, at source level, and after transformation but before optimizations.

d) Compilation Time: We provide the time taken by Jasmin to complete the compilation of the program, whether the transformation is enabled or not, in seconds.

Program	Constant-Time		Source code size		Var - Source		Var - Compiled	
	FaCT	Ours	Input	Output	Input	Output	C[Input]	C[Output]
BranchRemoval	✓	✓	3	8	1	3	6	6
PotentialOOB	∼	✓	3	5	1	2	6	6
ReturnDeferral	✓	×	–	–	–	–	–	–
`cswap` (P32)	✓	✓	27	45	10	21	6	7
BubbleSort	✓	✓	8	12	4	6	7	8
P0	×	✓	9	19	2	7	6	7
P12	×	✓	8	17	3	7	6	7
P33	×	✓	7	16	2	7	6	7
P34	×	✓	11	46	3	18	6	12
P35	×	✓	10	24	3	10	6	9
P36	×	✓	8	17	3	8	6	8
P37	×	✓	8	14	3	7	6	7

Table 5.1 – Case-study of our transformation - Limited to Jasmin

e) Assembly Size: We also keep track of the size of the compiled code, before and after transformation. Due to restrictions on the Jasmin compiler, our introduction of complex expressions sometimes prevents the compilation from terminating. To evaluate our compiled code, we export the resulting high-level program to C, and compile it using CompCert, thus preserving the Constant-Time property. As to compensate CompCert's lack of optimization, we also try compiling using GCC -O3.

5.3 Results and evaluation

The results of our evaluation are summarized in Tables 5.1 and 5.2.

5.3.1 Evaluation of Results

Constant-Time Property We are able to transform all the benchmarks except ReturnDeferral which is rejected because Jasmin only accept a *return* as the last instruction. For PotentialOOB, our generated program is different from FaCT which inserts an *assume* statement to ensure safety. Instead, we instrument the array access and get safety for free. Yet, our transformation is only semantically correct if the initial program has no array of bound access. Programs P0, P12, P33, P34, P35, P36 and P37 are rejected by FaCT but accepted by our enhanced transformation at the cost of some code duplication.

Code Size Overhead For most of the benchmarks, the resulting source code is around twice the size of the original. This observation is true for programs containing at most *1* nested conditional : the code duplication pass is only applied once.

For other programs, such as P34, the overhead is proportional to 2^n, with n the *depth* of the program. In the case of P34, the $t[y] = l_4$ instruction is duplicated by the first pass of delayed if-conversion, *inserted* into the **next** for the p conditional, and later duplicated again. This repeats at every level of nesting in the program.

Number of Variables - Source By the same reasoning as above, the number of variables in the transformed code, before compilation, should be around 2^n times the number of variables in the original program, with n the depth of the program. This is indeed what can be observed in Table 5.1, where the ratio $Var[Output]/Var[Input]$ is mostly between 2^{n-1} and 2^n.

Number of Variables - Compiled The Jasmin compiler applies a number of aggressive optimizations. To evaluate the impact of optimizations on the transformed code, we also compare the number of written variables with and without constant-time enforcement.

For most of the programs, the variable overhead is reduced to 1 or 2 and is almost insignificant. However, for programs such as P34, where there is more than one level, the overhead is around n times the initial source code, with n the depth of the program. The exponential 2^n presented above is gone thanks to the removal of redundant renamings. The transformation creates a renaming for each nested conditional, for each variable, but in practice, because y is not used in the p' conditional and x is not used in the p' conditional, these renamings can be merged.

Compilation Time For most of the programs, we have at most one order of magnitude of compilation time added by the transformation. However, when a secret conditional is within a loop, our variable overhead is multiplied by the loop unrolling of Jasmin, resulting in greater compilation time, although it stays reasonable.

Assembly Size For most of our benchmarks, the resulting assembly code using CompCert does not differ by much in size. Notable exceptions are `cswap` and P34. The one common factor between these two code snippets is the introduction of multiple **cmove** instructions. The construction of the expressions used within those instructions provokes

Program	Constant-Time		Compilation Time		Asm size (CompCert)		Asm Size (gcc -O3)	
	FaCT	Ours	Classic	Transformed	Classic	Transformed	Classic	Transformed
BranchRemoval	✓	✓	0.003	0.016	19	19	24	24
PotentialOOB	∼	✓	0.004	0.011	22	25	26	34
ReturnDeferral	✓	×	–	–	–	–	–	–
`cswap` (P32)	✓	✓	0.007	0.212	61	136	75	82
BubbleSort	✓	✓	0.268	2.126	42	52	53	56
P0	×	✓	0.004	0.028	27	43	31	31
P12	×	✓	0.004	0.036	24	27	27	28
P33	×	✓	0.006	0.023	21	24	25	26
P34	×	✓	0.004	0.093	28	94	31	67
P35	×	✓	0.005	0.025	27	24	26	29
P36	×	✓	0.007	0.034	28	25	26	24
P37	×	✓	0.005	0.056	27	37	25	45

where Asm stands for Assembly Code.

Table 5.2 – Case-study of our transformation - Compilation time and Exported to C

a significant overhead in code size. When compiling using all optimizations offered by GCC, we don't notice such a high overhead anymore, except for P34. Overall, both our transformations, whether the compilation method used, struggle with nested conditionals but offer satisfying results on other programs.

Summary of Evaluation Our type-directed transformation allows for more programs to be transformed into a Constant-Time semantically-equivalent version than FaCT. This transformation implies a performance and size overhead at most doubles in performance and size. Subsequent compiler optimizing passes remove most of the increase in code size and number of variables used. However, per the logic of our transformation, the overhead is worse for nested conditionals that are transformed and duplicated one at a time.

Conclusion

Side-channel attacks are a credible threat, and cryptography has to take it into account. To provide stronger guarantees, formal methods are often used on cryptographic primitives. However, most of the literature on the subject is centered on the preservation or the verification of a given property. This thesis proposes a way to enforce the constant-time property, to alleviate the burden on the programmer.

Summary

In Chapter 1, we formally defined the basics of the language on which we base our transformation. We also presented a more formal definition for the constant-time policy, as a non-interference policy and an information flow type system to ensure this non-interference policy. We have given an overview of the simplest solution to ensure the constant-time property: the use of a **ctselect**, the branch removal transformation, and the array traversal one. We talked about two limitations of the usual branch removal: the issue of safety and indirect leakage.

In Chapter 2, we gave an informal overview of our transformation, as well as a formal definition of each pass. This has been done, for each pass, by building on top of the previous one to alleviate its limitations. These passes are, in order of application in the final transformation, scope-increase, index sanitization, delayed if-conversion, and array-traversal. The scope-increase transformation relies on code motion to ensure that all side effects from a conditional are contained within its scope. To do so, we introduce a new construction **next**, which allows us to store the continuation of a conditional. The index sanitization pass introduces dynamic bound checks to ensure that any memory access is safe, even if we do not have any guarantee on the value of the index. Then, delayed if-conversion uses renamings to safely remove a conditional by delaying the introduction of direct flow. Finally, array traversal uses the classical transformation presented in Chapter 1 to take care of the remaining secret memory access. We also defined in this chapter the information needed for all these transformations to succeed, such as leaking annotations. For each one of those *sub*-transformations, as well as for the final, complete,

one, we presented an outline of the intuition to prove the semantics preservation.

In Chapter 3, we presented our information flow type-system used to fulfill the needs raised in the previous chapter. The type system uses an annotation system to keep track of the leaks and high conditionals within a statement and rejects programs performing memory accesses on secret indexes. We proved that without any annotations, our type system is equivalent to the constant-time one. Thus, if we succeed in transforming a program such as it is typable with empty annotations, the transformed program would be constant-time. Moreover, to differentiate the indirect flows from the direct flows we augment the usual type lattice with a new type $\mathbf{I}(l)$ which represents a variable dependent on the program points in l. Thus being a public, or low, variable means that we are not dependent on anything, and this variable can be typed as $\mathbf{I}(\emptyset)$. By defining this type system, we also updated the transformation, moving the use of array-traversal at the start, to ensure that any program is typable. And, thanks to our computational definition of the type system, the burden of annotations is not on the programmer, the type system is able to compute them as it types.

In Chapter 4, we proved that for any program, its transformation abides by the constant-time policy. To do so, we first proved that applying array-traversal always makes the program *more typable*, and that, by applying it enough times, any program becomes typable with our type system. Then, we proved that by identifying a problematic condition identified as p, moving it using scope-increase, securing it using index sanitization, and removing it using delayed if-conversion, we could remove p from all annotations within the program, and still types. Thus, by iterating this method enough times, we can type with only empty annotations, and thus have a constant-time program.

In Chapter 5, we implemented our transformation into the Jasmin compiler. This amounts to 4K lines of code in the Gallina language of Coq. With this implementation, we performed a benchmark on some hand-crafted examples. This confirmed that when we are able to transform, the result is constant-time; that we are able to transform more than the literature (FaCT for example), and that the overhead is great in number of variables, but that compiler's optimizations are able to greatly decrease it.

Perspectives

We now suggest some perspective for future work. These perspectives can be categorized into three groups: 1) the language *i.e.*, handling more complex constructions,

and more subtlety, 2) the implementation *i.e.*, improving the benchmarks and/or the implementation and, 3) the scope *i.e.*, going beyond just constant-time.

Increasing the scope of the language

Our current work is based on a simple language. If we want to generalize our transformation, the obvious way to do it is to extend the language.

Non constant-time operations As stated in Chapter 1, all operations are not constant-time. In particular, the integer division is implemented differently whether the divisor or the dividend is small. Thus, the trace leak won't be the same depending on the values of these two parameters. However, by extending our language to those operations, and typing them as leaking operations (just as the array read), we should be able to either prevent them from being called on secret or transform them away, perhaps with a method similar to array traversal. In practice, this would need us to separate operations into which are constant-time, and can be used indifferently, and which are not constant-time, and need to be used only on non-secrets inputs. Such a change requires an update both to the syntax (to differentiate $\oplus_{CT}$ to $\oplus_{NCT}$), the semantics, and the type system.

Unsafe operations Similarly, not all operations are equally safe. For example, the division cannot be called with 0 as the divisor. If we wanted to extend the language to handle properly such operations, we would need to generalize the index sanitization pass. Instead of working only on memory access, the transformation would sanitize all non-safe operations. For division, the rule could for example be $x/y \mapsto x/(y \neq 0?y{:}1)$. This would only need us to categorize operations as safe or unsafe, such that the index sanitization transformation is able to recognize unsafe statements.

Non constant loops Our current language is restricted to use **for** loops with constant bounds. Even if these kinds of loops are the most frequent in cryptographic code, some primitives make use of the **while** loop, or of non-constant bounds. In the case where the bounds of these loops are public, our transformation which prevents code motion inside loops is enough: we transform them *on the side*. However, if those bounds are secret (either because of indirect or direct flows), we have to transform those. Indeed, a **while** loop using a secret condition as a guard is definitely not constant-time: it will have a number of iterations directly linked to its secret guard. To fix this issue, one solution

would be to couple our transformation with static analysis as to determine a static upper bound to the number of iterations. Thus, we could always transform a **while** loop into a constant **for** loop using this upper bound, and predicate the body of the loop with h, where h is the guard of the **while** loop. This way, the semantics is preserved, and the leak from the number of iteration is removed. Moreover, running a **while** loop in a high security context could result in a non-terminating program. Indeed, executing the loop in a wrong branch could remove the guarantee that the loop terminates. One solution would be to use a pass similar to index sanitization to, as for high **while** loops, use the upper bounds of the number of iterations to ensure that the loop never goes beyond that point. Applying these two transformations would make it so that only public **while** loops within a public context would be left unchanged.

Functions Similarly, the way we handle functions in this thesis is by relying on external inlining passes. However, multiple works, such as FaCT[Cau+17] have succeeded in taking into account function calls during their transformation. One way we could implement it would be to add a *function context* to the parameters of any function. This context would keep track of whether or not we are within the scope of a secret conditional when called and would allow to spread the indirect typing of this conditional to all variables modified within the function. Then, we could imagine transforming any function into two versions: a secret and a public one. The call of the function within the scope of a secret conditional would be transformed to call the secret version with the correct function context, while any other call would use the public version.

Extending the implementation

Our current work is implemented as a pass within Jasmin and tested on a small benchmark composed of hand-crafted languages.

Benchmarks Unfortunately, this benchmark severely lacks real world examples. Most cryptographic algorithms are directly written as constant-time. Still, trying our transformation on cryptographic libraries, such as NaCl, could prove useful to provide insight into the usefulness and overhead of our transformation. Similarly, the current benchmark is limited to evaluating the size of the compiled code. One perspective for future work would be to compare the performance of execution between the original implementation and our transformed version.

Mechanized proofs Even though proofs of the transformation are proposed in Chapter 4 and a subsection of proofs are mechanized using the Coq assistant, the implementation within the Jasmin compiler is unproven. Thus, as a future work, proving the transformation within the context of an already established verified compiler could prove to be quite an interesting challenge.

Open challenges: speculative execution & power analysis

In this thesis, our work has been focused on alleviating timing vulnerability in non-speculative, in-order execution. However, such transformation would also be useful in the context of speculative execution, or to alleviate a different kind of side-channel.

Speculative Execution New attacks abusing the speculative execution of processors have emerged in the last years, such as Spectre [Koc+20]. These attacks try to access data wrongly stored in the cache after a mispeculation. To counter these attacks, a speculative constant-time policy has been proposed [Cau+20]. A future possible work would be to try and adapt our transformation to ensure this speculative constant-time policy instead of the constant-time one. Promising approaches would be to perform speculative post-analysis à la blade [Vas+21] over the constant-time program, or use *protect* program annotations, as Jasmin [Bar+21], on top of our transformation.

Power Analysis If the speculative execution has been the subject of intensive work for the past few years leading to the definition of speculative constant-time, it is not the case for other side-channels. For example, power analysis attacks are often mitigated using hardware solutions, such as complimentary gate [GG18]. Thus, an interesting future work could be the definition of a *constant-power* policy, along with corresponding semantics and type system. These constructions would allow us to see if a type-directed program transformation approach as the one of this thesis would be a good fit to tackle other side-channel attacks.

Bibliography

[Aga00] Johan Agat, « Transforming Out Timing Leaks », *in*: *POPL*, ed. by Mark N. Wegman and Thomas W. Reps, ACM, 2000, pp. 40–53, DOI: 10.1145/325694.325702, URL: https://doi.org/10.1145/325694.325702.

[Alm+16] José Bacelar Almeida et al., « Verifying Constant-Time Implementations », *in*: *25th USENIX Security Symposium*, ed. by Thorsten Holz and Stefan Savage, USENIX Association, 2016, pp. 53–70, URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

[Alm+17] José Bacelar Almeida et al., « Jasmin: High-Assurance and High-Speed Cryptography », *in*: *CCS*, ed. by Bhavani M. Thuraisingham et al., ACM, 2017, pp. 1807–1823, DOI: 10.1145/3133956.3134078, URL: https://doi.org/10.1145/3133956.3134078.

[Alm+20] José Bacelar Almeida et al., « The Last Mile: High-Assurance and High-Speed Cryptographic Implementations », *in*: *S&P*, IEEE, 2020, pp. 965–982, DOI: 10.1109/SP40000.2020.00028, URL: https://doi.org/10.1109/SP40000.2020.00028.

[Amm+22] Basavesh Ammanaghatta Shivakumar et al., « Enforcing Fine-grained Constant-time Policies », *in*: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 83–96, ISBN: 978-1-4503-9450-5, DOI: 10.1145/3548606.3560689, URL: https://dl.acm.org/doi/10.1145/3548606.3560689 (visited on 10/13/2023).

[Bar+13] Gilles Barthe et al., « EasyCrypt: A Tutorial », *in*: *FOSAD Tutorial Lectures*, ed. by Alessandro Aldini, Javier López, and Fabio Martinelli, vol. 8604, LNCS, Springer, 2013, pp. 146–166, DOI: 10.1007/978-3-319-10082-1_6, URL: https://doi.org/10.1007/978-3-319-10082-1%5C_6.

[Bar+14] Gilles Barthe et al., « System-level Non-interference for Constant-time Cryptography », *in*: *CCS*, ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li, ACM, 2014, pp. 1267–1279, DOI: 10.1145/2660267.2660283, URL: https://doi.org/10.1145/2660267.2660283.

[Bar+19] Gilles Barthe et al., « System-Level Non-interference of Constant-Time Cryptography. Part I: Model », *in*: *J. Autom. Reason.* 63.*1* (2019), pp. 1–51, DOI: 10.1007/s10817-017-9441-5, URL: https://doi.org/10.1007/s10817-017-9441-5.

[Bar+20a] Gilles Barthe et al., « Formal verification of a constant-time preserving C compiler », *in*: *Proc. ACM Program. Lang.* 4.*POPL* (2020), 7:1–7:30, DOI: 10.1145/3371075, URL: https://doi.org/10.1145/3371075.

[Bar+20b] Gilles Barthe et al., « System-Level Non-interference of Constant-Time Cryptography. Part II: Verified Static Analysis and Stealth Memory », *in*: *J. Autom. Reason.* 64.*8* (2020), pp. 1685–1729, DOI: 10.1007/s10817-020-09548-x, URL: https://doi.org/10.1007/s10817-020-09548-x.

[Bar+21] Gilles Barthe et al., « High-Assurance Cryptography in the Spectre Era », *in*: *2021 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, May 2021, pp. 1884–1901, DOI: 10.1109/SP40001.2021.00046, URL: https://ieeexplore.ieee.org/document/9519434 (visited on 10/13/2023).

[BB05] David Brumley and Dan Boneh, « Remote timing attacks are practical », en, *in*: *Computer Networks* 48.*5* (Aug. 2005), pp. 701–716, ISSN: 13891286, DOI: 10.1016/j.comnet.2005.01.010, URL: https://linkinghub.elsevier.com/retrieve/pii/S1389128605000125 (visited on 10/13/2023).

[Ben01] Mordechai Ben-Ari, « The bug that destroyed a rocket », *in*: *SIGCSE Bulletin* 33 (June 2001), pp. 58–59, DOI: 10.1145/571922.571958.

[Ber05] D. Bernstein, « Cache-timing attacks on AES », *in*: 2005, URL: https://www.semanticscholar.org/paper/Cache-timing-attacks-on-AES-Bernstein/352e74019d86163d73618f03429ae452ab429629 (visited on 10/13/2023).

[BJR23] Frédéric Besson, Thomas Jensen, and Gautier Raimondi, « Type-directed Program Transformation for Constant-Time Enforcement », *in*: *International Symposium on Principles and Practice of Declarative Programming*

(PPDP 2023), 2023, DOI: 10.1145/3610612.3610618, URL: https://doi.org/10.1145/3610612.3610618.

[Bon+17] Barry Bond et al., « Vale: Verifying High-Performance Cryptographic Assembly Code », *in*: *USENIX Security*, ed. by Engin Kirda and Thomas Ristenpart, USENIX Association, 2017, pp. 917–934, URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[BPT19] Sandrine Blazy, David Pichardie, and Alix Trieu, « Verifying constant-time implementations by abstract interpretation », *in*: *J. Comput. Secur.* 27.*1* (2019), pp. 137–163, DOI: 10.3233/JCS-181136, URL: https://doi.org/10.3233/JCS-181136.

[Cau+17] Sunjay Cauligi et al., « FaCT: A Flexible, Constant-Time Programming Language », *in*: *IEEE Cybersecurity Development, SecDev 2017*, IEEE Computer Society, 2017, pp. 69–76, DOI: 10.1109/SecDev.2017.24, URL: https://doi.org/10.1109/SecDev.2017.24.

[Cau+19] Sunjay Cauligi et al., « FaCT: a DSL for timing-sensitive computation », *in*: *PLDI*, ed. by Kathryn S. McKinley and Kathleen Fisher, ACM, 2019, pp. 174–189, DOI: 10.1145/3314221.3314605, URL: https://doi.org/10.1145/3314221.3314605.

[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*, PLDI 2020, New York, NY, USA: Association for Computing Machinery, 2020, pp. 913–926, ISBN: 978-1-4503-7613-6, DOI: 10.1145/3385412.3385970, URL: https://dl.acm.org/doi/10.1145/3385412.3385970 (visited on 10/16/2023).

[Dwo23] Morris J Dworkin, *Advanced Encryption Standard (AES)*, en, tech. rep. NIST FIPS 197-upd1, Gaithersburg, MD: National Institute of Standards and Technology, 2023, NIST FIPS 197–upd1, DOI: 10.6028/NIST.FIPS.197-upd1, URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf (visited on 10/15/2023).

[GG18] Edouard Giacomin and Pierre-Emmanuel Gaillardon, « Differential Power Analysis Mitigation Technique Using Three-Independent-Gate Field Effect Transistors », *in*: *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), ISSN: 2324-8440, Oct. 2018, pp. 107–112, DOI: 10.1109/VLSI-SoC.2018.8644747, URL: https://ieeexplore.ieee.org/document/8644747 (visited on 10/16/2023).

[Hal+08] Daniel Halperin et al., « Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses », *in*: *2008 IEEE Symposium on Security and Privacy (sp 2008)*, ISSN: 2375-1207, May 2008, pp. 129–142, DOI: 10.1109/SP.2008.31, URL: https://ieeexplore.ieee.org/document/4531149 (visited on 10/13/2023).

[HS06] Sebastian Hunt and David Sands, « On flow-sensitive security types », *in*: *POPL*, ACM, 2006, pp. 79–90, DOI: 10.1145/1111037.1111045.

[KM07] Boris Köpf and Heiko Mantel, « Transformational typing and unification for automatically correcting insecure programs », *in*: *Int. J. Inf. Sec.* 6.*2-3* (2007), pp. 107–131.

[Koc+11] Paul Kocher et al., « Introduction to differential power analysis », en, *in*: *Journal of Cryptographic Engineering* 1.*1* (Apr. 2011), pp. 5–27, ISSN: 2190-8516, DOI: 10.1007/s13389-011-0006-y, URL: https://doi.org/10.1007/s13389-011-0006-y (visited on 10/13/2023).

[Koc+20] Paul Kocher et al., « Spectre attacks: exploiting speculative execution », *in*: *Commun. ACM* 63.*7* (2020), pp. 93–101, DOI: 10.1145/3399742, URL: https://doi.org/10.1145/3399742.

[Koc96] Paul C. Kocher, « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems », *in*: *16th Annual International Cryptology Conference*, ed. by Neal Koblitz, vol. 1109, LNCS, Springer, 1996, pp. 104–113, DOI: 10.1007/3-540-68697-5_9, URL: https://doi.org/10.1007/3-540-68697-5%5C_9.

[Kum+14] Ramana Kumar et al., « CakeML: a verified implementation of ML », en, *in*: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego California USA: ACM, Jan. 2014,

pp. 179–191, ISBN: 978-1-4503-2544-8, DOI: 10.1145/2535838.2535841, URL: https://dl.acm.org/doi/10.1145/2535838.2535841 (visited on 10/13/2023).

[Lei10] K. Rustan M. Leino, « Dafny: An Automatic Program Verifier for Functional Correctness », *in*: *LPAR*, ed. by Edmund M. Clarke and Andrei Voronkov, vol. 6355, LNCS, Springer, 2010, pp. 348–370, DOI: 10.1007/978-3-642-17511-4_20, URL: https://doi.org/10.1007/978-3-642-17511-4%5C_20.

[Ler09] Xavier Leroy, « A Formally Verified Compiler Back-end », en, *in*: *Journal of Automated Reasoning* 43.*4* (Dec. 2009), pp. 363–446, ISSN: 0168-7433, 1573-0670, DOI: 10.1007/s10817-009-9155-4, URL: http://link.springer.com/10.1007/s10817-009-9155-4 (visited on 10/13/2023).

[Lev95] Nancy G. Leveson, *Safeware: system safety and computers*, New York, NY, USA: Association for Computing Machinery, Mar. 1995, ISBN: 978-0-201-11972-5.

[Mus+20] Maria Mushtaq et al., « Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA », *in*: *Information Systems* 92 (Sept. 1, 2020), p. 101524, ISSN: 0306-4379, DOI: 10.1016/j.is.2020.101524, URL: https://www.sciencedirect.com/science/article/pii/S0306437920300338 (visited on 10/13/2023).

[Pro+17] Jonathan Protzenko et al., « Verified low-level programming embedded in F », *in*: *Proc. ACM Program. Lang.* 1.*ICFP* (2017), 17:1–17:29, DOI: 10.1145/3110261, URL: https://doi.org/10.1145/3110261.

[RBV16] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede, *Dude, is my code constant time?*, Publication info: Published elsewhere. Minor revision. DATE 2017, 2016, URL: https://eprint.iacr.org/2016/1123 (visited on 10/13/2023).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, « A method for obtaining digital signatures and public-key cryptosystems », *in*: *Communications of the ACM* 21.*2* (1978), pp. 120–126, ISSN: 0001-0782, DOI: 10.1145/359340.359342, URL: https://dl.acm.org/doi/10.1145/359340.359342 (visited on 10/13/2023).

[SP21] Luigi Soares and Fernando Magno Quintão Pereira, « Memory-Safe Elimination of Side Channels », *in*: *IEEE/ACM CGO*, ed. by Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, IEEE, 2021, pp. 200–210, DOI: 10.1109/CGO51591.2021.9370305, URL: https://doi.org/10.1109/CGO51591.2021.9370305.

[Sun+16] Chengnian Sun et al., « Toward understanding compiler bugs in GCC and LLVM », *in*: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, New York, NY, USA: Association for Computing Machinery, 2016, pp. 294–305, ISBN: 978-1-4503-4390-9, DOI: 10.1145/2931037.2931074, URL: https://dl.acm.org/doi/10.1145/2931037.2931074 (visited on 10/13/2023).

[Swa+16] Nikhil Swamy et al., « Dependent types and multi-monadic effects in F », *in*: *POPL*, ed. by Rastislav Bodík and Rupak Majumdar, ACM, 2016, pp. 256–270, DOI: 10.1145/2837614.2837655, URL: https://doi.org/10.1145/2837614.2837655.

[Tho15] Ian Thomson, *That EVIL TEXT that will CRASH your iPhone: We pop the hood*, en, May 2015, URL: https://www.theregister.com/2015/05/27/text_message_unicode_ios_osx_vulnerability/ (visited on 10/13/2023).

[Vas+21] Marco Vassena et al., « Automatically eliminating speculative leaks from cryptographic code with blade », *in*: *Proc. ACM Program. Lang.* 5.*POPL* (2021), pp. 1–30, DOI: 10.1145/3434330, URL: https://doi.org/10.1145/3434330.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith, « A Sound Type System for Secure Flow Analysis », *in*: *J. Comput. Secur.* 4.*2/3* (1996), pp. 167–188, DOI: 10.3233/JCS-1996-42-304, URL: https://doi.org/10.3233/JCS-1996-42-304.

[Wu+18] Meng Wu et al., « Eliminating timing side-channel leaks using program repair », *in*: *ISSTA*, ed. by Frank Tip and Eric Bodden, ACM, 2018, pp. 15–26, DOI: 10.1145/3213846.3213851, URL: https://doi.org/10.1145/3213846.3213851.

[Zin+17] Jean Karim Zinzindohoué et al., « HACL*: A Verified Modern Cryptographic Library », *in*: *CCS*, 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.

[ZS18] Mark Zhao and G. Edward Suh, « FPGA-Based Remote Power Side-Channel Attacks », *in*: *2018 IEEE Symposium on Security and Privacy (SP)*, ISSN: 2375-1207, May 2018, pp. 229–244, DOI: 10.1109/SP.2018.00049, URL: https://ieeexplore.ieee.org/document/8418606 (visited on 10/13/2023).

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

Université de Rennes

Titre : Compilation Sécurisée contre les Attaques par Canaux Auxiliaires

Mot clés : Compilation Sécurisée, Constant-Time, Vérification Formelle, Transformation de Programme

Résumé : De par leur omniprésence, la sécurité des systèmes informatiques est un enjeu majeur. Dans cette thèse, nous visons à garantir une sécurité contre un certain type d'attaques : les attaques par canal caché temporel. Ces attaques utilisent le temps d'exécution d'un programme pour déduire des informations sur le système. En particulier, on dit d'un programme qu'il est constant-time lorsqu'il n'est pas sensible à ce type d'attaques. Cela passe par des contraintes sur le programme, qui ne doit ni réaliser de décisions en utilisant de valeurs secrètes, ni utiliser un de ces secrets pour accéder à la mémoire. Nous présentons dans ce document une méthode permettant de garantir la propriété constant-time d'un programme. Cette méthode est une transformation à haut niveau, suivi d'une compilation par Jasmin pour préserver la propriété. Nous présentons également la preuve de la sécurité et de la préservation sémantique de cette méthode.

Title: Secure Compilation against Side-Channel Attacks

Keywords: Secure Compilation, Constant-Time, Formal Verification, Program Transformation

Abstract: Given their ubiquity, the security of computer systems is a major issue. In this thesis, we aim to guarantee security against a certain type of attack: timing side-channel attacks. These attacks use the execution time of a program to deduce information about the system. In particular, a program is said to be constant-time when it is not sensitive to this type of attack. This requires constraints on the program, which must neither make decisions using secret values, nor use one of these secrets to access memory. In this document, we present a method for guaranteeing the constant-time property of a program. This method is a high-level transformation, followed by compilation using Jasmin to preserve the property. We also present a proof of the security and semantic preservation of this method.