Verifying constant-time implementations in a verified compilation toolchain

Alix Trieu

To cite this version:

Alix Trieu. Verifying constant-time implementations in a verified compilation toolchain. Cryptography and Security [cs.CR]. Université Rennes 1, 2018. English. NNT : 2018REN1S099 . tel-01944510v3

HAL Id: tel-01944510

https://hal.inria.fr/tel-01944510v3

Submitted on 17 Jun 2019

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT UNIVERSITÉ DE RENNES 1

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

École Doctorale N°601
Mathématique et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Informatique

Alix Trieu

Vérification d'implémentations constant-time dans une chaîne de compilation vérifiée

Thèse présentée et soutenue à Rennes, le 4 Décembre 2018
Unité de recherche : Unité Mixte de Recherche 6074 – IRISA
Thèse N° :

Rapporteurs avant soutenance :

M. Frank PIESSENS – Professeur – Katholieke Universiteit Leuven
Mme. Marie-Laure POTET – Professeur des universités – ENSIMAG

Composition du jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition ne comprend que les membres présents

Président :	Prénom Nom Fonction et établissement d'exercice
Examinateurs :	Mme. Stéphanie DELAUNE – Directrice de recherche – CNRS, IRISA
	M. Frank PIESSENS – Professeur – Katholieke Universiteit Leuven
	Mme. Marie-Laure POTET – Professeur des universités – ENSIMAG
	M. Alejandro RUSSO – Professeur – Chalmers University of Technology
Dir. de thèse :	Mme. Sandrine BLAZY – Professeur des universités – Université de Rennes 1
Co-dir. de thèse :	M. David PICHARDIE – Professeur des universités – ENS Rennes

Contents

List of Figures

RÉSUMÉ ÉTENDU EN FRANÇAIS

Depuis des temps immemoriaux, les hommes ont essayés de communiquer de manière privée en public, c'est-à-dire que seuls les participants pouvaient comprendre les communications mêmes en présence d'espions. Un des premiers exemples est celui de l'empereur Jules César qui encodait ses correspondances militaires et privées en utilisant le chiffrement de César. Le principe de cette technique de chiffrement était de substituer les lettres du texte original par une lettre décalée dans l'alphabet de la première par un nombre fixé. Le texte encodé de "bonjour" serait alors "erqmrxu" avec un décalage de 3, `b` est devenu `e`, `o` est devenu `r`, etc.

La cryptographie est la discipline de la sécurisation des communications en présence d'attaquants. Les primitives cryptographiques sont des algorithmes basiques qui fournissent des services cryptographiques tels que coder ou décoder un message. Ces algorithmes sont généralement prouvés par leur créateurs comme étant *calculatoirement* sécurisés, c'est-à-dire qu'il faudrait un temps ridiculeusement long à un attaquant afin d'outrepasser les sécurités de l'algorithme en utilisant une immense puissance de calcul. Les protocoles cryptographiques sont construits en utilisant ces primitives comme briques de base, ils spécifient comment utiliser ces primitives afin de communiquer de manière sûre.

Les communications électroniques devenant de plus en plus importantes dans notre monde par la démocratisation d'Internet, des cartes bancaires, du paiement sans contact, des smartphones, etc. La cryptographie est devenue une pièce centrale dans la protection de notre vie privée. Ainsi, des erreurs dans la création ou l'implémentation de primitives cryptographiques ou de protocoles pourraient avoir des conséquences désastreuses, que ce soit en terme d'argent ou de vies. Par exemple, les applications mobiles pour communiquer en privé sont devenus répandues dans les pays où la liberté d'expression n'est pas garantie. Si ces communications étaient révélées, cela pourrait mettre leurs auteurs en grand danger.

Un autre exemple est celui du bug informatique *Heartbleed*. En avril 2014, un bug de sécurité a été découvert dans la librairie cryptographique OpenSSL[1], une des, si ce n'est

[1] D'après https://arstechnica.com/information-technology/2014/04/critical-crypto-bug-in-openssl-opens-two-thirds-of-the-web-to-eavesdropping/.

pas la plus populaire des librairies cryptographiques sur Internet. À cette période, la librairie fournissait une implémentation du protocole TLS utilisé par les sites Internet afin de s'authentifier auprès des utilisateurs et d'assurer une communication sécurisée entre eux. Cette implémentation était utilisée par deux tiers de tous les sites Internet[1], ce qui les a rendus par conséquence vulnérable au bug Heartbleed. Ce bug était dû au fait qu'une vérification des bornes était manquante dans le code source d'OpenSSL. Des attaquants ont pu exploiter ce bug afin de récupérer aléatoirement des bouts de données privées depuis les serveurs hébergeant les sites Internet affectés. Ces bouts de mémoire pouvaient tout aussi bien contenir des informations ordinaires que des mots de passe ou encore des clés privées rendant ce bug extrêmement dangereux.

Bien que les cryptographes ont conçu leur schémas de cryptage ou leur protocoles de communication afin qu'ils soient *mathématiquement* corrects et sécurisés, il reste néanmoins que l'implémentation sous-jacente est exécutée dans le monde *physique*. L'exécution de ces implémentations affecte donc le monde de diverses façons qui pourraient faire fuire des informations confidentielles, ces différentes manières sont appelées des canaux cachés. Par exemple, la consommation d'électricité, le bruit engendré, ou encore la durée d'exécution peuvent tous être utilisés afin de récupérer des secrets cryptographiques.

Dans le cas de la consommation d'électricité, un exemple peut être trouvé dans [Koc+11] et est reproduit dans la Figure 1 qui illustre la consommation d'électricité mesurée par un oscilloscope d'une puce calculant une boucle de l'exponentiation module de l'algorithme RSA. La trace montre une séquence de différentes opérations qui peuvent être utilisées pour récupérer la clé secrète. Afin de décoder un message avec l'algorithme RSA, l'exponentiation modulaire est utilisée. Le message encrypté est la *base*, la clé secrète est l'*exposant* et le module est choisi au début de l'algorithme RSA. Une implémentation que l'on pourrait retrouver dans des livres de cours est présentée ci-dessous.

```c
int modular_exp(int base, int exponent, int modulus)
{
  int result = 1;
  base = base % modulus;
  while (exponent > 0) {
    if (exponent % 2 == 1) {
      result = (result * base) % modulus;
    }
    exponent = exponent >> 1;
    base = (base * base) % modulus;
  }
  return result;
```

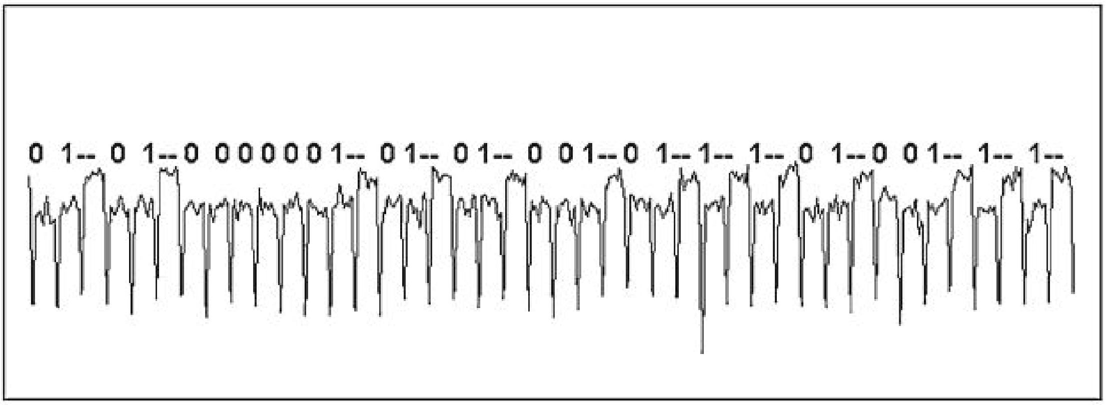

Figure 1: Exemple de Simple Power Analysis de [Koc+11]

```
}
```

La boucle itère sur les bits de l'exposant, c'est-à-dire la clé secrète. Les multiplications consomment plus d'énergie que calculer un carré et apparaissent donc avec des pics plus hauts dans la trace. Puisque les multiplications ne sont exécutées seulement lorsque le bit l'exposant est 1, alors que le calcul du carré est effectué à chaque itération de la boucle, le bit 1 est representé par un léger pic suivi dans grand pic, alors qu'un 0 est représenté par un pic court. Cela nous permet de décrypter la trace illustrée dans la Figure 1. Une limitation de cette approche est qu'elle nécessite un accès direct à l'appareil.

Une autre attaque possible consiste à utiliser le temps déxécution du programme, ce qui peut être accompli à distance (par le réseau par exemple) lorsque les fluctuations sont assez grandes, ce qui est généralement le cas en pratique [Ber05a]. À nouveau dans le cas de l'exponentiation modulaire, lorsque le bit évalué de l'exposant est 1, une multiplication et un calcul de modulo sont effectués en plus par rapport au cas où l'exposant est 0. Par conséquent, la durée de l'exécution est proportionnelle au nombre de bits à 1 dans la clé secrète. Cela diminue grandement le nombre de clés possibles.

D'autres attaques exploitant la durée d'exécution sont présentées dans [Koc96; YGH17; Ber05a], ces attaques exploitent ce qu'on appelle un canal caché temporel. Nous nous concentrons dans cette thèse sur le canal caché temporel car il est considéré l'un des canaux cachés les plus dangereux. En effet, il est exploitable à distance alors que des canaux cachés reposant sur la consommation d'énergie ou le bruit demandent un accès physique à l'appareil visé.

Sécurité constant-time Afin de fermer le canal caché temporel, les cryptographes, les développeurs de librairies cryptographiques ainsi que les ingénieurs de sécurité suivent une disciple de programmation très stricte appellée programmation constant-time[2]. Ce nom est légèrement inapproprié. En effet, les programmes ne sont pas écrits afin que leur exécutions soient litérallement en temps constant, mais seulement en temps constant *par rapport* aux secrets. C'est-à-dire que le temps d'exécution ne dépend pas des secrets. Cela est accompli en s'assurant que le flot de contrôle (les branchements conditionnels) et les acccès mémoire des programmes ne dépendent pas de secrets. Le temps d'exécution d'un programme n'est pas seulement affecté par son flot de contrôle, mais aussi par ses accès mémoire qui sont affectés par le cache. Considérez le scénario suivant illustré dans la Figure 2, un programme cryptographique est exécuté dans le cloud et ses données sont chargées en cache dans la Figure 2a. Dans la Figure 2b, l'attaquant qui partage la même machine dans le clooud exécute à son tour un programme qui va remplacer certaines des lignes de cache par d'autres données. Le programme cryptographtique continue son exécution et charge à nouveau ses données mises en cache. Du point de vue de l'attaquant, il n'est pas possible de savoir à qui appartiennent les données en cache, comme illustré dans la Figure 2c par les cases grises. Lorsque l'attaquant essaie de charger ses propres données, deux scénarios sont possibles. Soit l'accès mémoire est lent, car les données du premier programme ont été remises en cache comme illustré par la Figure 2d, ou l'accès mémoire est rapide car le programme cryptographique n'a pas utilisé cette partie de la mémoire comme montré par la Figure 2e. Dans le premier cas, l'attaquant apprend quelle ligne de cache a été utilisée par le programme cryptographique. Cela est dangereux car certains algorithmes cryptographiques tels qu'AES utilisent des accès mémoire de la forme `table[secret & 0xff]`, l'attaquant peut donc apprendre que la valeur `secret & 0xff` est bornée entre 40 et 47 par exemple dans le cas de la Figure 2d au lieu d'être bornée entre 0 et 255.

Certaines personnes ont proposé des mitigations différentes consistant à repousser la fin des calculs, ce qui permet d'effacer observationnellement l'influence des données sur le temps d'exécution. Cependant, il a été démontré que ces mitigations ne sont pas suffisantes. En effet, [ZS18] présente une attaque sur l'exemple de l'exponentiation modulaire précédent combinant les canaux cachés énergétiques et temporels. En supposant que des instructions inutiles aient été rajoutés à la branche `else` du branchement conditionnel afin d'équilibrer les temps d'exécution, il est possible de retrouver les temps "originaux" en suivant la consommation énergétique puisque les instructions rajoutées consomment moins d'énergie que les autres.

Importance de la vérification formelle Comme la plupart des librairies cryptographiques sont écrites en C, adhérer à la discipline de programmation constant-time demande d'écrire les programmes de façons souvent compliquées et enclins à être éronnées. Il est en effet souvent nécessaire d'utiliser des fonctionnalités basniveau de C

[2] Voir par exemple, https://www.bearssl.org/constanttime.html ou https://cryptocoding.net/index.php/Coding_rules.

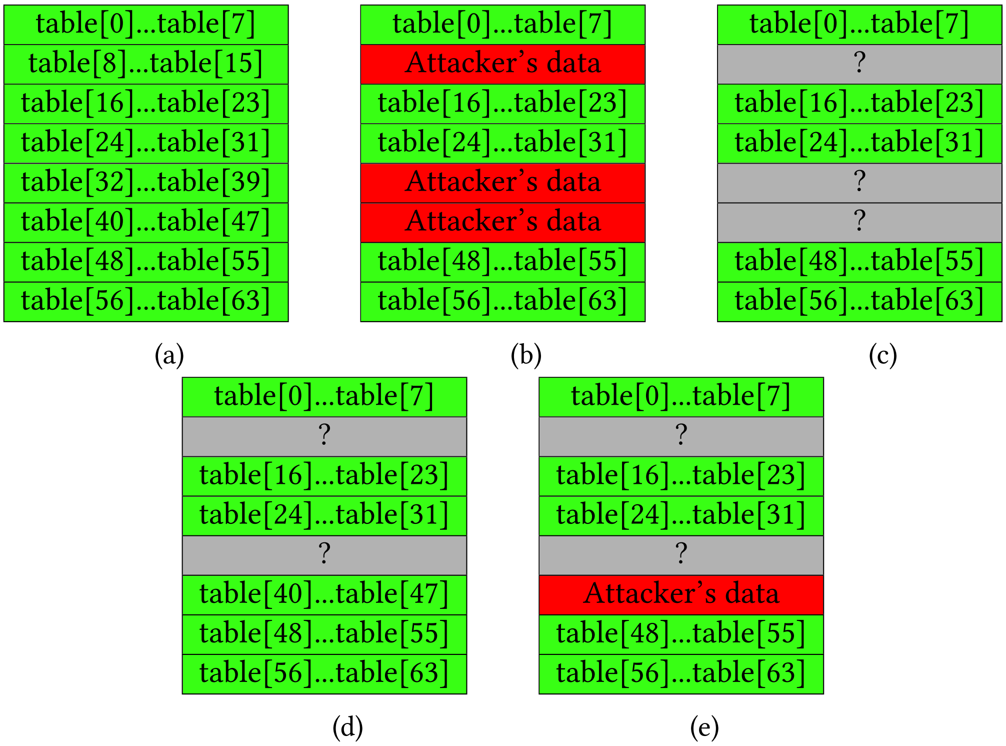

Figure 2: Principe d'une attaque par cache

telles que des manipulations au niveau des bits. Par exemple, simplement choisir entre deux variables `x` et `y` selon un booléen `b`, i.e., `return b ? x : y` peut être réecrit de manière compliquée. Dans la librairie OpenSSL, cela est défini ainsi[3].

```c
unsigned int constant_time_select(bool b,
                                  unsigned int x,
                                  unsigned int y)
{
  unsigned int mask = - (unsigned int) b;
  return (mask & x) | (~mask & y);
}
```

En C, un `bool` représente un seul bit 0 ou 1. Ainsi, lorsque `b` est transtypé en unsigned, `b` reste 0 ou 1 mais est maintenant représenté sur 32 ou 64 bits selon l'architecture machine. En exploitant le comportement des nombre non signés, `mask` peut soit être 0 si `b` est 0, ou `0xf...f` si `b` est 1. Enfin, en utilisant le ET bit à bit `&` et le OU bit à bit `|`, la valeur de retour est `x` si `b` est 1 ou `y` si `b` est 0.

Malgré que la sécurité constant-time soit une discipline de programmation facile à définir, il est difficile de l'appliquer comme le montre la citation suivante de [AP16]:

> At the time of its release, Amazon announced that s2n had undergone three external security evaluations and penetration tests. We show that, despite this, s2n — as initially released — was vulnerable to a timing attack [...].

Cela démontre que les audits de sécurité, bien que nécessaires, sont *insuffisants* afin de s'assurer que des librairies de cryptographie soient sans erreurs. Puisque les conséquences de telles erreurs peuvent étre disastreuses, l'utilisation de *méthodes formelles* devient critique. Les méthodes formelles sont un ensemble de techniques et d'outils reposant sur des bases mathématiques qui peuvent être utilisées afin de vérifier que des implémentations satisfont une spécification donnée.

La *vérification formelle* est une forme plus stricte de méthode formelle où la rigueur mathématique nécessaire à l'utilisation de différentes techniques est déléguée à un programme informatique appelé un *assistant de preuve*. Cela permet de fournir un niveau de confiance sans précédent puisqu'il n'est plus nécessaire de faire confiance au raisonnement des preuves, mais uniquement au vérificateur de preuve qui se trouve être généralement petit et vérifiable manuellement.

[3] Extrait de `https://github.com/openssl/openssl/blob/0d66475908a5679ee588641c43b3cb6a2d6b164a/include/internal/constant_time_locl.h#L220-L225`, le code original utilise directement un "mask" comme argument à la place d'un `bool`. L'exemple est réecrit pour souci d'illustration.

Un des plus récents succès de la vérification formelle est CompCert [Ler06], un compilateur pour le langage C formellement vérifié, le premier compilateur vérifié pour un langage réaliste.

Un compilateur est un outil prenant en entrée des programmes écrits dans un langage *source* et les traduit (compile) dans un langage *cible*. La correction d'un compilateur est prouvé par un théorème dit de *préservation sémantique*. L'intuition de ce théorème est d'exprimer que le compilateur n'introduit pas de bug dans les programmes qu'il compile. CompCert prouve un tel théorème et démontre l'utilité de la vérification formelle dans les domaines critiques comme le montre cette citation de [Yan+11] cherchant des bugs dans des compilateurs:

> The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Contributions et Organisation Nous avons démontré l'importance d'assurer la propriété de sécurité constant-time au début. Il est par conséquent naturel de la joindre à la vérification formelle. Dans cette thèse, nous essayons de répondre à deux défis suivants en appliquant les techniques des méthodes formelles au domaine de la sécurité constant-time.

1. Comment pouvons nous nous assurer qu'un programme satisfait à la sécurité constant-time ?

2. Comment pouvons nous être certain que le code qui est réellement exécuté est constant-time ?

Les deux questions ciblent des niveaux différents de la chaîne de compilation. La première correspond au code source, le programmeur a-t-il bien respecté la sécurité constant-time ? La seconde question cible le code assembleur, le compilateur a-t-il bien respecté l'intention du programmeur ? Est-ce que le code assembleur est encore constant-time ?

- Le chapitre 2 présente les outils sur lesquels nous nous appuierons. Plus précisémemt, l'assistant de preuve Coq, le compilateur C CompCert et l'analyseur statique Verasco. Nous présenterons aussi une étude des domaines de recherche liés à la sécurité constant-time et plus généralement de la cryptographie *high-assurance*.

- Le chapitre 3 présente notre première contribution correspondant à une méthodologie pour améliorer un interprète abstrait et en faire un vérificateur de sécurité constant-time. Un prototype a été implémenté en s'appuyant sur l'analyseur statique Verasco. Ce chapitre est une version longue du travail présenté au 22$^{\text{ème}}$ European Symposium on Research in Computer Security (ESORICS) [BPT17] et a aussi été accepté pour publication au Journal of Computer Security [BPT18].

- Le chapitre 4 présente une méthodologie pour transmettre des résultats d'analyses au niveau source jusqu'à un niveau inférieur. Cela permet d'utiliser des information hautement précise qui n'auraient pas été possibles d'obtenir directement au niveau assembleur. Cette méthodologie a été utilisée afin d'implémenter un analyseur de sécurité constant-time au niveau assembleur. Ce chapitre est basé sur du travail fait en collaboration avec Gilles Barthe et Vincent Laporte, il a été présenté 30$^{\text{ème}}$ Computer Security Foundations Symposium (CSF) [Bar+17].

- Le chapitre 5 présente une méthodologie de preuve pour montrer qu'un compilateur préserve la propriété de sécurité constant-time en adaptant les preuves standards de simulation.

- Finalement, le chapitre 6 conclut cette thèse et présente différentes suites possibles.

CHAPTER 1

INTRODUCTION

Since immemorial times, people have tried to communicate privately in a public setting, i.e., only participants could understand the communication even in presence of eavesdroppers. One early example is Julius Caesar which encoded his military and private correspondence using the eponymous Caesar's cypher. The idea of this encryption technique was to substitute each letter in the original text (plaintext) with a different letter some fixed number of positions down the alphabet. The encoded text (cyphertext) of "hello" would thus be "khoor" with a right shift of 3, i.e., a becomes d, b becomes e, etc.

Cryptography is the discipline of securing communication in the presence of an attacker. Cryptographic primitives are basic algorithms to provide cryptographic services such as encrypting or decrypting a message. These algorithms are usually proven by their designers to be *computationally* secure, i.e., it would take a ludicrous amount of time for an attacker with a great amount of computational power to break its security properties by brute force. Cryptographic protocols are then built on top of these primitives, they specify how to use the primitives in order to communicate securely.

As electronic communications become more and more prevalent in our world through the democratization of the Internet, credit cards, contactless payment, smartphones, etc, cryptography has also become a centerpiece in ensuring that our privacy is secured. Therefore, mistakes in the design or the implementation of cryptographic primitives or protocols could have devastating consequences, whether economical or even life endangering. For instance, applications to privately communicate have become widely used in countries where freedom of speech is not a right and if those communications were revealed, it could considerably endanger their author's lives.

Another example is the Heartbleed bug. In April 2014, a security bug was discovered in the OpenSSL cryptography library[1], one of if not the most popular cryptography

[1] According to https://arstechnica.com/information-technology/2014/04/critical-crypto-bug-in-openssl-opens-two-thirds-of-the-web-to-eavesdropping/.

library on the Internet. At the time, it provided an implementation of the TLS protocol which is the protocol used by websites to authenticate themselves to the end user and ensure secure communication between them. That implementation was used by two thirds of all websites[1], which consequently made them vulnerable because of the Heartbleed exploit. This bug was due to a missing bounds check in the OpenSSL source code. Attackers could exploit this in order to recover random chunks of private memory data from servers running the affected websites. Those chunks of memory could contain mundane information but also passwords or private keys making this bug rather harmful.

While cryptographers design their encryption schemes or communication protocols to be *mathematically* sound and secure to use, there still remains an underlying implementation that runs in the *physical* world. Execution of these implementations affects the world in ways that could leak confidential information, we call these different ways to affect the world side-channels. For instance, power consumption, noise, duration of execution can all be used to recover cryptographic secrets.

In the case of power consumption, an example can be found in [Koc+11] and is reproduced in Figure 1.1 illustrating the power consumption of a chip computing a modular exponentiation loop in RSA measured using an oscilloscope. The trace shows a sequence of different operations that can be used to recover the secret key. In order to decode a message in RSA, modular exponentiation is used. The coded message is considered the *base*, the secret key is the *exponent* and the modulus chosen at the beginning of the RSA algorithm is needed. A textbook implementation of modular exponentiation in C is provided below.

```c
int modular_exp(int base, int exponent, int modulus)
{
  int result = 1;
  base = base % modulus;
  while (exponent > 0) {
    if (exponent % 2 == 1) {
      result = (result * base) % modulus;
    }
    exponent = exponent >> 1;
    base = (base * base) % modulus;
  }
  return result;
}
```

The loop iterates over the bits of the exponent, i.e., the secret key. Multiplications consume more power than squares and thus appear as higher peaks in the trace. As

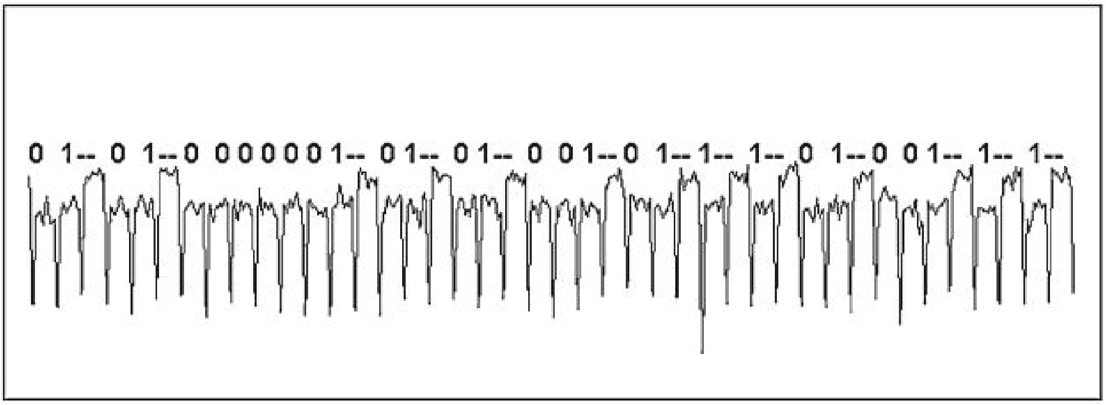

Figure 1.1: Simple Power Analysis example from [Koc+11]

the multiplications are only executed when a bit of the exponent is 1, while the square operation is executed at every iteration of a loop, a 1 bit is represented as a short bump followed by taller bump, while a 0 bit is represented by a short bump, which allows us to decrypt the trace as illustrated in Figure 1.1. One shortcoming of this approach is that it requires direct access to the device.

Another possible attack is (ab)using the execution duration of the program which can be done remotely (i.e., over the network) when fluctuations are noticeable enough, it is generally the case in practice [Ber05a]. Again, in the case of modular exponentiation, when the inspected bit of the exponent is 1, an additional multiplication and a modulo operation are performed. This has for consequence that the execution duration of the function is proportional to the number of 1 in the secret key, which sensibly decreases the number of possible keys.

Other attacks exploiting execution duration have been documented in [Koc96; YGH17; Ber05a], they all exploit what is called a timing side-channel. We focus in this thesis on timing side-channels. They are considered one of the most dangerous side-channels since they can be remotely exploitable while side-channels based on power consumption or noise require physical access to the attacked device.

Constant-time security In order to close the timing side-channel, cryptographers, cryptography libraries developers and security engineers follow a very strict programming discipline called cryptographic constant-time programming[2]. This name is a bit of a misnomer, as they do not intend to make the programs they write literally constant-time, but constant-time *with regards to* secrets, i.e., the running times of programs do not depend on secrets. This is achieved by ensuring that neither control-flow

[2] See for example, `https://www.bearssl.org/constanttime.html` or `https://cryptocoding.net/index.php/Coding_rules`.

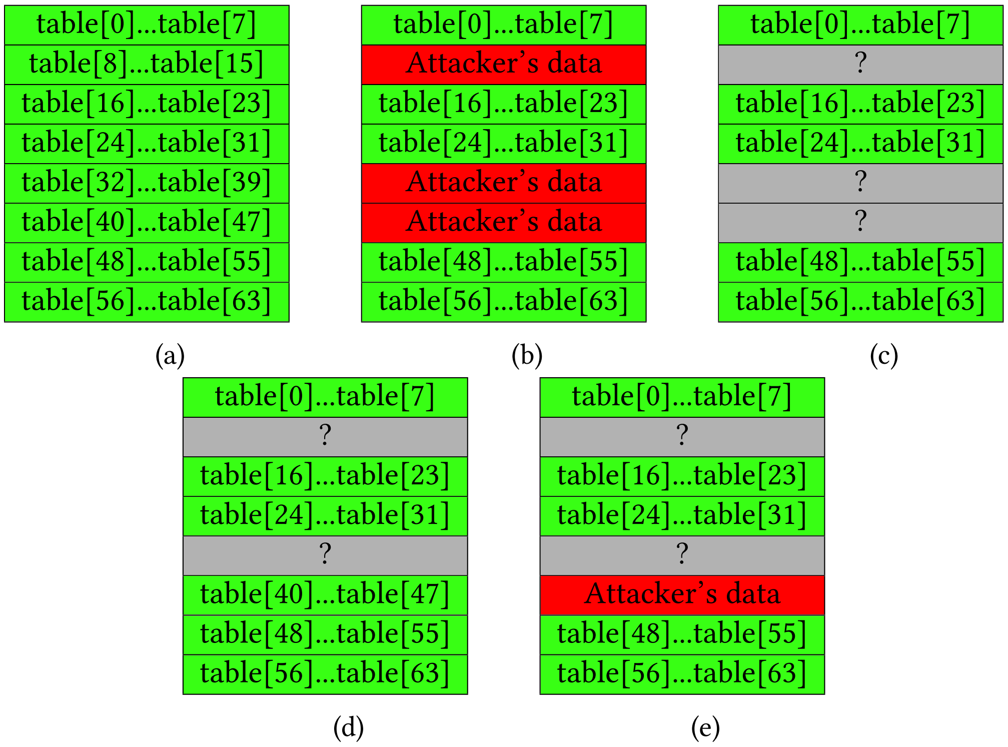

Figure 1.2: General principle of an cache attack

(branchings) nor memory access pattern of the programs depends on secrets. Indeed, not only control flow affects the execution duration of a program, memory accesses also affect it due to cache. Consider the scenario illustrated in Figure 1.2, a cryptography program executing in the cloud has its data loaded into cache in Figure 1.2a. In Figure 1.2b, the attacker program sharing the same host computer replaces some cache lines. The first program continues again and reloads its data, from the point of view of the attacker, it is unknown whose data is in the cache, as illustrated in Figure 1.2c. When the attacker program attempts to load its data, two possible scenarios can happen. Either the memory access is slow because the first program's data has been reloaded into cache as in Figure 1.2d, or the memory access is fast because the first program has not tried to access this part of the memory as in Figure 1.2e. In the first scenario, the attacker learns which cache line the cryptography program tried to access. This is dangerous because some cryptographic algorithms such as AES have memory accesses of the form `table[secret & 0xff]`, the attacker can thus learn that `secret & 0xff` is restricted between 40 and 47 for instance in the case of Figure 1.2d instead of the most general bounds possible 0 and 255.

People have proposed different mitigations by delaying the return time of computations, thus observationally removing the data-dependent timing channel. However, it has been shown that it is not sufficient as attackers become more and more shrewd. For instance, [ZS18] shows an attack that combines the power and timing side-channels on the previous modular exponentiation loop routine where bogus instructions have been added to the `else` branch of the conditional branching so that the branchings are balanced timing-wise, each loop iteration thus takes the same time. They use a power attack in order to recover the "actual" timings of the modular exponentiation loop in RSA as the added delays noticeably consume less power. The usual timing attack described earlier can then be used to recover the secret. Cryptographic constant-time programming would have avoided this attack and is sufficiently secure in practice with "success story after another of constant-time code" as noted by Daniel Bernstein, and proposing other protections is "ridiculous"[3].

Importance of formal verification As most cryptography libraries are written in C, adhering to the constant-time programming discipline usually involves writing programs in a certain way that is oftentimes tricky and error prone as it regularly requires using low-level features of C such as bit-level manipulations. For instance, simply selecting between two variables `x` and `y` based on a selection bit `b`, i.e., `return b ? x : y`, can be rewritten in a complex manner. In OpenSSL, this is defined as follows[4].

```
unsigned int constant_time_select(bool b,
                                  unsigned int x,
                                  unsigned int y)
{
  unsigned int mask = - (unsigned int) b;
  return (mask & x) | (~mask & y);
}
```

In C[5], a `bool` is represented by a single bit 0 or 1. Thus, when casted to unsigned, `b` is still 0 or 1 but is now represented over 32 or 64 bits depending on the architecture instead of only a single bit. Next, by exploiting the wrap around behavior of unsigned integers, `mask` is either 0 if `b` is 0, or `0xf...f` if `b` is 1. Finally, using the bitwise AND operator `&` and bitwise OR operator `|`, `x` is returned if `b` is 1 and `y` is returned if `b` is 0.

Even though constant-time security is a programming discipline simple to define, it is still difficult to correctly enforce as illustrated by the following quote from [AP16]:

[3] Source: `https://twitter.com/hashbreaker/status/902422845069946880`.

[4] Taken from `https://github.com/openssl/openssl/blob/0d66475908a5679ee588641c43b3cb6a2d6b164a/include/internal/constant_time_locl.h#L220-L225`, the original code directly takes a "mask" as argument instead of a `bool`, it was slightly rewritten for illustration purpose.

[5] More precisely, since C99, when `<stdbool.h>` is included.

> At the time of its release, Amazon announced that s2n had undergone three external security evaluations and penetration tests. We show that, despite this, s2n — as initially released — was vulnerable to a timing attack [...].

This shows that security audits, albeit necessary, are *not enough* to be sure that industrial-strength cryptography libraries are free from errors. Because the consequences of implementation errors can be disastrous, the use of *formal methods* become critical. Formal methods are a set of techniques and tools relying on a mathematical foundation that can be used to verify that implementations satisfy a given specification. For instance, a program analysis is a form of formal methods, i.e., it is a tool that takes a program as input and verifies that it satisfies a specification.

Formal verification is a stricter form of formal methods where the mathematical rigor necessary to the application of the different techniques is verified by a computer program called a *proof assistant*. This provides a high-level of confidence as one does not have to trust the reasoning of the proofs, but only trust in the proof checker. The proof checker itself is usually very small in size such that it can be manually verified.

Formal verification has had a list of illustrious successes in recent years in the theoretical areas to more applied areas. For instance, in 2012, Gonthier and his team finally finished specifying and proving the Feit-Thompson theorem (also known as the odd order theorem) in the Coq proof assistant [Gon+13]. In 2006, Leroy and his team presented CompCert [Ler06], a formally verified C compiler, the first verified compiler for a realistic language.

A compiler is a tool that takes programs in a *source* language and translates (compiles) them into a *target* language. The correctness of a compiler is proved through a theorem that is named *semantic preservation theorem*, its intuitive meaning is that a compiler does not introduce bugs in the programs it compiles. In order to give a formal definition of the theorem, we need to introduce the notion of program behavior. A program behavior can be *defined* in which case, the program either *terminates* normally with a return value or *diverges* and the program loops forever. A program behavior can also be *undefined* if the program performs illegal operations such as a division by zero, out-of-bounds memory access, etc. The semantic preservation theorem can then be stated as follows: for all source programs $\mathcal{S}$, if $\mathcal{S}$ is safe (i.e., its behavior is defined) and compiles into target program $\mathcal{T}$, then $\mathcal{T}$ behaves as $\mathcal{S}$.

The behavior of a program is formally defined by its *semantics*, i.e., how it is executed. For instance, the C standard (informally) defines the semantics of C programs. One of the strengths of CompCert is that it gives a formal definition of the semantics for the C language and other languages. This forms the basis on which the proofs in CompCert are built upon. A second exploit accomplished by CompCert is that it gives a strong argument in favor for the use of formal verification in critical domains as illustrated by this quote from [Yan+11] which looked for bugs in compilers:

> The striking thing about our CompCert results is that the middle-end bugs

> we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

CompCert's compiler correctness theorem holds for *safe* programs. This is needed as compilers should be able to optimize away code that may exhibit undefined behavior. For instance, if a variable is assigned the result of a division by zero but is never used afterwards, the compiler should be able to optimize away this code as it is never used for anything useful, i.e., computing a result.

However, how do we make sure that a program is safe? In 2015, the formally verified Verasco static analyzer [Jou+15] bridged the gap and provides a way to verify that a program is safe. A static analyzer is a special form of program analysis in which the analyzed program is not run. However, Rice's theorem [Ric53] states that all non-trivial semantic properties are undecidable; they cannot be automatically verified by a program. This has for consequence that a static analyzer such as Verasco is necessarily incomplete, i.e., it may not be able to verify that a program is safe even though it actually is. In order to be able to analyze as many programs as possible, static analyzers are generally finely tuned and possess complex code. The ground breaking exploit of the Verasco team is to have been able to *formally verify* a static analyzer and secondly, to make it scale to realistic programs. This breakthrough provides a proof that formal verification can be successfully applied to verifying constant-time security, which is a challenge that we tackle in this thesis.

The beginning of this introduction has explained the importance of verifying constant-time security, but what about the second part of this thesis' title, namely "a verified compilation toolchain". Indeed, most cryptographic libraries are written in a high-level source language such as C. However, the language actually understood by computers is not C, but assembly. This raises the question of whether a source program that is constant-time still stays secure after being compiled. Hence, the advantage of placing our work within a verified compilation toolchain is twofold. First, a verified compiler provides formal semantics for the languages it uses, this gives us a way to reason on these languages and more specifically, to define what it means to be constant-time for these languages. Second, this allows us to control the compiler and make sure it won't do anything unexpected to the code and remove all security measures.

Contributions and Organization of this Document. In this thesis, we describe our answers to the following two challenges pertaining to constant-time security:

1. How can we help programmers be certain that their source code adheres to the constant-time security policy ?
2. How can we make sure that the code that *actually* runs is constant-time ?

The two questions target different levels in the compiler toolchain. The first targets the source code which in our case is written in C. We do not need to rely on a compilation toolchain to answer this question. However, using CompCert allows us to reuse its formal semantics of the C language and gives us a way to formally reason on C programs.

The second question targets the assembly code which is notoriously harder to analyze than the source code as most abstractions have been lost during compilation. For instance, structured control-flow such as conditional branchings and loops are no longer available at the assembly level. Furthermore, types are also not available, i.e., it is no longer possible to know whether the value contained by a variable represents an integer, a float or a pointer, etc. We explain how we answer these different challenges as follows.

- In Chapter 2, we give more information on the multiple tools that we use. More specifically, the Coq proof assistant, the CompCert C compiler and the Verasco static analyzer. A particular focus is given to CompCert, upon which most of this work is built upon. We explain the overall architecture of the compilation chain and give background on the proof techniques used for the correctness proof. We also present a survey of research domains related to constant-time security and most specifically to the emerging area of high-assurance cryptography.
- Chapter 3 describes our first contribution which is a methodology to improve an abstract interpreter into a verifier for constant-time security. A prototype for verifying C programs has been made by leveraging the Verasco static analyzer. This allows us to give precise feedback to the programmers and help them understand where information leaks may appear. Experiments show that our tool is competitive with state-of-the-art tools and manages to analyze previously out of reach programs. This chapter is a longer version of the work that has been presented in the 22nd European Symposium on Research in Computer Security (ESORICS) in [BPT17] and has also been accepted for publication in the Journal of Computer Security [BPT18].
- Chapter 4 describes a methodology to translate results of analyses at source level down to assembly. This allows us to use highly precise information for enhancing analyses at the assembly level. The methodology has been instantiated on points-to information provided by Verasco and then used to design a verifier for constant-time security at assembly level. This chapter is based on joint work with Gilles Barthe and Vincent Laporte, it has been presented at the 30th

Computer Security Foundations Symposium (CSF) in [Bar+17]. Our personal contribution in this work is the implementation of the defensive encoding of points-to annotations as well as its formal proof of correctness.

- While the previous chapter can be considered a possible answer to the second question we asked, it uses *a posteriori* validation, meaning that each compiled program must be checked for security. Chapter 5 provides a more direct answer to the question by describing a proof methodology to show that a compiler preserves constant-time security, i.e., if the source code is constant-time, then so is the compiled code. It adapts the standard simulation-based proofs used for proving safety preservation in formally verified compilers and details some experiments in modifying CompCert. An early version of this work has been presented at the workshop on Foundations of Computer Security (FCS) in 2018.
- Finally, Chapter 6 concludes this thesis and summarizes the results we achieved. Some perspectives are also offered and we also compare different views of how the challenge presented in Chapter 5 can be tackled.

CHAPTER 2

CONTEXT

Before detailing our technical contributions in the next chapters, we present in this chapter the tools that our work builds upon and we also present a survey of related work on the verification of security properties.

2.1 Formal Verification of Programming Tools

We present in this section the tools that our work builds upon, namely the Coq proof assitant, the CompCert C compiler and the Verasco static analyzer.

2.1.1 Coq proof assistant

The thesis is intended to be readable with very little knowledge of Coq, we only provide here a broad overview of this proof assistant.

2.1.1.1 Proof assistant

The second half of the 20$^{\text{th}}$ century has seen the emergence of proof assistants, computer programs that allow users to do logical reasoning within a mathematical theory. The major reason for this emergence was due to the increasing complexity of mathematical proofs for which experts must spend months or years to manage to understand some proofs and vet them. A recent example is Mochizuki's proofs for several famous conjectures in number theory in 2012. Six years laters, in 2018, there is still no consensus among the mathematical community towards acceptance of the proofs, not towards their refutation. This is mainly due to the fact that these proofs rely on a brand new so called inter-universal Teichmüller theory and "the actual length [of the proof] is about 550 pages. But to understand his theory, one also has to know well various appropriate prerequisites, so we are talking, approximately, about 1000 pages of prerequisites and

550 pages of IUT theory”[1].

An example of success of proof assistants is Gonthier’s proof of the four color theorem in 2005[2] which states that the regions of any planar map can be colored with only four colors, in such a way that any two adjacent regions have different colors. The original proof dates back from 1976 when Appel and Haken provided the first computer-aided proof in history. The proof consisted in reducing the infinite number of possible configurations to 1,936 configurations (and later 1,476). This reduction was proven correct on paper and could be reasonably checked by fellow mathematicians. However, the remaining configurations had to be checked one by one by a computer program, which the mathematicians were not exactly comfortable with as they could not verify its correctness. The advantage of using a proof assistant appears here, one can use it to both program and verify at the same time. If one trusts the proof assistant, all that is needed to do is then to check that the definitions are right and that theorems are the ones we want. In the case of the four color theorem, this all fits on a single A4 page according to Gonthier (p.14, [Geu09]), which is quite easily verifiable by a human.

Finally, with the proliferation of the usage of computers, more specifically in critical systems, this also calls for the use of proof assistants in order to reason about programs that are often too colossal to manually verify.

2.1.1.2 Coq

Coq is a proof assistant based on a dependently typed theory, the Calculus of Inductive Constructions which allows users to write programs and proofs in the same language. The verification of proofs is based on the *Curry-Howard correspondence* which presents the analogy between proofs and programs. A *type* can be seen as the statement of a *property*, while a *term* (program) of this type can be seen as a *proof* of this property. This means that one only has to check that a term has a certain type to know that the property is proved and true. One can wonder if a relatively massive proof assistant such as Coq can be trusted, but a consequence of the Curry-Howard correspondence is that we only need to trust its type-checker and not how the terms are constructed, this is a reasonably smaller thing to manually verify.

Let’s illustrate the usage of Coq with an example. Inductive types can be defined in Coq using the Inductive keyword and by providing a set of rules to construct its terms. This means that a term of this type can only be constructed by using these rules, they are also called *constructors*. For instance, below we define the type[3] of conjunctions of propositions.

[1] Ivan Fesenko, `http://www.ams.org/news?news_id=3711`.

[2] `http://www.ams.org/notices/200811/tx081101382p.pdf`

[3] `Prop` denotes the type of propositions in Coq, while `Type` correponds to data. This separation is not necessary in theory but is useful for extraction of programs in OCaml or Haskell which only uses the computational parts (i.e., the things in `Type`) of a development.

```
Inductive and (A B: Prop) : Prop :=
| conj (a: A) (b: B): and A B.
```

This means that a term of type `and A B` is necessarily of the form `conj a b` where `a` is a proof (or a term) of `A` and `b` is a proof of `B`. Now, in order to prove that $A \wedge B$ implies A for instance, we must build a term of the following type.

```
forall (A B: Prop), and A B -> A
```

This corresponds to a function that "projects" the left part of the conjunction.

```
fun (A B: Prop): and A B -> A =>
  fun (ab: and A B): A =>
    match ab with
    | conj a b => a
    end.
```

Naturally, manually building proof terms for complex properties rapidly becomes impossible. Coq provides a set of *tactics* to interactively build the proof term. For instance, here is a proof of the previous statement.

```
Lemma proj1: forall (A B: Prop), and A B -> A.
Proof.
  intros A B ab. (* Introduce the hypotheses *)
  destruct ab as [a b]. (* We destruct ab to say that it is necessarily
                           of the form conj a b *)
  apply a. (* We need to prove A, and a is a proof of A by definition *)
Qed.
```

We have given a very simple use of inductive types, but they are far more general. Specifically, they can be used to define *semantics* of language, each constructor corresponds to a semantic rule. An example will be presented in the next subsection.

One important feature of the Coq proof assistant is *extraction*, Coq can be used to transform executable specification written in Coq to executable code in languages such as OCaml or Haskell. This allows users to obtain relatively efficient code without having to manually translate the Coq specifications into more standard languages which could be a task prone to errors. This is especially advantageous in some cases such as compilers or static analyzers.

2.1.2 CompCert

CompCert is a moderately optimizing C compiler. It compiles C source code into assembly language for four different architectures: x86, PowerPC, ARM and more recently RISC-V. CompCert is a formally verified compiler, in a sense that we will define later. It is written, specified and proven in the Coq proof assistant. This mechanization of the correctness of CompCert gives it an unprecedented level of confidence in a compiler.

We will first give an introduction to what is a formal semantics and then, what it means to be a formally verified compiler. Next, we present the proof method used in CompCert to prove compiler correctness. We finish by presenting the overall architecture of the CompCert compiler.

2.1.2.1 Formal Semantics

A compiler is used to translate programs written in a source language $\mathcal{S}$ into programs written in a target language $\mathcal{T}$ such that compiled programs behave similarly to their source programs. It is thus necessary to define how a program behaves. The C standard provides an informal specification of the meaning of C programs. However, formal verification relies on a formal and explicit specification of the meaning of programs, this is called a *formal semantics*.

Formal semantics can be defined in multiple ways which are all equivalent, most popular among them are denotational semantics [SS71], axiomatic semantics [Flo67] and operational semantics [Plo81]. Denotational semantics describes the meaning of programs by associating them to their *denotations*, i.e., a mathematical function representing what a program does. Axiomatic semantics define the meaning of a program by giving proof rules to reason about the program, the most known example of axiomatic semantics is Hoare logic [Hoa69]. Operational semantics is the form of semantics used by CompCert, it describes the meaning of programs by interpreting them as sequences of computational steps, i.e., a transition system.

Definition 2.1 (Labeled transition system (LTS))**.** A labeled transition system is a tuple $(\Sigma, \mathcal{E}, I, F, \rightarrow)$ where Σ is a set of states, $\mathcal{E}$ is a set of events including a silent event ε, $I \subseteq \Sigma$ is a set of initial states, $F \subseteq \Sigma$ is a set of final states and $\rightarrow \subseteq \Sigma \times \mathcal{E} \times \Sigma$ is a set of transitions (σ, e, σ'), also written $\sigma \xrightarrow{e} \sigma'$ which describes a transition from state σ to state σ' emitting an event e.

Events are used to model what is observable by an external observer, for instance, outputs and inputs. These events are language agnostic and are used to describe the *observable behavior* of a program by concatenating them into a *trace*. We define an observable behavior as follows.

Definition 2.2 (Observable behavior)**.** Let $S = (\Sigma, \mathcal{E}, I, F, \longrightarrow)$ be a labeled transition system corresponding to a program P. An observable behavior of P has one of the following forms:

- Terminates($e_0 \ldots e_n$) if there exists $(\sigma_i)_{0 \leq i \leq n+1}$ such that $\sigma_0 \in I$ and for all $0 \leq i \leq n+1$, $\sigma_i \xrightarrow{e_i} \sigma_{i+1}$, and $\sigma_{n+1} \in F$.

- GoesWrong($e_0 \ldots e_n$) if there exists $(\sigma_i)_{0 \leq i \leq n+1}$ such that $\sigma_0 \in I$ and for all $0 \leq i \leq n+1$, $\sigma_i \xrightarrow{e_i} \sigma_{i+1}$, $\sigma_{n+1} \notin F$, and there exists no σ_{n+2} and e_{n+1} such that $\sigma_{n+1} \xrightarrow{e_{n+1}} \sigma_{n+2}$.

- Diverges($(e_i)_{i \in \mathbb{N}}$) if there exists $(\sigma_i)_{i \in \mathbb{N}}$ such that $\sigma_0 \in I$ and for all $i \in \mathbb{N}$, $\sigma_i \xrightarrow{e_i} \sigma_{i+1}$.

The first case corresponds to an execution that terminates normally, i.e., on a final state. The second case corresponds to an execution of the program that *went wrong*, i.e., the execution is stuck on a non-final state, no more step can be taken. The third case corresponds to an infinite execution of the program, for instance, an infinite loop. The program is said to be diverging.

The behavior of a program P is then simply defined as the set of all observable behaviors of P, it is noted Beh(P). We say that a state is safe if it can eventually silently (i.e., all the produced events are silent) reach a final state or there exists an infinite silent execution from this state or it can reach a state from which the next step is non-silent, i.e.,

$$\text{safe}(\sigma) \Leftrightarrow (\forall \sigma', \sigma \xrightarrow{\varepsilon}^{*} \sigma' \implies \sigma' \in F \vee \exists e, \exists \sigma'', \sigma' \xrightarrow{e} \sigma'')$$

2.1.2.2 Formally Verified Compiler

A compiler is a program that translates programs written in a source language $\mathcal{S}$ into programs written in a target language $\mathcal{T}$. For instance, a C compiler usually generates assembly programs from C programs.

A formally verified compiler is a compiler than provides formal guarantees about the code it produces. The intuitive notion that we would like a compiler to satisfy is that the generated code should behave as its source code. This corresponds to semantic preservation. [Ler09] provides multiple possible instantiations of the notion and gradually refines them. The strongest notion of semantic preservation is called bisimilarity and is defined as follows.

Definition 2.3 (Bisimilarity)**.** Two programs S and T are bisimilar if both programs have exactly the same set of possible behaviors, i.e., Beh(S) = Beh(T).

Bisimilarity intuitively captures the notion of equivalent programs, i.e., both programs behave identically. This is however too strong a notion in the case of compilation. For instance, in the case of CompCert, the target language (assembly) is deterministic while the source C language is not. The compiler should be free to choose one particular evaluation order. For instance, consider `z = x + x++`, this can either be compiled into `a = x; b = x; x = x + 1; z = a + b` or `a = x; x = x + 1; b = x; z = b + a` depending on whether the compiler decides to evaluate `x` or `x++` first. This effectively reduces the set of possible behaviors. This is not only an issue with CompCert, but for all compilers.

To take account of this constraint, a possible refinement of the property is *backward simulation* and is defined as follows.

Definition 2.4 (Backward simulation)**.** All the behaviors of program T are included in the behaviors of program S, i.e., $\mathrm{Beh}(T) \subseteq \mathrm{Beh}(S)$.

This definition allows compilers to reduce non-determinism by choosing a particular evaluation order, it is still however sligthly too strong. Indeed, this definition implies that if S "goes wrong", then necessarily T must also go wrong. This requirement is too restrictive as it is violated by multiple desirable compiler optimizations. For instance, consider a program that contains `x = x / 0` at the beginning but never uses `x` afterwards. The original program goes wrong due to a division by zero, but dead code elimination would remove this code as it is never used afterwards and this would result in a compiled program that does not go wrong on this division. A more flexible notion is to restrict preservation of behaviors for *safe* source programs, i.e., programs that do not go wrong.

Definition 2.5 (Backward simulation for safe programs)**.** If S is safe, then all the behaviors of program T are included in the behaviors of program S, i.e., $\mathrm{Safe}(S) \implies \mathrm{Beh}(T) \subseteq \mathrm{Beh}(S)$.

This is the compiler correctness property proven in CompCert. This is an interesting property as a direct consequence of this is that if S is safe then so is T. Indeed, if T could go wrong, then necessarily S cannot be safe, an intuitive interpretation of this is that *the compiler does not introduce bugs.*

2.1.2.3 Simulation Relations

We explain here the proof techniques used in CompCert for proving compiler correctness. The methodology relies on simulation relations that are used to relate program states through whole executions. This is a common technique used when reasoning with operational semantics.

Definition 2.6 (Simulation relation for a backward simulation)**.** Let $S = (\Sigma_1, \mathcal{E}_1, I_1, F_1, \rightarrow_1$) and $T = (\Sigma_2, \mathcal{E}_2, I_2, F_2, \rightarrow_2)$ be two labeled transition systems, binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a simulation relation for a backward simulation as defined in Definition 2.5 if:

- given an initial state σ_2 of T, there exists an initial state $\sigma_1$of S matching σ_2:

$$\forall \sigma_2 \in I_2, \exists \sigma_1 \in I_1, \sigma_1 \mathcal{R} \sigma_2$$

- every final state σ_2 of T is only matched with a state σ_1 in S that will eventually silently reach a final state σ_1':

$$\forall \sigma_1 \in \Sigma_1, \forall \sigma_2 \in F_2, \sigma_1 \mathcal{R} \sigma_2 \implies \exists \sigma_1' \in F_1, \sigma_1 \xrightarrow{\varepsilon}_1^* \sigma_1'$$

- (progress) for every safe state σ_1 of S and matching state σ_2 of T, either σ_2 is a final state, or there exists a possible step from σ_2:

$$\forall \sigma_1 \in \Sigma_1, \forall \sigma_2 \in \Sigma_2, \sigma_1 \mathcal{R} \sigma_2 \implies \text{safe}(\sigma_1) \implies \sigma_2 \in F_2 \lor \exists e, \exists \sigma_2', \sigma_2 \xrightarrow{e}_2 \sigma_2'$$

- (simulation) for any safe state $\sigma_1 \in \Sigma_1$ and any matching state $\sigma_2 \in \Sigma_2$ such that σ_2 advances to some state $\sigma_2' \in \Sigma_2$, there exists a state $\sigma_1' \in \Sigma_1$ that can be reached by σ_1 such that σ_1' and σ_2' are matched:

$$\forall \sigma_1 \in \Sigma_1, \forall \sigma_2 \in \Sigma_2, \sigma_1 \mathcal{R} \sigma_2 \implies \text{safe}(\sigma_1) \implies \forall e, \forall \sigma_2', \sigma_2 \xrightarrow{e}_2 \sigma_2' \implies \exists \sigma_1', \sigma_1 \xrightarrow{e}_1^* \sigma_1'$$

Proving that a binary relation is a simulation relation for a backward simulation suffices to prove that there exists a backward simulation for safe programs. However, this demands to inductively reason on steps of the *target* program which is quite uncomfortable. Indeed, a step in the source execution is often compiled into multiple instructions at the machine level. It is thus necessary to look at multiple instructions before one can guess the corresponding source expression. Furthermore, optimizations can make this very difficult. This is occasionally unavoidable, but when the target language is deterministic as it is often the case, it can be possible to use a *forward simulation* that implies the backward simulation.

Definition 2.7 (Forward simulation for safe programs)**.** If S is safe, then all the behaviors of program S are included in the behaviors of program T, i.e., $\text{Safe}(S) \implies \text{Beh}(S) \subseteq \text{Beh}(T)$

If T is deterministic, it can be easily seen why Definition 2.7 implies Definition 2.5. Indeed, $\text{Beh}(T)$ is a singleton as T is deterministic and since $\text{Beh}(S) \subseteq \text{Beh}(T)$, it ensues that $\text{Beh}(S) = \text{Beh}(T)$, thus Definition 2.5 is implied.

A simulation relation for a forward simulation is defined as follows.

Definition 2.8 (Simulation relation for a forward simulation)**.** Let $S = (\Sigma_1, \mathcal{E}_1, I_1, F_1, \rightarrow_1)$ and $T = (\Sigma_2, \mathcal{E}_2, I_2, F_2, \rightarrow_2)$ be two labeled transition systems, binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a simulation relation for a forward simulation as defined in Definition 2.7 if:

$$\begin{tikzcd}
\sigma_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "\varepsilon"'] & \sigma_2 & \text{or} & \sigma_1 \arrow[r, dash, "\mathcal{R}"] \arrow[d, "e"'] & \sigma_2 \arrow[d, dashed, "+e"] \\
\sigma_1' \arrow[ur, dash, dashed, "\mathcal{R}", "m(\sigma_1)>m(\sigma_1')"'] & & & \sigma_1' \arrow[r, dash, dashed, "\mathcal{R}"'] & \sigma_2'
\end{tikzcd}$$

Figure 2.1: Star Forward Simulation
(Hypotheses in plain lines, conclusion in dashed lines)

- for all initial state of S, there exists a matching initial state of T:

$$\forall \sigma_1 \in I_1, \exists \sigma_2 \in I_2, \sigma_1 \mathcal{R} \sigma_2$$

- any final state of S can only be matched with a final state of T:

$$\forall \sigma_1 \in F_1, \forall \sigma_2 \in \Sigma_2, \sigma_1 \mathcal{R} \sigma_2 \implies \sigma_2 \in F_2$$

- (star diagram) for any matching states $\sigma_1 \in \Sigma_1$ and $\sigma_2 \in \Sigma_2$, if σ_1 advances, then σ_2 can match this step:

$$\forall \sigma_1 \in \Sigma_1, \forall \sigma_2 \in \Sigma_2, \sigma_1 \mathcal{R} \sigma_2 \implies \forall \sigma_1', \forall e, \sigma_1 \xrightarrow{e}_1 \sigma_1' \implies \exists \sigma_2', \sigma_2 \xrightarrow{e}_2^* \sigma_2' \wedge \sigma_1' \mathcal{R} \sigma_2'$$

Proving that these properties are satisfied is sufficient to prove that T "simulates" S. A more graphical representation of the third point is illustrated in Figure 2.1, when σ_2 does not advance (left figure), it is necessary that a "measure" decreases, i.e., a function that maps states to natural integers such that $m(\sigma_1) > m(\sigma_1')$, this is to ensure that there is no infinite "stuttering". Indeed, was it not the case, it would be possible to prove that an infinite source execution is simulated by a finite target one.

There are other forms of diagram that do not require a measure function, for instance a *lockstep* or *plus* diagram, they are illustrated in Figure 2.2. They both imply the original star diagram but are easier to prove. Indeed, the lockstep diagram requires that each step in the source execution is simulated by a single corresponding step in the target simulation. As the target execution always advances, there is no need to define a decreasing measure. Similarly, the plus diagram requires that each step in the source execution is simulated by a positive number of steps in the target execution.

The general idea of using simulation relations is that it allows to inductively reason on a *local* step, but concludes on the *global* behavior of the program by chaining the diagrams one after another.

For instance, suppose that we have an execution starting from an initial state σ_0 to final state σ_f. If we assume that $\mathcal{R}$ is a simulation relation for a forward simulation, by the first point of Definition 2.8, there exists an initial state σ_0' that matches σ_0. By using

$$\begin{tikzcd}
\sigma_1 \arrow[r, dash, "\sim"] \arrow[d, "e"'] & \sigma_2 \arrow[d, dashed, "e"] \\
\sigma_1' \arrow[r, dash, dashed, "\sim"'] & \sigma_2'
\end{tikzcd}$$

(a) Lockstep diagram

$$\begin{tikzcd}
\sigma_1 \arrow[r, dash, "\sim"] \arrow[d, "e"'] & \sigma_2 \arrow[d, dashed, "+e"] \\
\sigma_1' \arrow[r, dash, dashed, "\sim"'] & \sigma_2'
\end{tikzcd}$$

(b) Plus diagram

Figure 2.2: Forward simulations

$$\begin{tikzcd}
\sigma_0 \arrow[r, "e_0"] \arrow[d, dash, "\mathcal{R}"] & \sigma_1 \arrow[r, "\varepsilon"] \arrow[d, dash, "\mathcal{R}"] & \sigma_2 \arrow[r, "e_2"] & \sigma_3 \arrow[r, dash] \arrow[d, dash, "\mathcal{R}"] & \cdots \arrow[r] & \sigma_f \arrow[d, dash, "\mathcal{R}"] \\
\sigma_0' \arrow[r, "e_0"] & \sigma_1' \arrow[r, "e_2"] \arrow[ur, dash, "\mathcal{R}"'] & \sigma_2' \arrow[r, "\varepsilon"] & \sigma_3' \arrow[r, dash] & \cdots \arrow[r] & \sigma_f'
\end{tikzcd}$$

the third point of the definition, σ_0' can match the step made from σ_0 to σ_1, for instance, $\sigma_0' \xrightarrow{e_0} \sigma_1'$. It's also possible that the target program makes no step, for instant when the source program does $\sigma_1 \xrightarrow{\varepsilon} \sigma_2$, σ_1' does not advance, which is possible since the event ε is silent. Conversely, there can also be multiple steps, as illustrated by that when $\sigma_2 \xrightarrow{e_2} \sigma_3$, it is simulated by two steps $\sigma_1' \xrightarrow{e_2} \sigma_2' \xrightarrow{\varepsilon} \sigma_3'$. The crux is that the diagrams can be chained consecutively in order to show that the behavior of the top program is simulated by the bottom one as illustrated by the figure, both programs have the same trace of events.

2.1.2.4 Architecture of the CompCert Compiler

CompCert is a moderately optimizing compiler for the C language, it targets four different architectures: x86, PowerPC, ARM and more recently RISC-V. The compiler goes through 10 intermediate languages composed of an architecture independent *front-end* and an architecture dependent *backend*. CompCert considers a common memory model for all intermediate languages. The architecture of the compiler is represented in Figure 2.3, the top line from CompCert C to Cminor represents the front-end while the rest forms the backend.

The compilation starts by choosing an evaluation order and effectively determinizing the semantics of C. It should be noted that this pass is the only one directly proven using a backward simulation instead of a forward one as it is not possible otherwise. Indeed, the forward simulation property does not hold, there may be some behaviors of the (non-deterministic) source program that do not appear in the transformed program.

After determinization, the next step is pulling variables which addresses are not taken (scalar variables) and putting them into temporaries, i.e., pseudo registers at the Clight level. This is then followed by type elimination which makes explicit which

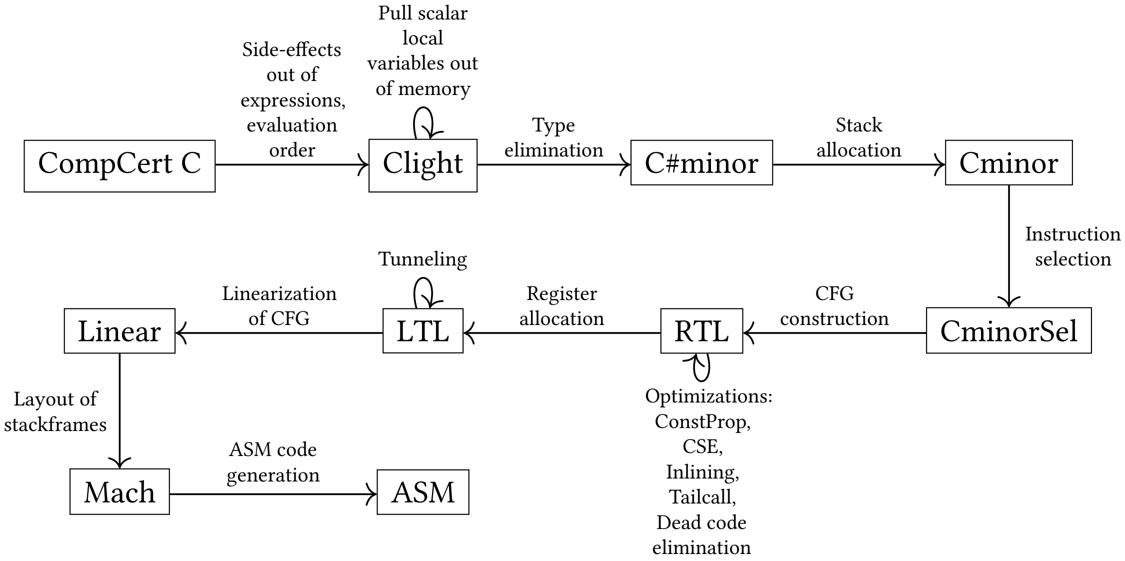

Figure 2.3: Architecture of the CompCert compiler

operators should be used. For instance, instead of having only one addition operator and having to guess from the context which exact addition semantics to use, it can be transformed into an addition for longs, floats, pointers, etc. The last step of the front-end is stack allocation where a stackframe is built for every function, and accesses to non-scalar variable are translated into accesses into the stackframe.

The backend starts by instruction selection in which the compiler takes advantage of the instructions available on the target architecture. For instance, multiplication by 2 can be replaced by a logical left shift. CminorSel programs are then transformed into RTL programs. RTL stands for register transfer language, the functions are represented by control-flow graphs and possess infinitely many pseudo-registers. As the structure of the language is simple, RTL is the host for most of the optimizations found in CompCert such as constant propagation, dead code elimination, inlining, common subexpression elimination and tailcall recognition.

After the optimizations, RTL code is transformed into LTL code through register allocation. LTL programs are roughly the same as RTL programs but only manipulate finitely many registers. The code is then linearized into Linear code, and further transformed into Mach code through the Stacking pass where stackframes of functions are made mode concrete. The machine specific layout of stackframes is specified at this level. Finally, assembly code is generated.

As RTL is the intermediate language where most of the optimizations happen and which we heavily relied on in Chapter 4, we will detail its definition here. Similarly to all other intermediate languages, RTL represents programs as a list of functions definitions and global variables. Each function is represented by a control-flow graph

Instructions:

$i ::= \mathsf{nop}(l)$	no operation (go to l)
$\mid \mathsf{op}(op, \vec{r}, r, l)$	arithmetic operation
$\mid \texttt{if}(cond, \vec{r}, l_{true}, l_{false})$	if statement
$\mid \texttt{return}$	function end
$\mid \texttt{load}(\kappa, addr, \vec{r}, r, l)$	memory load
$\mid \texttt{store}(\kappa, addr, \vec{r}, r, l)$	memory store
$\mid \texttt{call}(sig, id, \vec{r}, r, l)$	function call
$\mid \texttt{return}\ r$	function return

Control-flow graphs:

$f: \quad l \mapsto i$	finite map

Figure 2.4: Syntax of the RTL intermediate language

```
Inductive instruction: Type :=
| Inop: node -> instruction
| Iop: operation -> list reg -> reg -> node -> instruction
| Icond: condition -> list reg -> node -> node -> instruction
| Ireturn: option reg -> instruction
| Iload: memory_chunk -> addressing -> list reg -> reg -> node -> instruction
| Istore: memory_chunk -> addressing -> list reg -> reg -> node -> instruction
| Icall: signature -> reg + ident -> list reg -> reg -> node -> instruction
```

Figure 2.5: Coq definition of the RTL syntax

with explicit program points. A control-flow graph is represented by a mapping from program points to instructions which are detailed in Figure 2.4. To illustrate, the Coq definition of the syntax is illustrated in Figure 2.5.

Instructions in RTL can either be a no-op $\mathsf{nop}(l)$ which simply jumps to instruction at program point l, an arithmetic operation $\mathsf{op}(op, \vec{r}, r, l)$ which uses the values held in registers $\vec{r}$ to compute operation op and stores the result in register r. $\texttt{if}(cond, \vec{r}, l_{true}, l_{false})$ computes the condition $cond$ using the values held in registers $\vec{r}$ and jumps to l_{true} if the condition holds and l_{false} otherwise. $\texttt{load}(\kappa, addr, \vec{r}, r, l)$ and $\texttt{store}(\kappa, addr, \vec{r}, r, l)$ respectively represent a memory read and a memory write. The κ is used to indicate the chunk size of the memory to access, i.e., 8, 16, 32 or 64 bits. The addressing is provided by $addr$ and registers $\vec{r}$ and in the case of a memory read, the value is written in register r, while the value in register r is written in the case of a memory write. $\texttt{call}(sig, id, \vec{r}, r, l)$ calls the function id with signature sig and uses the values held in registers $\vec{r}$ as arguments of the called function. The returned value is stored in register *r*. The last instruction is the `return` instruction which can return no value or a value in a register.

Formally, the smallstep semantics of RTL is presented in Figure 2.6. The execution states of RTL are defined as follows.

```
  Inductive stackframe : Type :=
  | Stackframe:
      forall (res: reg) (* where to store the result *)
             (f: function) (* calling function *)
             (sp: val) (* stack pointer in calling function *)
             (pc: node) (* program point in calling function *)
             (rs: regset), (* register state in calling function *)
      stackframe.

Inductive state : Type :=
  | State:
      forall (stack: list stackframe) (* call stack *)
             (f: function) (* current function *)
             (sp: val) (* stack pointer *)
             (pc: node) (* current program point in c *)
             (rs: regset) (* register state *)
             (m: mem), (* memory state *)
      state
  | Callstate:
      forall (stack: list stackframe) (* call stack *)
             (f: fundef) (* function to call *)
             (args: list val) (* arguments to the call *)
             (m: mem), (* memory state *)
      state
  | Returnstate:
      forall (stack: list stackframe) (* call stack *)
             (v: val) (* return value for the call *)
             (m: mem), (* memory state *)
      state.
```

An execution state can either be a `State`, a `Callstate` or a `Returnstate`. All states contain a list of stackframes which records a list of suspended functions. A `Callstate` represents the moment when the execution is about to enter a function, while a `Returnstate` represents the moment when the execution is returning from a function. This is illustrated by the `exec_Icall` and `exec_return` rules. The first one states that if a function fd is called, then the current function f is added to the call stack s. Conversely, the exec_return rule states that execution from a Returnstate is returned to the function at the top of the call stack.

```
Inductive step: state -> trace -> state -> Prop :=
  | exec_Inop:
      forall s f sp pc rs m pc',
      (fn_code f)!pc = Some(Inop pc') ->
      step (State s f sp pc rs m)
        E0 (State s f sp pc' rs m)
  | exec_Iop:
      forall s f sp pc rs m op args res pc' v,
      (fn_code f)!pc = Some(Iop op args res pc') ->
      eval_operation ge sp op rs##args m = Some v ->
      step (State s f sp pc rs m)
        E0 (State s f sp pc' (rs#res <- v) m)
  | exec_Iload:
      forall s f sp pc rs m chunk addr args dst pc' a v,
      (fn_code f)!pc = Some(Iload chunk addr args dst pc') ->
      eval_addressing ge sp addr rs##args = Some a ->
      Mem.loadv chunk m a = Some v ->
      step (State s f sp pc rs m)
        E0 (State s f sp pc' (rs#dst <- v) m)
  | exec_Istore:
      forall s f sp pc rs m chunk addr args src pc' a m',
      (fn_code f)!pc = Some(Istore chunk addr args src pc') ->
      eval_addressing ge sp addr rs##args = Some a ->
      Mem.storev chunk m a rs#src = Some m' ->
      step (State s f sp pc rs m)
        E0 (State s f sp pc' rs m')
  | exec_Icond:
      forall s f sp pc rs m cond args ifso ifnot b pc',
      (fn_code f)!pc = Some(Icond cond args ifso ifnot) ->
      eval_condition cond rs##args m = Some b ->
      pc' = (if b then ifso else ifnot) ->
      step (State s f sp pc rs m)
        E0 (State s f sp pc' rs m)
  | exec_Icall:
      forall s f sp pc rs m sig ros args res pc' fd,
      (fn_code f)!pc = Some(Icall sig ros args res pc') ->
```

```
    find_function ros rs = Some fd ->
    funsig fd = sig ->
    step (State s f sp pc rs m)
      E0 (Callstate (Stackframe res f sp pc' rs :: s) fd rs##args m)
| exec_Ireturn:
    forall s f stk pc rs m or m',
    (fn_code f)!pc = Some(Ireturn or) ->
    Mem.free m stk 0 f.(fn_stacksize) = Some m' ->
    step (State s f (Vptr stk Ptrofs.zero) pc rs m)
      E0 (Returnstate s (regmap_optget or Vundef rs) m')
| exec_function_internal:
    forall s f args m m' stk,
    Mem.alloc m 0 f.(fn_stacksize) = (m', stk) ->
    step (Callstate s (Internal f) args m)
      E0 (State s f (Vptr stk Ptrofs.zero)
                f.(fn_entrypoint)
                (init_regs args f.(fn_params))
                m')
| exec_return:
    forall res f sp pc rs s vres m,
    step (Returnstate (Stackframe res f sp pc rs :: s) vres m)
      E0 (State s f sp pc (rs#res <- vres) m).
```

Figure 2.6: Coq definition of the RTL semantics

The way to read the Coq definition of the semantic rules is that if all preconditions are satisfied, then the step can happen. For instance, for the simplest rule exec_Inop, only the (fn_code f)!pc = Some(Inop pc') precondition needs to be satisfied for the state (State s f sp pc rs m) to step to state (State s f sp pc' rs m), i.e., the instruction at program point pc must be of the form Inop pc'. In the case of exec_Iop, it is necessary that the instruction at program point pc is of the form Iop op args res pc' and that evaluating the operation op using the values of registers args results in some value v which is then stored into register res.

We will come back to the RTL intermediate language in Chapter 4 as it is the host of the program transformations that we use to make sure that some properties are satisfied. The next subsection is devoted to a second tool upon which our work builds upon, namely the Verasco static analyzer.

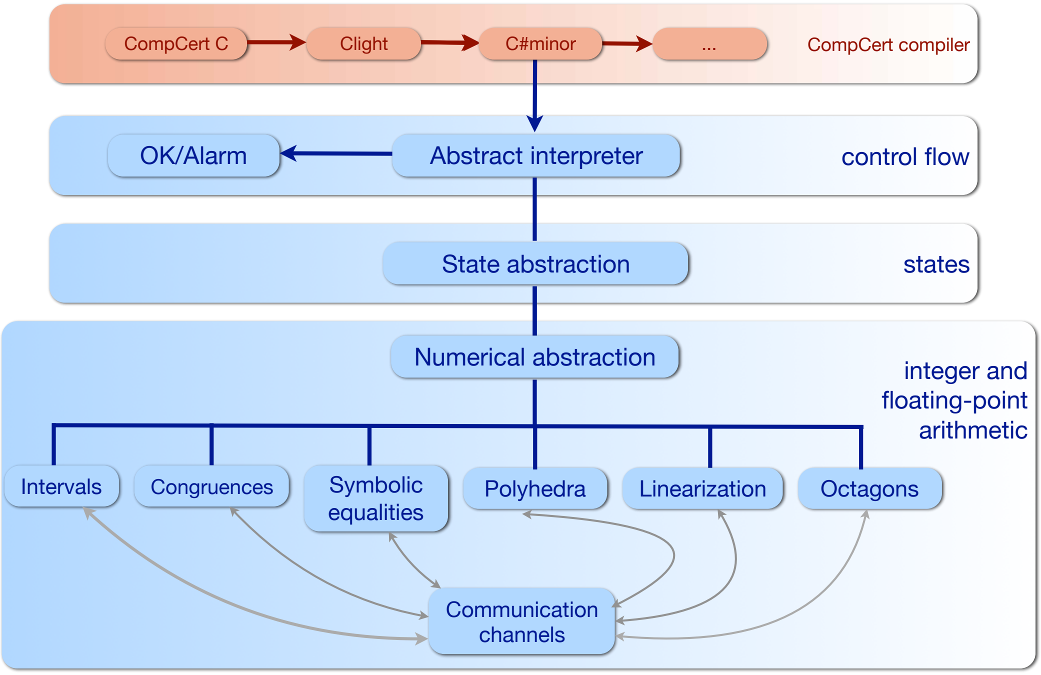

Figure 2.7: Architecture of the Verasco static analyzer

2.1.3 Verasco

Verasco is a static analyzer based on abstract interpretation that is formally verified in Coq [Jou+15] and builds upon CompCert. Its proof of correctness ensures that if the analyzer does not raise any warning, then the analyzed program is *safe*, it will execute without any runtime error such as out-of-bound array accesses, null pointer dereference, division by zero, etc. We present here general information about the Verasco static analyzer, it is extensively described in the two theses [Lap15; Jou16] devoted to its development.

The global design of the Verasco static analyzer is described in Figure 2.7. At the top of the figure, the frontend of CompCert is used up to the C#minor intermediate language where the analyzer is plugged in. Using an intermediate language of CompCert has multiple advantages, the first one is that there is no need to redesign a formal semantics for the input language and this makes it possible to combine the formal guarantees of Verasco and those of CompCert: any C#minor program that passes the analysis without raising alarm compiles into assembly code free from runtime error. The second advantage of analyzing C#minor programs instead of directly analyzing C programs is that the language is more prone to analysis, expressions are side-effects free, evaluation order is specified, operators are type-dependent instead of being overloaded, etc.

The next component is the abstract interpreter which iterates over the C#minor code and infers abstract states at every program point to check for runtime errors. The abstract states are computed a state abstraction that concretizes to memory states. The abstraction includes a points-to domain to precisely handle pointers, and some specialized domain to handle allocation and deallocation of memory. The state abstraction is parameterized by a numerical abstract domain capable of inferring numerical invariants on the program. It is separated into multiple abstract domains with each of them handling different properties. For instance, the properties can be relational using the *polyhedral* or *octagonal* domain, i.e., the property can relate multiple variables, $x \leq y$ for example. All these domains finely analyze the behavior of machine integers and floating-points (with potential overflows) while unsound analyzers would assume ideal arithmetic. They are connected all-together via *communication channels* that allow each domain to improve its own precision via specific queries to other domains. As a consequence, Verasco is able to infer subtle numerical invariants that require complex reasoning about linear arithmetic, congruence and symbolic equalities.

The design of Verasco is modular and inspired from Astrée [Bla+03], a milestone analyzer that was able to successfully analyze realistic safety-critical software systems for avionics and space flights. Modularity ensures that removing any of the domain and replacing them with domains that satisfy the same signature does not threaten the soundness of the analyzer, only its precision.

2.2 Verification of Security Properties

We present in this section different related works. First, as constant-time security can be seen as a form of non-interference, we will provide some general information on non-interference. Second, we present a technique used for verifying non-interference, namely *tainting*. Finally, we introduce some works in the field of high assurance software and more specifically high assurance cryptography which is a domain that really started to democratize itself slightly earlier than this thesis started.

2.2.1 Non-Interference

Non-interference is a baseline security property formalizing the non-dependence of public outputs on confidential inputs. In their seminal paper, Goguen and Meseguer [GM82] propose a security property that ensures that “one group of users, using a certain set of commands, [...] has no effect on what the second group of users can see”.

In [VIS96], Volpano et al. presents a type system to verify a variant of non-interference. Under their security policy, a program is secure if for any two terminating executions, the public outputs are the same when they differ only on confidential inputs. This definition is known as termination-insensitive non-interference (TINI) as its ignores information leaks due to the observation of termination or divergence of the program.

In a subsequent paper, Volpano and Smith [VS97] refines the previous type system to take termination into account, i.e., they verify termination-sensitive non-interference (TSNI).

One of the limitations of the type system proposed in [VIS96] was that it was flow-insensitive, i.e., the security level of a variable could not change between different program points. This limitation prevented many non-interferent programs to be accepted by their type system. Hunt and Sands [HS06] improves on the type system presented by Volpano et al. by modfying it in order to be flow-sensitive and thus making it more permissive, i.e., the number of accepted non-interferent programs is larger.

2.2.2 Tainting

Tainting, also known as taint tracking, is a popular method to track direct data dependencies. The idea is that in order to track which variables depend on some other chosen variables, it is sufficient to initially taint these chosen variables and taint each variable which definition depends on already tainted variables; the taint is "propogating" from tainted variables to the one they affect. How taints are precisely propagated is defined by the taint policy.

Tainting only concerns itself with tracking *explicit* flows of the form `l = h` where the value of `h` is explicitly leaked into variable `h`, but ignores *implicit* flows of the form `if (h) {l = 1;} else {l = 0;}` where the value of `h` is leaked into `l` by using the control-flow of the program. This makes taint tracking obviously unsound in some cases but makes it highly practical. Indeed, tracking control-flow is challenging, but the lightweight approach of tainting has made it popular as evidenced by its usage in languages such as Ruby[4] and Perl[5]. It is also the most popular approach for static analysis of Android applications [Li+17].

Tainting is used in many analyses to verify security policies. One popular use is to ensure that user inputs do not affect critical parts of the code, this is *integrity*. Another popular usage is *confidentiality*, i.e., to verify that applications do not leak users' private information, as illustrated by the many tools available for the Android mobile operating system [Arz+14; Enc+10; Sch+16].

Taint tracking can be static or dynamic, each with its advantages and drawbacks. Dynamic taint checking obviously slows down execution compared to a static approach, however, the latter approach may be less precise than the former one. [SAB10] provides a formalization of dynamic taint analysis and a survey of different tainting policies and what security policy they entail.

[4] http://phrogz.net/programmingruby/taint.html

[5] http://perldoc.perl.org/perlsec.html

2.2.3 High Assurance Cryptography

Constant-time security is part of the larger field of *high-assurance cryptography* [Bar15] which is a fertile area that has spawned many recent projects. There are two broad categories of methods of ensuring high assurance: either it is formally verified using a proof assistant such as Coq or F* [Swa+11], or it is verified using automatic tools such as Boogie [Lei08]. Each method has its own drawbacks: the former usually needs a highly experimented user to be accomplished while the latter is automatic but needs to trust in unverified and non-trivial tools such as SMT solvers.

[Bar+14] presents the first formally verified automated analyzer for constant-time security. It is formally verified in Coq and is based on the CompCert compiler. It operates on the Mach intermediate language of CompCert which allowed it to provide enough trust that the code that *actually* runs is effectively constant-time. However, it suffered from many limitations that are detailed in Chapter 4.

ct-verif [Alm+16] is another tool for verifying constant-time security. It operates on LLVM bytecode and its verification is based on a reduction of constant-time security of a program to the safety of a program product that simulates two parallel execution of the original program. The verification is made by the Boogie tool which generates verification conditions that are passed to SMT solvers. The tool is automatic but suffers from limitations due to the use of SMT solvers as they do not handle memory separation well, i.e., they usually consider a whole array as a single cell instead of multiple separate cells.

Vale [Bon+17] is a tool for producing verified cryptographic assembly code. Users write code in the Vale language which is similar to assembly, and then add a functional specification of the code in Dafny [Lei10], an automatic program verifier. The tool then automatically verifies that the code complies with the specification by using SMT solvers such as Z3 [DB08]. The authors also implemented a verified analyzer to ensure absence of timing and cache based side channels. However, they also seem to have limitations due to memory separation issues.

HACL* [Zin+17] is a formally verified C cryptographic library. Similarly to [Bon+17], the library was created by first writing cryptographic code in the proof assistant F* [Swa+11]. The code is then verified for functional correctness, memory safety and freedom of timing side channels. However, unlike [Bon+17], proofs are not automatic and must be manually written by an experimented user. While the produced C code is claimed to be constant-time, there remains the issue of preserving this security policy through compilation. It should be noted that their generated C code is now used in Firefox[6] which forms a strong statement for the democratization of formal verification.

Jasmin [Alm+17] is a formally-verified compiler from the Jasmin language down to assembly. The Jasmin language is a small low-level language similar to Bernstein's qhasm[Ber05b] that also supports function calls and high-level control-flow constructs

[6] `https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/`

such as loops. The authors have implemented a sound embedding of Jasmin into Dafny and users can thus automatically prove memory safety and constant-time security of their Jasmin programs using SMT solvers. Constant-time security is proven using product programs similarly to the ct-verif tool [Alm+16]. However, they do not mention if they suffer from the same memory separation issues.

Fiat-crypto [Erb+19] is a formally verified compiler specifically optimized for generating efficient elliptic-curve C code used in cryptography. However, the proven properties are only concerned with functional correctness. The compilation results in straightline code and they thus do not have to worry about secret dependent branching, but only about secret dependent memory accesses. The resulting code is thus not entirely proven constant-time. It should also be noted that their implementations of Curve25519 and P-256 have been integrated into the BoringSSL cryptography library which supports the Google Chrome internet browser, this forms a second strong statement in favor of formal verification.

In a series of publications [App15; Ber+15; Ye+17], the authors leverage the Verified Software Toolchain [App11] and CompCert to prove the functional correctness of a C implementation of SHA-256 and an implementation of HMAC with SHA-256, as well as functional correctness and cryptographic security of an implementation of HMAC-DBRG. However, they do not prove anything about side-channels resistance.

FaCT [Cau+17] proposes a domain-specific language to replace C as it is very prone to errors that can enable side channels. Their DSL can be basically seen as C enhanced with new annotations for expressing security levels such as which inputs can be considered secret or public. The language contains also new instructions that directly map to useful hardware instructions such as add-with-carry that are rarely produced by general purpose compilers. They use the Z3 SMT solver to prove memory safety of code written in this new language. Furthermore, as secret and public annotations are built in the language, they can adjust the compiler in order to take advantage of constant-time aware optimizations. Finally, as the tool is built upon LLVM, they can use the ct-verif tool [Alm+16] to verify that the generated code is secure, but must thus suffer the same limitations.

CHAPTER 3

VERIFICATION AT THE C LEVEL

As reviewing code written following the constant-time programming discipline is quite an arcane task, it is critical that programmers can be assured that they did not make any mistake. We argue that they need tool assistance to help them, first, to verify that the code they write is actually constant-time, and second, to assist them in understanding why it is not if they made an error. In this chapter, we present a static analysis at the source level so that reported errors by the tool can be better understood by coders. The static analysis is based on abstract interpretation methodology and advanced techniques such as context-sensitive (different invocations of a same function are distinguished) analyses, and powerful alias analyses technique that can distinguish between the different cells of an array.

Unfortunately, it is uncertain whether security properties that are ensured at source level also translate to lower levels. Indeed, compiler optimizations may break the constant-time transformations performed by the programmer. This issue will be tackled in the following chapter by introducing a static analysis at the assembly level and by Chapter 5 which presents a methodology to prove that a compiler preserves constant-time security.

This chapter is divided as follows, we first present the syntax and semantics of a small imperative While language in Section 3.1, we then formally define what it means for a program in the While language to be constant-time in Section 3.2 and show that proving a program secure can be reduced to proving it safe in an instrumented semantics in Section 3.3. Finally, we present the abstract interpreter for the While language and its correctness in Sections 3.4 and 3.5. We also have made a prototype by modifying Verasco and present an experimental evaluation of our static analyzer in Section 3.6.

A short version of this chapter has been published at the 22$^{\text{nd}}$ European Symposium on Research in Computer Security (ESORICS) in [BPT17], an extended version has been accepted for publication in the Journal of Computer Security [BPT18]. The companion development is available at `http://www.irisa.fr/celtique/ext/esorics17/`.

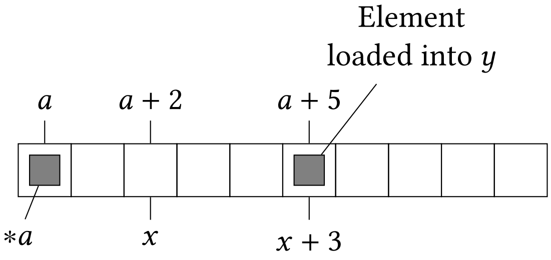

Figure 3.1: Example of aliasing

Expressions: $e ::= n \mid a \mid x \mid e_1 \oplus e_2 \quad \oplus \in \{+, -, \times, /, =, <, >\}$

Statements: $p ::= \texttt{skip} \mid *e_1 \leftarrow e_2 \mid x \leftarrow e \mid x \leftarrow *e \mid p_1 ; p_2$
$\mid \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \mid \texttt{while}\ e\ \texttt{do}\ p$

Figure 3.2: Syntax of While programs

3.1 The While Language

We present a small imperative While language, it is classically structured in statements and expressions, as shown in Figure 3.2. Expressions include integer constants, array identifiers, variable identifiers, arithmetic operations and tests. Statements include skip statements, stores $*x \leftarrow y$, loads $x \leftarrow *y$, assignments $x \leftarrow y$, sequences, if and while statements.

Our While language is peculiar as it supports arrays and pointers in order to model memory aliasing. We will mainly use a for array identifiers and x for variable identifiers. As an example, consider the program $x \leftarrow a + 2; y \leftarrow *(x + 3)$. a is an array and also corresponds to the address of its first cell which contains a value that can be accessed by $*a$. The program first starts by assigning the value $a + 2$ (pointer to the third cell of the array) to variable x and then loads the value at offset 5 of the array a into the variable y. In this example, $x - 2$ is an alias of a since x is an alias of $a + 2$. This example is illustrated in Figure 3.1.

Similarly to the semantics defined in CompCert, the semantics of While is defined in Figure 3.3 using a small-step style for statements and a big-step style for expressions, supporting the reasoning on non-terminating programs. Contrary to the C language, the semantics is deterministic (and so is the semantics of C#minor, the intermediate language Verasco operates over).

A memory location or location, usually named l, is a pair of an array identifier and an offset represented by a positive integer. A value v can either be a location or an integer. An environment σ is a pair $(\sigma_{\mathbb{X}}, \sigma_{\mathbb{A}})$ composed of a partial map from variables in set

$\mathbb{X}$ of variable identifiers to values and a partial map from memory locations $\mathbb{A} \times \mathbb{N}$ to values where $\mathbb{A}$ is a set of array identifiers and values $\mathbb{V}$ are either locations or integers. We will write $\sigma(x)$ to mean $\sigma_{\mathbb{X}}(x)$ and $\sigma(l)$ for $\sigma_{\mathbb{A}}(l)$.

Given an environment σ, an expression e evaluates to a value v, written $\langle \sigma, e \rangle \longrightarrow v$. A constant is interpreted as an integer. An array identifier a evaluates to its location, it is equivalent to writing $a + 0$. To evaluate a variable, its value is looked up in the environment σ and more precisely in its $\sigma_{\mathbb{X}}$ component. Finally, to evaluate $e_1 \oplus e_2$, it is simply needed to evaluate e_1 and e_2 separately and combine the resulting values by interpreting $\oplus$ into its corresponding operator $\boxplus$, where $\boxplus$ is the usual semantics of the operator $\oplus \in \{+, -, \times, /, =, <, >\}$. For example, $e_1 = e_2$ returns 0 if the test is false, and 1 otherwise.

The execution of a statement s results in an updated state with a new environment σ' and a new statement to execute s', written $\langle \sigma, s \rangle \longrightarrow \langle \sigma', s' \rangle$. We write $\sigma(e)$ to denote the value of expression e in state σ (i.e., $\langle \sigma, e \rangle \longrightarrow \sigma(e)$). We write $\sigma(x)$ to denote the value of variable x in environment σ and $\sigma(a, n)$ for the value of the cell of array a at offset n. We also use $\sigma[l \mapsto v]$ to be the environment σ where the location l has been updated to value v. We consider all arrays to be of finite size and initially declared, similarly to global variables in C. Thus, $\sigma(a, n)$ and $\sigma[(a, n) \mapsto v]$ may fail either because a is not a valid array name (i.e., $a \notin \mathbb{A}$) or because it is an out-of-bound access. $\sigma(l) = v$ means that l is a valid location for σ, whereas $\sigma(l) = \bot$ means the opposite. Similarly, $\sigma[l \mapsto v] = \sigma'$ indicates the success of the update. We assume a memory model similar to CompCert's except that addresses of variables cannot be taken and the variables behave more like registers.

To execute a store $*e_1 \longleftarrow e_2$, it is first needed for e_1 to evaluate into a location l and e_2 to evaluate into a value v; the environment is then updated so that location l maps to v. Similarly, to execute a load $x \longleftarrow *e$, the expression e must first evaluate into a location l. It is then needed to retrieve its corresponding value v in the environment and update the environment so that x maps to v. To execute the assignment $x \longleftarrow e$, it is only needed to evaluate e and update the environment so that x maps to the resulting value. To execute a sequence $p_1; p_2$, either p_1 is a skip and p_2 is the only statement left to execute, or we first need to execute p_1, resulting in a new state $\langle \sigma', p_1' \rangle$. Then, $p_1'; p_2$ is left to execute in the new environment σ'. Classically, in order to execute a conditional branching `if` e `then` p_1 `else` p_2, it is needed to evaluate e and execute accordingly the appropriate branch. We take `false` to be zero and `true` to be non-zero. Similarly, a loop `while` e `do` p stops if e evaluates to false and continues otherwise.

The evaluation of an expression can only be stuck in two ways, either because it is trying to retrieve the value of an undefined variable (i.e., $\sigma(x)$ fails when x is not defined in σ), or because $v_1 \boxplus v_2$ is not defined (e.g., because of a division by 0). Finally, the execution of statements can only be stuck when the semantic rule evaluates an expression and gets stuck, or the corresponding result has the wrong value type (i.e.,

$$l \in \mathbb{L} = \mathbb{A} \times \mathbb{N} \quad v \in \mathbb{V} = \mathbb{L} + \mathbb{Z}$$

$$\sigma = (\sigma_{\mathbb{X}}, \sigma_{\mathbb{A}}) \in \mathbb{M} = (\mathbb{X} \rightarrow \mathbb{V} \cup \{\bot\}) \times (\mathbb{L} \rightarrow \mathbb{V} \cup \{\bot\})$$

$$\frac{}{\langle \sigma, n \rangle \longrightarrow n}$$

$$\frac{}{\langle \sigma, a \rangle \longrightarrow (a, 0)}$$

$$\frac{\sigma(x) = v}{\langle \sigma, x \rangle \longrightarrow v}$$

$$\frac{\langle \sigma, e_1 \rangle \longrightarrow v_1 \qquad \langle \sigma, e_2 \rangle \longrightarrow v_2}{\langle \sigma, e_1 \oplus e_2 \rangle \longrightarrow v_1 \boxplus v_2}$$

$$\text{store}\ \frac{\langle \sigma, e_1 \rangle \longrightarrow l \qquad \langle \sigma, e_2 \rangle \longrightarrow v \qquad \sigma[l \mapsto v] = \sigma'}{\langle \sigma, *e_1 \longleftarrow e_2 \rangle \longrightarrow \langle \sigma', \mathtt{skip} \rangle}$$

$$\text{load}\ \frac{\langle \sigma, e \rangle \longrightarrow l \qquad \sigma(l) = v \qquad \sigma[x \mapsto v] = \sigma'}{\langle \sigma, x \longleftarrow *e \rangle \longrightarrow \langle \sigma', \mathtt{skip} \rangle}$$

$$\text{assign}\ \frac{\langle \sigma, e \rangle \longrightarrow v \qquad \sigma[x \mapsto v] = \sigma'}{\langle \sigma, x \longleftarrow e \rangle \longrightarrow \langle \sigma', \mathtt{skip} \rangle}$$

$$\text{skipseq}\ \frac{}{\langle \sigma, \mathtt{skip};p \rangle \longrightarrow \langle \sigma, p \rangle}$$

$$\text{seq}\ \frac{\langle \sigma, p_1 \rangle \longrightarrow \langle \sigma', p_1' \rangle}{\langle \sigma, p_1;p_2 \rangle \longrightarrow \langle \sigma', p_1';p_2 \rangle}$$

$$\text{iftrue}\ \frac{\langle \sigma, e \rangle \longrightarrow \mathtt{true}}{\langle \sigma, \mathtt{if}\ e\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2 \rangle \longrightarrow \langle \sigma, p_1 \rangle}$$

$$\text{iffalse}\ \frac{\langle \sigma, e \rangle \longrightarrow \mathtt{false}}{\langle \sigma, \mathtt{if}\ e\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2 \rangle \longrightarrow \langle \sigma, p_2 \rangle}$$

$$\text{whiletrue}\ \frac{\langle \sigma, e \rangle \longrightarrow \mathtt{true}}{\langle \sigma, \mathtt{while}\ e\ \mathtt{do}\ p \rangle \longrightarrow \langle \sigma, p;\mathtt{while}\ e\ \mathtt{do}\ p \rangle}$$

$$\text{whilefalse}\ \frac{\langle \sigma, e \rangle \longrightarrow \mathtt{false}}{\langle \sigma, \mathtt{while}\ e\ \mathtt{do}\ p \rangle \longrightarrow \langle \sigma, \mathtt{skip} \rangle}$$

Figure 3.3: Semantics of While programs

is a location when an integer was expected, or vice versa), or the result is a non-valid location (e.g., the location is out of bound). For instance, a branching statement cannot branch on a location value, or $\sigma(l)$ fails because it is an out-of-bound access or there is no associated value yet in the environment. This will be useful to prove Theorem 3.1.

The reflexive transitive closure of this small-step semantics represents the execution of a program. When the program terminates (resp. diverges, e.g. when an infinite loop is executed), it is a finite (resp. infinite) execution of steps. The execution of a program is *safe* iff either the program terminates (i.e., its final semantic state is $\langle\sigma, \texttt{skip}\rangle$, meaning that there is no more statement to execute) or the program diverges. The execution of a program is *stuck* on $\langle\sigma, s\rangle$ when s differs from `skip` and no semantic rule can be applied. A program is *safe* when all of its executions are safe. We write $(\langle\sigma_i, p_i\rangle)_i$ the execution $\langle\sigma_0, p_0\rangle \longrightarrow \langle\sigma_1, p_1\rangle \longrightarrow \ldots$ of program p_0 with initial environment σ_0.

3.2 Constant-Time Security

We now formally define what it means for a program in the While language to be constant-time. Informally, we said that a program is constant-time if none of its branching instructions nor its memory accesses depend on secret information. In order to model this, we use a definition similar to the one in [Alm+16] and also similar to the standard definition of non-interference.

Given a type of observations O, we define a *leakage model* $\mathscr{L}$ as a map from semantic states $\langle\sigma, p\rangle$ to sequences of observations (or leakages) $\mathscr{L}(\langle\sigma, p\rangle) \in O$ with ε being the empty observation.

Definition 3.1 (Constant-time leakage model)**.** Our leakage model is such that the following equalities hold.

1. $\mathscr{L}(\langle\sigma, \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2\rangle) = \sigma(e)$
2. $\mathscr{L}(\langle\sigma, \texttt{while}\ e\ \texttt{do}\ p\rangle) = \sigma(e)$
3. $\mathscr{L}(\langle\sigma, *e_1 \leftarrow e_2\rangle) = \sigma(e_1)$
4. $\mathscr{L}(\langle\sigma, x \leftarrow *e\rangle) = \sigma(e)$
5. $\mathscr{L}(\langle\sigma, p_1; p_2\rangle) = \mathscr{L}(\langle\sigma, p_1\rangle)$
6. $\mathscr{L}(\langle\sigma, p\rangle) = \varepsilon$ otherwise

The first and second lines mean that the value of branching conditions is considered as leaked. The third and fourth lines mean that the address of store and load accesses are also considered as leaked. The fifth line explains that a sequence leaks exactly what is leaked by the first part of the sequence; this is due to the semantics of sequence which depends on the execution of the first statement. As we use a small-step semantics, when executing $p_1;p_2$, we only execute p_1 until an execution step is done, p_2 is not affected. Finally, the other statements produce a silent observation. We also define the leakage of an execution as the concatenation of the leakage of all its states, i.e., $\mathscr{L}((\langle\sigma_i,p_i\rangle)_i) = \mathscr{L}(\langle\sigma_0,p_0\rangle)\cdot\mathscr{L}(\langle\sigma_1,p_1\rangle)\cdot\ldots$.

Given this leakage model, if executing a statement in two different environments leads to the same leak, then the next statements to execute are the same as illustrated by the following lemma.

Lemma 3.1 (Same control-flow). If $(\langle\sigma_1,p_1\rangle) \rightarrow (\langle\sigma_2,p_2\rangle)$ and $\left(\langle\sigma_1',p_1'\rangle\right) \rightarrow \left(\langle\sigma_2',p_2'\rangle\right)$ such that $p_1 = p_1'$ and $\mathscr{L}(\langle\sigma_1,p_1\rangle) = \mathscr{L}(\langle\sigma_1',p_1'\rangle)$ then $p_2 = p_2'$.

Proof. By induction on $(\langle\sigma_1,p_1\rangle) \rightarrow (\langle\sigma_2,p_2\rangle)$:

- In the assign, store and load cases, $p_2 = p_2' = \texttt{skip}$.
- In the skipseq case, there exists p such that $p_1 = p_1' = \texttt{skip};p$ and thus $p_2 = p_2' = p$.
- In the seq case, there exists q_1, q_1', q_1'' and q_2 such that $p_1 = p_1' = q_1;q_2$ and $\langle\sigma_1,q_1\rangle \rightarrow \langle\sigma_2,q_1'\rangle$ and $\langle\sigma_1',q_1\rangle \rightarrow \langle\sigma_2',q_1''\rangle$. In order to use the induction hypothesis to prove $q_1' = q_1''$, we first need to prove that $\mathscr{L}(\langle\sigma_1,q_1\rangle) = \mathscr{L}(\langle\sigma_1',q_1\rangle)$. This is true by definition since $\mathscr{L}(\langle\sigma_1,p_1\rangle) = \mathscr{L}(\langle\sigma_1,q_1;q_2\rangle) = \mathscr{L}(\langle\sigma_1,q_1\rangle)$ and also $\mathscr{L}(\langle\sigma_1',p_1'\rangle) = \mathscr{L}(\langle\sigma_1',q_1;q_2\rangle) = \mathscr{L}(\langle\sigma_1',q_1\rangle)$ and $\mathscr{L}(\langle\sigma_1,p_1\rangle) = \mathscr{L}(\langle\sigma_1',p_1'\rangle)$. Thus, since $p_2 = q_1';q_2$ and $p_2' = q_1'';q_2$, we have finally $p_2 = p_2'$.
- In the iftrue, iffalse, whiletrue and whilefalse cases, we simply use $\mathscr{L}(\langle\sigma_1,p_1\rangle) = \mathscr{L}(\langle\sigma_1',p_1'\rangle)$ to justify that the same branch is taken.

□

We now define what it means for two executions to be *indistinguishable.*

Definition 3.2 (Indistinguishable executions). Two executions $(\langle\sigma_i,p_i\rangle)_i$ and $\left(\langle\sigma_i',p_i'\rangle\right)_i$ are said to be *indistinguishable* when their observations are the same:

$$\mathscr{L}(\langle\sigma_0,p_0\rangle)\cdot\mathscr{L}(\langle\sigma_1,p_1\rangle)\cdot\ldots = \mathscr{L}(\langle\sigma_0',p_0'\rangle)\cdot\mathscr{L}(\langle\sigma_1',p_1'\rangle)\cdot\ldots.$$

Finally, the following theorem generalizes the previous lemma to indistinguishable executions.

Theorem 3.1. Two indistinguishable executions of a program necessarily have the same control flow.

Proof. Suppose we have two indistinguishable executions $(\langle\sigma_i,p_i\rangle)_i$ and $\left(\langle\sigma_i',p_i'\rangle\right)_i$. We know that $p_0 = p_0'$ since we consider indistinguishable executions of a same program. We prove by induction on i that for all i, $p_i = p_i'$.

- It's true by hypothesis for $i = 0$.
- Suppose that $p_i = p'_i$.
  - If the execution is stuck for $\langle \sigma_i, p_i \rangle$, then, as explained earlier, it is because p_i tries to write or read an invalid location (i.e., the value is not a location but a constant or it is an out-of-bound location) or it tries to branch on a non-integer value (i.e., a location). However, by definition of indistinguishability and the leakage model, these values must be the same in both executions, thus the execution is also stuck for $\langle \sigma'_i, p'_i \rangle$.
  - Symmetrically, if the execution is stuck for $\langle \sigma'_i, p'_i \rangle$, it is also stuck for $\langle \sigma_i, p_i \rangle$.
  - The two previous cases show that if one execution is stuck, then so is the other. Thus if $\langle \sigma_i, p_i \rangle \longrightarrow \langle \sigma_{i+1}, p_{i+1} \rangle$, then there must exist σ'_{i+1}, p'_{i+1} such that $\langle \sigma'_i, p'_i \rangle \longrightarrow \langle \sigma'_{i+1}, p'_{i+1} \rangle$. By using the previous lemma, we prove that $p_{i+1} = p'_{i+1}$.

Both executions have thus the same control flow. □

Given a program, we assume that the attacker has access to the values of some of its inputs, which we call the *public* input variables, and does not have access to the other ones, which we call the *secret* input variables. Given a set X of identifiers, and two environments σ and σ', we say that σ and σ' are X-equivalent if σ and σ' both share the same public input values. Two executions $(\langle \sigma_i, p_i \rangle)_i$ and $\left(\langle \sigma'_i, p'_i \rangle\right)_i$ are initially X-equivalent if σ_0 and σ'_0 are X-equivalent and $p_0 = p'_0$.

Definition 3.3 (Constant-time security)**.** A program p is constant time with regards to set X_i of public input variables, if all of its initially X_i-equivalent executions are indistinguishable.

This definition means that a constant-time program is such that, any pair of its executions that only differ on its secrets must leak the exact same information, i.e., secrets do not influence leaks. This definition corresponds to the informal definition given at the beginning of Section 3.2 that branching instructions and memory accesses shall not depend on secret information.

3.3 Reducing Security to Safety

In order to prove that a program satisfy constant-time security as defined in Definition 3.3, we reduce the problem to checking whether the program is safe in a different semantics. The issue is thus twofold, we first need to prove that safety in this instrumented semantics implies constant-time security in the standard semantics and second, we need to design an analyzer for this second semantics. This can also be obtained by

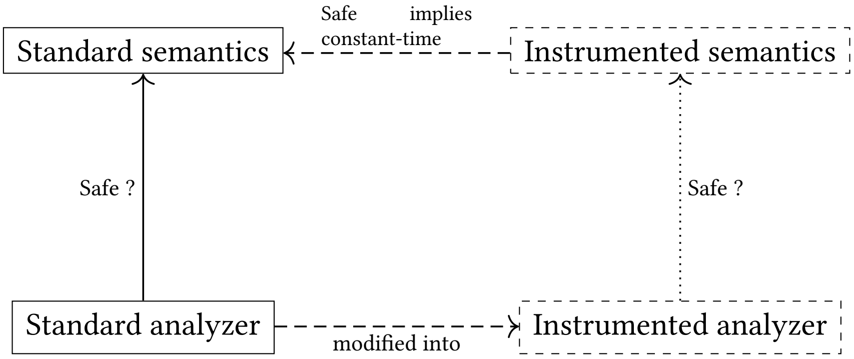

Figure 3.4: Methodology

modifying an analyzer for the standard semantics as illustrated by Figure 3.4. Plain lines indicate what we assume to already have, while dashed lines indicates what need to be designed or proved.

In this section, we will present an instrumented semantics (later named tainting semantics) and show that safety in this semantics implies constant-time security in the standard semantics. The instrumented analyzer is presented in Section 3.4 and its correctness in Section 3.5.

A high level view of our methodology can be found in Figure 3.5 which summarizes the relationships between the different semantics and the theorems that links them. The general idea is that our analyzer tries to establish whether a program is safe in a so called tainting semantics. We prove that the analyzer is indeed correct, i.e., if it decides that a program is safe, then it is actually safe. The proof is done using standard abstract interpretation techniques based on a collecting semantics. Furthermore, we prove that a program safe in the tainting semantics satisfies constant-time security in the standard semantics, effectively making our analyzer an analyzer for constant-time security.

We introduce an intermediate tainting semantics for While programs in Figure 3.6, and use the $\rightsquigarrow$ symbol to distinguish its executions from those of the original semantics. The tainting semantics is an instrumentation of the While semantics that tracks dependencies related to secret values. In the tainted semantics, a program gets stuck if branchings or memory accesses depend on secrets. We introduce taints, either $\mathcal{H}$ (High security) or $\mathcal{L}$ (Low security) to respectively represent secret and public values and a union operator on taints defined as follows: $\mathcal{L} \sqcup \mathcal{L} = \mathcal{L}$ and for all t, $\mathcal{H} \sqcup t = t \sqcup \mathcal{H} = \mathcal{H}$. It is used to compute the taint of a binary expression, if any of its subexpression has a High taint, then the whole expression also has a High taint. The purpose of the taints is to track whether the leakages may depend on secrets (High taint) or are benign (Low). In the instrumented semantics, we take into account taints in semantic values: the semantic state σ becomes a tainted state $\overline{\sigma}$, where locations are now mapped to pairs

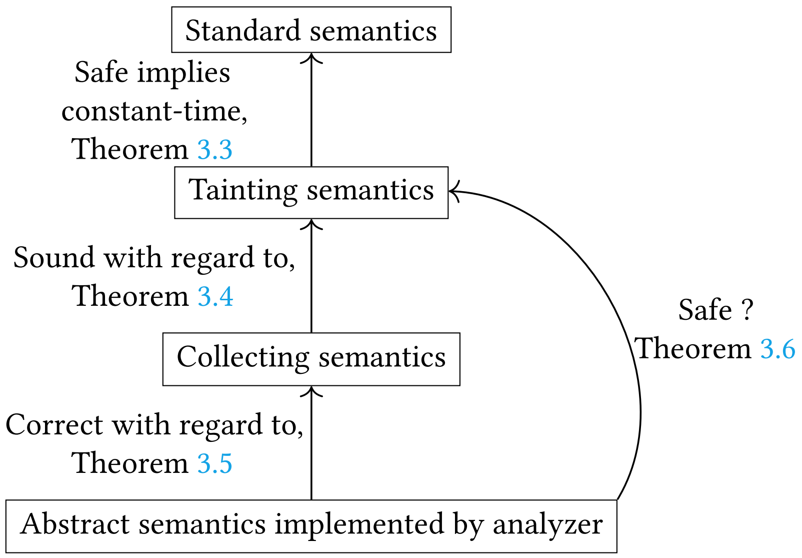
Standard semantics
Safe implies constant-time, Theorem 3.3
Tainting semantics
Sound with regard to, Theorem 3.4
Collecting semantics
Correct with regard to, Theorem 3.5
Abstract semantics implemented by analyzer
Safe ? Theorem 3.6

Figure 3.5: Diagram relating the different semantics

made of a value and a taint.

Let us note that for the dereferencing of an expression $*e$ to not fail, the taint associated to e must be $\mathcal{L}$. Indeed, we forbid memory read accesses that might leak secret values. This concerns dereferencing expressions (loads) and assignment statements. Similarly, test conditions in branching statements must also have a $\mathcal{L}$ taint.

The instrumented semantics strictly forbids more behaviors than the standard semantics (defined in Figure 3.3) as shown by the following lemma.

Lemma 3.2. Any execution $(\langle \overline{\sigma_i}, p_i \rangle)_i$ of program p_0 in the tainting semantics implies that $(\langle \sigma_i, p_i \rangle)_i$ is an execution of p_0 in the standard semantics where $\mathcal{E}(a, b) = a$ for all pairs (a, b) is an erasure function and for all i, $\sigma_i = \mathcal{E} \circ \overline{\sigma_i}$.

Proof. For all $\overline{\sigma}$, $\overline{\sigma}'$, p, p' such that $\langle \overline{\sigma}, p \rangle \rightarrow \langle \overline{\sigma}', p' \rangle$, we can prove by immediate induction that $\langle \sigma, p \rangle \rightarrow \langle \sigma', p' \rangle$ where $\sigma = \mathcal{E} \circ \overline{\sigma}$ and $\sigma' = \mathcal{E} \circ \overline{\sigma}'$.

Finally, by induction on the execution and using this lemma, the theorem is easily proven. □

However, the converse is not necessarily true. For instance, suppose that variable x contains a secret value. Then, $*(a + x) \leftarrow 2$ is not safe in the instrumented semantics because $a + x$ has taint $\mathcal{H}$, while it is safe in the standard semantics provided that $a + x$ corresponds to a valid location.

An immediate consequence of the lemma is that the instrumented semantics preserves the safe behavior of programs, as stated by the following theorem.

$$t \in \mathbb{T} = \{\mathcal{L}, \mathcal{H}\}$$

$$\overline{\mathbb{V}} = \mathbb{V} \times \mathbb{T}$$

$$\overline{\sigma} = (\overline{\sigma_{\mathbb{X}}}, \overline{\sigma_{\mathbb{A}}}) \in \overline{\mathbb{M}} = (\mathbb{X} \to \overline{\mathbb{V}} \cup \{\bot\}) \times (\mathbb{L} \to \overline{\mathbb{V}} \cup \{\bot\})$$

$$\frac{}{\langle \overline{\sigma}, n \rangle \rightsquigarrow (n, \mathcal{L})} \qquad \frac{}{\langle \overline{\sigma}, a \rangle \rightsquigarrow ((a, 0), \mathcal{L})}$$

$$\frac{\overline{\sigma}(x) = (v, t)}{\langle \overline{\sigma}, x \rangle \rightsquigarrow (v, t)} \qquad \frac{\langle \overline{\sigma}, e_1 \rangle \rightsquigarrow (v_1, t_1) \qquad \langle \overline{\sigma}, e_2 \rangle \rightsquigarrow (v_2, t_2)}{\langle \overline{\sigma}, e_1 \oplus e_2 \rangle \rightsquigarrow (v_1 \boxplus v_2, t_1 \sqcup t_2)}$$

$$\frac{\langle \overline{\sigma}, e_1 \rangle \rightsquigarrow (l, \mathcal{L}) \qquad \langle \overline{\sigma}, e_2 \rangle \rightsquigarrow (v, t) \qquad \overline{\sigma}[l \mapsto (v, t)] = \sigma'}{\langle \overline{\sigma}, *e_1 \leftarrow e_2 \rangle \rightsquigarrow \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (l, \mathcal{L}) \qquad \overline{\sigma}(l) \rightsquigarrow (v, t) \qquad \overline{\sigma}[x \mapsto (v, t)] = \sigma'}{\langle \overline{\sigma}, x \leftarrow *e \rangle \rightsquigarrow \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \to (v, t) \qquad \overline{\sigma}[x \mapsto (v, t)] = \overline{\sigma}'}{\langle \overline{\sigma}, x \leftarrow e \rangle \to \langle \overline{\sigma}', \texttt{skip} \rangle}$$

$$\frac{}{\langle \overline{\sigma}, \texttt{skip}; p \rangle \rightsquigarrow \langle \overline{\sigma}, p \rangle} \qquad \frac{\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow \langle \overline{\sigma}', p_1' \rangle}{\langle \overline{\sigma}, p_1 ; p_2 \rangle \rightsquigarrow \langle \overline{\sigma}', p_1' ; p_2 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{true}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \rangle \rightsquigarrow \langle \overline{\sigma}, p_1 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{false}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \rangle \rightsquigarrow \langle \overline{\sigma}, p_2 \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{true}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{while}\ e\ \texttt{do}\ p \rangle \rightsquigarrow \langle \overline{\sigma}, p; \texttt{while}\ e\ \texttt{do}\ p \rangle}$$

$$\frac{\langle \overline{\sigma}, e \rangle \rightsquigarrow (\texttt{false}, \mathcal{L})}{\langle \overline{\sigma}, \texttt{while}\ e\ \texttt{do}\ p \rangle \rightsquigarrow \langle \overline{\sigma}, \texttt{skip} \rangle}$$

Figure 3.6: Tainting semantics for While programs

Theorem 3.2. Any safe execution $(\langle\overline{\sigma_i}, p_i\rangle)_i$ of program p_0 in the tainting semantics implies that the execution $(\langle\sigma_i, p_i\rangle)_i$ is also safe in the standard semantics.

As an immediate corollary, any safe program according to the tainting semantics is also safe according to the standard semantics.

Proof. Let $(\langle\overline{\sigma_i}, p_i\rangle)_i$ be a safe execution of p_0 in the tainting semantics. As it is a safe execution, it can either diverge or terminate.

- If $(\langle\overline{\sigma_i}, p_i\rangle)_i$ is diverging (i.e. infinite), then so is $(\langle\sigma_i, p_i\rangle)_i$ thanks to the previous lemma.

- If $(\langle\overline{\sigma_i}, p_i\rangle)_i$ is terminating, then there exists some n such that $p_n = \texttt{skip}$, therefore $(\langle\sigma_i, p_i\rangle)_{i\leq n}$ is also terminating.

$(\langle\sigma_i, p_i\rangle)_i$ is a safe execution in the standard semantics. □

Theorem 3.2 is useful to prove our main theorem relating our instrumented semantics and the constant-time property we want to verify on programs. Given a set X_i of public input variables, a program is constant-time with regards to X_i if any of its executions such that the variables in X_i are given an initial low taint, is safe in the tainting semantics. The intuition is that from an execution in the standard semantics, we can build a corresponding one in the tainting semantics that is guaranteed to be safe by hypothesis which implies that the initial execution is also safe thanks to Theorem 3.2. Finally, as the execution in the tainting semantics cannot leak information with a high taint, we can conclude that the leakage in the execution in the standard semantics does not depend on secret.

Theorem 3.3. Let X_i be a set of public variables. If any execution $(\langle\overline{\sigma_i}, p_i\rangle)_i$ of program p_0 in the tainting semantics, such that for all $x \in X_i$, $\overline{\sigma_0}(x)$ has a low taint is safe, then p_0 is constant-time with regards to X_i.

Proof. Let $(\langle\sigma_i, p_i\rangle)_i$ and $\left(\langle\sigma'_i, p'_i\rangle\right)_i$ be two safe executions of p_0 that are initially X_i-equivalent.

We now need to prove that both executions are indistinguishable. Let $\overline{\sigma_0}$ be such that for all $x \in X_i,\ n \in \mathbb{N},\ \overline{\sigma_0}(x, n) = (\sigma_0(x, n), \mathcal{L})$ and also for all $x \notin X_i,\ n \in \mathbb{N},\ \overline{\sigma_0}(x, n) = (\sigma_0(x, n), \mathcal{H})$.

By safety of program p_0 according to the tainting semantics, there exists some states $\overline{\sigma_1}, \overline{\sigma_2}, \ldots$ such that $\langle\overline{\sigma_0}, p_0\rangle \rightsquigarrow \langle\overline{\sigma_1}, p_1\rangle \rightsquigarrow \ldots$ is a safe execution. Let $\sigma_{n'} = \mathcal{E} \circ \overline{\sigma_n}$, we prove by strong induction on n that $\sigma_{n'} = \sigma_n$.

- It is clearly true for $n = 0$ by definition of $\overline{\sigma_0}$.

- Suppose it is true for all $k < n$ and let us prove it for n. By using theorem 3.2, we know that there exists a safe execution $\langle\sigma_0, p_0\rangle \rightarrow \langle\sigma_{1'}, p_{1'}\rangle \rightarrow \ldots \rightarrow \langle\sigma_{n'}, p_{n'}\rangle \rightarrow \ldots$. Furthermore, the standard semantics is deterministic and we know that $\langle\sigma_0, p_0\rangle \rightarrow \langle\sigma_1, p_1\rangle \rightarrow \ldots$. Therefore, we have the following series of equalities: $\sigma_{1'} = \sigma_1, p_{1'} = p_1, \ldots \sigma_{n'} = \sigma_n, p_{n'} = p_n$.

Thus, for all $k \in \mathbb{N}$, the state $\overline{\sigma_k}$ verifies $\sigma_k = \mathcal{E} \circ \overline{\sigma_k}$. Similarly, we define $\overline{\sigma_0}', \overline{\sigma_1}', \ldots$ for the second execution which also verifies the same property by construction.

Finally, we need to prove that for all $n \in \mathbb{N}$, $L(\langle\sigma_n, p_n\rangle) = L(\langle\sigma'_n, p'_n\rangle)$.

First, we informally define the notation $\sigma_n =_{\mathcal{L}} \sigma'_n$ for all $n \in \mathbb{N}$ as $\overline{\sigma}_n$ and $\overline{\sigma}'_n$, as previously defined, agree on the taints of both variables and locations, and if the taint is $\mathcal{L}$, then they also agree on the value. Formally, this means that for all r where r is either a location l or a variable x, either $\overline{\sigma}_n(r)$ and $\overline{\sigma}'_n(r)$ are undefined, or there exists a taint t such that $\overline{\sigma}_n(r) = (\sigma_n(r), t)$ and $\overline{\sigma}'_n(r) = (\sigma'_n(r), t)$ and if $t = \mathcal{L}$, then $\sigma_n(r) = \sigma'_n(r)$. Second, we introduce the following lemma.

Lemma 3.3. For all n and e such that if $\sigma_n =_{\mathcal{L}} \sigma'_n$, $\langle\overline{\sigma_n}, e\rangle \rightsquigarrow (v, t)$ and $\langle\overline{\sigma'_n}, e\rangle \rightsquigarrow (v', t')$, then $t = t'$ and if $t = t' = \mathcal{L}$, then $v = v'$.

This is proven by induction on e.

- This is trivially true if $e = n$ or $e = a$.

- If $e = x$, then it is true by definition of $\sigma_n =_{\mathcal{L}} \sigma'_n$

- If $e = e_1 \oplus e_2$, then we apply the induction hypotheses on $\langle\overline{\sigma_n}, e_1\rangle \rightsquigarrow (v_1, t_1)$ and $\langle\overline{\sigma_n}', e_1\rangle \rightsquigarrow (v'_1, t'_1)$ and on $\langle\overline{\sigma_n}, e_2\rangle \rightsquigarrow (v_2, t_2)$ and $\langle\overline{\sigma_n}', e_2\rangle \rightsquigarrow (v'_2, t'_2)$. Since $t = t_1 \sqcup t_2$ and $t' = t'_1 \sqcup t'_2$ and $t_1 = t'_1$ and $t_2 = t'_2$, we have that $t = t'$. If $t = t' = \mathcal{L}$, then $t_1 = t'_1 = \mathcal{L}$ and $t_2 = t'_2 = \mathcal{L}$, thus $v = v_1 \boxplus v_2 = v'_1 \boxplus v'_2 = v'$.

This lemma is thus proven.

Finally, for all $n \in \mathbb{N}$, let us prove by induction on p_n that if $p_n = p'_n$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$, then $p_{n+1} = p'_{n+1}$ and $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$.

- If $p_n = \texttt{skip;}p'$, it is true because $p_{n+1} = p'_{n+1} = p'$, $\sigma_{n+1} = \sigma_n$ and also $\sigma'_{n+1} = \sigma'_n$.

- If $p_n = p;\ p'$, it is true by induction hypothesis.

- If $p_n = \texttt{if}\ e \ldots$ or $p_n = \texttt{while}\ e \ldots$, we have $\sigma_{n+1} = \sigma_n$ and $\sigma'_{n+1} = \sigma'_n$. Furthermore, we know that there exists some v such that $\langle\overline{\sigma_n}, e\rangle \rightsquigarrow (v, \mathcal{L})$ and similarly, there exists v' such that $\langle\overline{\sigma_n}', e\rangle \rightsquigarrow (v', \mathcal{L})$ because of the safety in the tainting semantics. Since $\sigma_n(e) = v$, $\sigma'_n(e) = v'$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$, we have $v = v'$ by using the previous lemma 3.3 and thus $p_{n+1} = p'_{n+1}$.

- If $p_n = x \leftarrow *e$, we can prove as previously that $\sigma_n(e_1) = \sigma'_n(e_1) = l$. Furthermore, we have $p_{n+1} = p'_{n+1} = \texttt{skip}$. It is left to prove that $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. If $\overline{\sigma_n}(l) = (v, t)$ and $\overline{\sigma_n}'(l) = (v', t')$, then $t = t'$ since $\sigma_n =_{\mathcal{L}} \sigma'_n$. If $t = t' = \mathcal{L}$, then $v = v'$ and $\sigma_{n+1} = \sigma_n[x \mapsto v]$ and $\sigma'_{n+1} = \sigma'_n[x \mapsto v']$, thus $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. Similarly, if $t = t' = \mathcal{H}$, then $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$.

- If $p_n = x \leftarrow e$, we know that $p_{n+1} = p'_{n+1} = \texttt{skip}$. Furthermore, there exists v, v', t, t' such that $\langle \overline{\sigma_n}, e \rangle \rightsquigarrow (v, t)$ and $\langle \overline{\sigma_n}', e \rangle \rightsquigarrow (v', t')$. By using the previous lemma, we know that $t = t'$, and if $t = t' = \mathcal{L}$, then $v = v'$. Thus $\sigma_{n+1} = \sigma_n[x \mapsto v] =_{\mathcal{L}} \sigma'_n[x \mapsto v'] = \sigma'_{n+1}$.

- If $p_n = e_1 \leftarrow e_2$, we have $p_{n+1} = p'_{n+1} = \texttt{skip}$. By using the same reasoning as previously, we can prove that $\sigma_n(e_1) = \sigma'_n(e_1) = l$. There exists v, v', t, t' such that $\langle \overline{\sigma_n}, e_2 \rangle \rightsquigarrow (v, t)$ and $\langle \overline{\sigma_n}', e_2 \rangle \rightsquigarrow (v', t')$ and thus $\sigma_{n+1} = \sigma_n[l \mapsto v]$ and $\sigma'_{n+1}[l \mapsto v']$. By using the previous lemma, we know that $t = t'$ and if $t = t' = \mathcal{L}$, then $v = v'$ and $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$. If $t = t' = \mathcal{H}$, then $\sigma_{n+1} =_{\mathcal{L}} \sigma'_{n+1}$ by definition.

Finally, by exploiting this second lemma, an induction proves that for all $n \in \mathbb{N}$, $p_n = p'_n$ and $\sigma_n =_{\mathcal{L}} \sigma'_n$. Furthermore, a direct consequence is that for all $n \in \mathbb{N}$, $L(\langle \sigma_n, p_n \rangle) = L(\langle \sigma'_n, p'_n \rangle)$ and thus both executions are indistinguishable: the program is constant time. □

The theorem is thus proven, but what about its converse, is a constant-time program necessarily safe with regards to the tainting semantics? This is however not true, indeed, consider `if (secret - secret) { ... } else { ... }`. This program is constant-time since the value of the conditional guard does not depend on secrets, it always evaluate to 0. However, it is not safe with regards to our tainting semantics, as `(secret - secret)` is considered to have a high taint.

We have shown that a program safe with regards to the tainting semantics is constant-time, we will now see how to prove that a program is safe according to this semantics.

3.4 Abstract Interpreter

To prove that a program is safe according to the tainting semantics, we design a static analyzer based on abstract interpretation. It computes a correct approximation of the execution of the analyzed program, thus if the approximative execution is safe, then the actual execution must necessarily be safe.

Similarly to how we built a tainting semantics from a standard semantics, we explain how to modify an abstract interpreter for the standard semantics into an abstract interpreter for the tainting semantics. First, we suppose that the regular abstract interpreter has the same structure as the one illustrated in Figure 3.7. It provides a

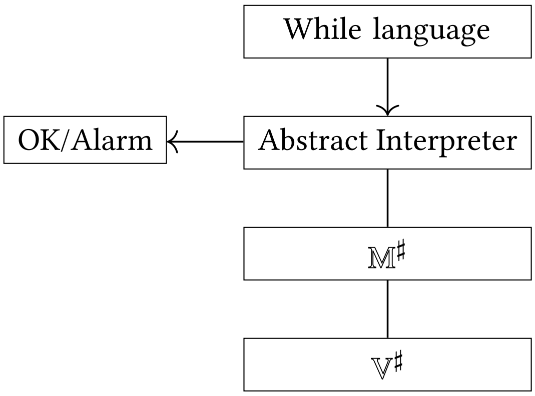

Figure 3.7: Structure of an abstract interpreter

domain of abstract values $\mathbb{V}^\sharp$ that supports an operator $\text{concretize}^\sharp : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{V})$ which takes an abstract value and returns the concrete values represented by the abstract value. We also suppose that the abstract interpreter provides $\mathbb{M}^\sharp$, an abstraction of concrete environments built upon $\mathbb{V}^\sharp$ that maps locations and variables to values. We do not need nor want to know exactly how $\mathbb{M}^\sharp$ is defined, as it might use relational definitions which are quite complex. We only need to use $\mathbb{M}^\sharp$ to modify the abstract analyzer.

Finally, we suppose that the abstract analyzer provides the following abstract operators:

- $\text{eval}^\sharp : \mathbb{M}^\sharp \to \text{expr} \to \mathbb{V}^\sharp$ takes an abstract environment, an expression and evaluates it in the abstract environment and returns the corresponding abstract value;
- $\text{assign}^\sharp : \mathbb{M}^\sharp \to \mathbb{X} \to \text{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, a variable identifier, an expression and models an assignment to a variable;
- $\text{store}^\sharp : \mathbb{M}^\sharp \to \text{expr} \to \text{expr} \to \mathbb{M}^\sharp$ takes an abstract environment and two expressions e_1 and e_2 and models $*e_1 \leftarrow e_2$;
- $\text{load}^\sharp : \mathbb{M}^\sharp \to \mathbb{X} \to \text{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, a variable identifier, an expression and models a load $x \leftarrow *e$;
- $\text{assert}^\sharp : \mathbb{M}^\sharp \to \text{expr} \to \mathbb{M}^\sharp$ takes an abstract environment, an expression and returns an abstract environment where the expression is true. This is useful when analyzing a branching condition such as $x < 5$, if we know beforehand that $x \in [0, 42]$, we can restrict x to $[0, 4]$ in the “then” branch, and restrict it to $[5, 42]$ in the “else” branch.

$\mathcal{H}^\sharp$
$\mathcal{L}^\sharp$

Figure 3.8: Abstract taint lattice $\mathbb{T}^\sharp$

The abstract operators form an interface that is parameterized by $\mathbb{V}^\sharp$ and $\mathbb{M}^\sharp$ that we will name $\mathsf{AbMem}(\mathbb{V}^\sharp, \mathbb{M}^\sharp)$.

Now, in order for the analyzer to handle the tainting semantics, we need to introduce an abstraction of taints $\mathbb{T}^\sharp = \{\mathcal{L}^\sharp, \mathcal{H}^\sharp\}$ which forms a lattice represented in Figure 3.8. We will use $\mathcal{L}^\sharp$ to indicate a value that has exactly taint $\mathcal{L}$ while $\mathcal{H}^\sharp$ indicates that a value may have taint $\mathcal{L}$ or $\mathcal{H}$. In order to analyze the following snippet, it is necessary to correctly approximate the taint of the value that will be assigned to variable x after execution.

```
if /* low expr */
  x ⟵ /* high expr */
else
  x ⟵ /* low expr */
```

As it can either be $\mathcal{L}$ or $\mathcal{H}$, we use the approximation $\mathcal{H}^\sharp$. We could have used $\mathcal{H}^\sharp$ to indicate that a variable or location can only have a $\mathcal{H}$ value, however constant-time security is not interested in knowing that value has exactly $\mathcal{H}$ taint, but only in knowing that it *may* have a $\mathcal{H}$ taint. Similarly to $\sqcup$, we define $\sqcup^\sharp$ as $\mathcal{L}^\sharp \sqcup^\sharp \mathcal{L}^\sharp = \mathcal{L}$ and for all $t^\sharp$, $\mathcal{H}^\sharp \sqcup t^\sharp = t^\sharp \sqcup \mathcal{H}^\sharp = \mathcal{H}^\sharp$.

Now, we explain how to modify the analyzer so that it can track abstract taints, this process effectively forms a functor from the previous interface $\mathsf{AbMem}(\mathbb{V}^\sharp, \mathbb{M}^\sharp)$ to a new interface $\mathsf{AbMem}(\overline{\mathbb{V}}^\sharp, \overline{\mathbb{M}}^\sharp)$ that can track abstract taints where $\overline{\mathbb{V}}^\sharp = \mathbb{V}^\sharp \times \mathbb{T}^\sharp$ and $\overline{\mathbb{M}}^\sharp = \mathbb{M}^\sharp \times ((\mathbb{X} + \mathbb{L}) \to \mathbb{T}^\sharp)$.

We first start by defining $\overline{\mathsf{taint}}^\sharp : \overline{\mathbb{M}}^\sharp \longrightarrow \mathsf{expr} \longrightarrow \mathbb{T}^\sharp + \bot$ which returns the abstract taint corresponding to the evaluation of an expression. We use $\mathcal{T}(a, b) = b$ as tainting function, the companion of the erasure function $\mathcal{E}$.

$$\begin{aligned}
\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, n) &= \mathcal{L}^\sharp \\
\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, a) &= \mathcal{L}^\sharp \\
\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, x) &= \mathcal{T}(\overline{\sigma}^\sharp)(x) \\
\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_1 \oplus e_2) &= \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_1) \sqcup^\sharp \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_2)
\end{aligned}$$

We now define the following abstract operators (i.e., transfer functions).

- $\overline{\mathsf{eval}}^\sharp(\overline{\sigma}^\sharp, e) = (\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e), \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e))$
- $\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e) = (\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e), \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e)])$
- $\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e) = (\mathsf{assert}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e), \mathcal{T}(\overline{\sigma}^\sharp))$
- $\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2) =$
  $(\mathsf{store}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1, e_2), \mathcal{T}(\overline{\sigma}^\sharp)[l \mapsto \mathcal{T}(\overline{\sigma}^\sharp)(l) \sqcup^\sharp \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e_2)]_{l \in \mathsf{concretize}^\sharp(\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1))})$
- $\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e) = (\mathsf{load}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e), \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \bigsqcup^\sharp_{l \in \mathsf{concretize}^\sharp(\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e))} \mathcal{T}(\overline{\sigma}^\sharp)(l)])$

The definitions of $\overline{\mathsf{eval}}^\sharp$, $\overline{\mathsf{assign}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$ reuse the operators of $\mathsf{AbMem}(\overline{\mathbb{V}}^\sharp, \overline{\mathbb{M}}^\sharp)$ and modify slightly the tainting part. The definitions of $\overline{\mathsf{store}}^\sharp$ and $\overline{\mathsf{load}}^\sharp$ are more complex. In both cases, we need to use $\mathsf{eval}^\sharp$ to deduce all possible locations affected by the memory accesses and suitably update the tainting parts. For $\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2)$, all possible write locations l given by the concretization of $\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e_1)$ are updated with the union of the taint of the value contained in l and the taint given by e_2. This is due to the fact that the analysis does not know precisely where the write happens and must thus be conservative. However, if the analysis managed to pinpoint an unique location, it would be possible to use a strong update instead of a weak one. As for $\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e)$, we approximate the taints from all possible read locations given by the concretization of $\mathsf{eval}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), e)$. This concludes the definition of $\mathsf{AbMem}(\overline{\mathbb{V}}^\sharp, \overline{\mathbb{M}}^\sharp)$.

Finally, the abstract analysis $[\![p]\!](\sigma^\sharp, \tau^\sharp)$ of program p starting with tainted abstract environment $\overline{\sigma}^\sharp$ is defined in Figure 3.9. To analyze $(p_1; p_2)$, first p_1 is analyzed and then p_2 is analyzed using the environment given by the first analysis. Similarly, to analyze a statement (`if` e `then` p_1 `else` p_2), p_1 is analyzed assuming that e is true and p_2 is analyzed assuming the opposite, $\sqcup^\sharp$ is then used to get an over-approximation of both results.

$$\begin{aligned}
[\![\texttt{skip}]\!]^\sharp(\overline{\sigma}^\sharp) &= \overline{\sigma}^\sharp \\
[\![*e_1 \leftarrow e_2]\!]^\sharp(\overline{\sigma}^\sharp) &= \overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2) \\
[\![x \leftarrow *e]\!]^\sharp(\overline{\sigma}^\sharp) &= \overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e) \\
[\![x \leftarrow e]\!]^\sharp(\overline{\sigma}^\sharp) &= \overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e) \\
[\![p_1; p_2]\!]^\sharp(\overline{\sigma}^\sharp) &= [\![p_2]\!]^\sharp([\![p_1]\!]^\sharp(\overline{\sigma}^\sharp)) \\
[\![\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2]\!]^\sharp(\overline{\sigma}^\sharp) &= [\![p_1]\!]^\sharp(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e)) \sqcup^\sharp \\
&\ [\![p_2]\!]^\sharp(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, \mathsf{not}\ e)) \\
[\![\texttt{while}\ e\ \texttt{do}\ p]\!]^\sharp(\overline{\sigma}_0^\sharp) &= \overline{\mathsf{assert}}^\sharp(\mathsf{pfp}(\mathsf{iter}(e, p, \overline{\sigma}_0^\sharp, \cdot)), \mathsf{not}\ e) \\
\mathsf{iter}(e, p, \overline{\sigma}_0^\sharp, \overline{\sigma}^\sharp) &= \overline{\sigma}_0^\sharp \sqcup^\sharp \overline{\mathsf{assert}}^\sharp([\![p]\!]^\sharp(\overline{\sigma}^\sharp), e)
\end{aligned}$$

Figure 3.9: Abstract execution of statements

The loop ($\texttt{while}\ e\ \texttt{do}\ p$) is the trickiest part to analyze, as the analysis cannot just analyze one iteration of the loop body and then recursively analyze the loop again since this may never terminate. The analysis thus tries to find a loop invariant. The standard method in abstract interpretation is to compute a post-fixpoint of the function $\mathsf{iter}(e, p, \overline{\sigma}_0^\sharp, \cdot)$ as defined in Figure 3.9. It represents a loop invariant, the final result is thus the invariant where the test condition does not hold anymore. In order to compute the post-fixpoint, we use $\mathsf{pfp}(f)$ which computes a post-fixpoint of monotone function f by successively computing $\bot, f(\bot), f(f(\bot)), \ldots$, and forces convergence using a widening-narrowing operator [CC76] on the $\mathbb{M}^\sharp$ part. The taint part does not require convergence help because taints form a finite lattice.

3.5 Correctness of the Abstract Interpreter

In order to specify and prove the correctness of the analyzer, we follow the usual methodology in abstract interpretation and define a *collecting* semantics, aiming at facilitating the proof. The semantics still expresses the dynamic behavior of programs but takes a closer form to the analysis. It operates over properties of concrete environments, thus bridging the gap between concrete environments and abstract environments, which represent sets of concrete environments.

The collecting semantics aims at describing the resulting environments that can be reached given a specific instruction and a set of environments. The collecting semantics of a program p with a set of concrete environments Σ is written $[\![p]\!](\Sigma)$.

$$
\begin{aligned}
[\![\texttt{skip}]\!](\Sigma) &= \Sigma \\
[\![*e_1 \leftarrow e_2]\!](\Sigma) &= \overline{\mathsf{Store}}(\Sigma, e_1, e_2) \\
[\![x \leftarrow *e]\!](\Sigma) &= \overline{\mathsf{Load}}(\Sigma, x, e) \\
[\![x \leftarrow e]\!](\Sigma) &= \overline{\mathsf{Assign}}(\Sigma, x, e) \\
[\![p_1 ; p_2]\!](\Sigma) &= [\![p_2]\!]([\![p_1]\!](\Sigma)) \\
[\![\texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2]\!](\Sigma) &= [\![p_1]\!](\overline{\mathsf{Assert}}(\Sigma, e)) \cup [\![p_2]\!](\overline{\mathsf{Assert}}(\Sigma, \mathsf{not}\ e)) \\
[\![\texttt{while}\ e\ \texttt{do}\ p]\!](\Sigma) &= \overline{\mathsf{Assert}}(I, \mathsf{not}\ e)
\end{aligned}
$$

where I is the least fixpoint of the equation $I == \Sigma \cup [\![p]\!](\mathsf{Assert}(I, e))$

Figure 3.10: Definition of the collecting semantics $[\![\cdot]\!](\cdot)$

Similarly to the abstract interpreter, we define $\overline{\mathsf{Assign}}$, $\overline{\mathsf{Store}}$, $\overline{\mathsf{Load}}$, $\overline{\mathsf{Assert}}$. They will respectively serve as counterparts to $\overline{\mathsf{assign}}^\sharp$, $\overline{\mathsf{store}}^\sharp$, $\overline{\mathsf{load}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$. We first start with $\overline{\mathsf{Assign}}$:

$$\overline{\mathsf{Assign}}(\Sigma, x, e) = \{\overline{\sigma}[x \mapsto (v, t)] | \exists v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e) = (v, t) \land \overline{\sigma} \in \Sigma\}$$

Given a set of concrete environments Σ, $\overline{\mathsf{Assign}}(\Sigma, x, e)$ computes the set of all possible reachable environments from environments in Σ after executing $x \leftarrow e$ in the tainting semantics.

Next are $\overline{\mathsf{Store}}$ and $\overline{\mathsf{Load}}$:

$$\overline{\mathsf{Store}}(\Sigma, e_1, e_2) = \{\overline{\sigma}[l \mapsto (v, t)] | \exists l \in \mathbb{L}, v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e_1) = (l, \mathcal{L}) \land \overline{\sigma}(e_2) = (v, t) \land \overline{\sigma} \in \Sigma\}$$
$$\overline{\mathsf{Load}}(\Sigma, x, e) = \{\overline{\sigma}[x \mapsto (v, t)] | \exists l \in \mathbb{L}, v \in \mathbb{V}, t \in \mathbb{T}, \overline{\sigma}(e) = (l, \mathcal{L}) \land \overline{\sigma}(l) = (v, t) \land \overline{\sigma} \in \Sigma\}$$

Given a set of concrete environments Σ, $\overline{\mathsf{Store}}(\Sigma, e_1, e_2)$ (resp. $\overline{\mathsf{Load}}(\Sigma, x, e)$) computes the set of all possible reachable environments from environments in Σ after executing $*e_1 \leftarrow e_2$ (resp. $x \leftarrow *e$) in the tainting semantics.

$\overline{\mathsf{Assert}}$ removes the environments where e is not true:

$$\overline{\mathsf{Assert}}(\Sigma, e) = \{\overline{\sigma} \in \Sigma | \exists t, \overline{\sigma}(e) = (\mathsf{true}, t)\}$$

Finally, the collecting semantics is defined in Figure 3.10. Looking at the rules in Figure 3.9 and Figure 3.10, one can notice that the collecting semantics follows closely the shape of the abstract interpreter. The collecting semantics of assignment is defined using $\overline{\mathsf{Assign}}$, the counterpart of $\overline{\mathsf{assign}}^\sharp$. Similarly to the abstract interpreter, to evaluate a conditional branching, the first branch is evaluated assuming the condition is true using Assert and the second branch is evaluated assuming the opposite. The results are then merged to obtain all the possible states that can be reached.

We first start by proving that the collecting semantics is sound with regards to the tainting semantics.

Theorem 3.4. For all program p and environment $\overline{\sigma}$, $\langle \overline{\sigma}, p \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle \implies \overline{\sigma}' \in [\![p]\!](\{\overline{\sigma}\})$.

Proof. This is a fairly standard proof in abstract interpretation. As the theorem statement does not directly fit well with induction, we first start by proving the following more general lemma:

$$\forall p, \overline{\sigma}, \overline{\sigma}', \Sigma, \overline{\sigma} \in \Sigma \implies \langle \overline{\sigma}, p \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle \implies \overline{\sigma}' \in [\![p]\!](\Sigma)$$

The proof is by induction on p.

- If $p = \texttt{skip}$, it is trivially true.

- If $p = *e_1 \leftarrow e_2$ or $p = x \leftarrow *e$ or $p = x \leftarrow e$, it is true by definition of $\overline{\text{Store}}$, $\overline{\text{Load}}$, $\overline{\text{Assign}}$ and by definition of the tainting semantics.

- If $p = p_1; p_2$, then there exists $\overline{\sigma}''$ such that $\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow^* \langle \overline{\sigma}'', \texttt{skip} \rangle$ and $\langle \overline{\sigma}'', p_2 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$. By induction hypothesis on the first execution, we obtain that $\overline{\sigma}'' \in [\![p_1]\!](\Sigma)$. Combining this with using the induction hypothesis on the second execution allows us to conclude that $\overline{\sigma}' \in [\![p_2]\!]([\![p_1]\!](\Sigma)) = [\![p_1; p_2]\!](\Sigma) = [\![p]\!](\Sigma)$.

- If $p = \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2$, then either $\overline{\sigma}(e) = \text{true}$ and $\langle \overline{\sigma}, p_1 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$ or $\overline{\sigma}(e) = \text{false}$ and $\langle \overline{\sigma}, p_2 \rangle \rightsquigarrow^* \langle \overline{\sigma}', \texttt{skip} \rangle$. In the first case, $\overline{\sigma} \in \overline{\text{Assert}}(\Sigma, e)$ and in the latter, $\overline{\sigma} \in \overline{\text{Assert}}(\Sigma, \text{not } e)$ which allows us to conclude in both cases by using the induction hypothesis.

- If $p = \texttt{while}\ e\ \texttt{do}\ p$, then we know that $\overline{\sigma}'(e) = \text{false}$. Furthermore, we remark that for all $\overline{\sigma}''$ such that $\langle \overline{\sigma}, \texttt{while}\ e\ \texttt{do}\ p \rangle \rightsquigarrow^* \langle \overline{\sigma}'', \texttt{while}\ e\ \texttt{do}\ p \rangle$, $\overline{\sigma}'' \in I$ by definition of I, the least fixpoint of the equation $I == \Sigma \cup [\![p]\!](\text{Assert}(I, e))$. Thus, $\overline{\sigma}' \in I$ and since $\overline{\sigma}'(e) = \text{false}$, $\overline{\sigma}' \in \overline{\text{Assert}}(I, \text{not } e)$.

The lemma is thus proven, and the theorem is a direct consequence of it. □

The standard semantics also has a collecting semantics with the operators Assign, Store, Load, Assert and a corresponding soundness theorem that we will not detail. The operators are defined as follows:

$$
\begin{aligned}
\mathsf{Assign}(\Sigma, x, e) &= \{\sigma[x \mapsto v] | \exists v \in \mathbb{V}, \sigma(e) = v \land \sigma \in \Sigma\} \\
\mathsf{Store}(\Sigma, e_1, e_2) &= \{\sigma[l \mapsto v] | \exists l \in \mathbb{L}, v \in \mathbb{V}, \sigma(e_1) = l \land \sigma(e_2) = v \land \overline{\sigma} \in \Sigma\} \\
\mathsf{Load}(\Sigma, x, e) &= \{\sigma[x \mapsto v] | \exists l \in \mathbb{L}, v \in \mathbb{V}, \sigma(e) = l \land \sigma(l) = v \land \sigma \in \Sigma\} \\
\mathsf{Assert}(\Sigma, e) &= \{\sigma \in \Sigma | \sigma(e) = \mathsf{true}\}
\end{aligned}
$$

Finally, we also need to introduce the concept of concretization to state and prove the correctness of our abstract interpreter. We already introduced $\mathsf{concretize}^\sharp$ previously which is actually a concretization function. We will rename it $\gamma_{\mathbb{V}^\sharp}$ as γ is the usual name for a concretization function in abstract interpretation. We use $v \in \gamma_{\mathbb{V}^\sharp}(v^\sharp)$ to say that v is in the concretization of abstract value $v^\sharp$, which means that $v^\sharp$ represents a set of concrete values of which v is a member.

The abstract memory domain $\mathbb{M}^\sharp$ also provides a concretization function $\gamma_{\mathbb{M}^\sharp} : \mathbb{M}^\sharp \longrightarrow \mathcal{P}(M)$ which is used to define the correctness of the $\mathsf{assign}^\sharp$, $\mathsf{store}^\sharp$, $\mathsf{load}^\sharp$ and $\mathsf{assert}^\sharp$ operators:

$$
\begin{aligned}
\mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), x, e) &\subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\sigma^\sharp, e)) \\
\mathsf{Store}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), e_1, e_2) &\subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{store}^\sharp(\sigma^\sharp, e_1, e_2)) \\
\mathsf{Load}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), x, e) &\subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{load}^\sharp(\sigma^\sharp, x, e)) \\
\mathsf{Assert}(\gamma_{\mathbb{M}^\sharp}(\sigma^\sharp), e) &\subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assert}^\sharp(\sigma^\sharp, e))
\end{aligned}
$$

We now need to define $\gamma_{\mathbb{T}^\sharp} : \mathbb{T}^\sharp \longrightarrow \mathcal{P}(\mathbb{T})$ and $\gamma_{\overline{\mathbb{M}}^\sharp} : \overline{\mathbb{M}}^\sharp \longrightarrow \mathcal{P}(\overline{M})$.

The first one is simple, $\gamma_{\mathbb{T}^\sharp}(\mathcal{L}^\#) = \{\mathcal{L}\}$ and $\gamma_{\mathbb{T}^\sharp}(\mathcal{H}^\#) = \{\mathcal{L}, \mathcal{H}\}$. $\mathcal{L}^\#$ corresponds to values that we know are *necessarily* public data, while $\mathcal{H}^\#$ corresponds to values that we only know *may* depend on secrets.

Now, we define $\gamma_{\overline{\mathbb{M}}^\sharp}$:

$$
\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp) = \{\overline{\sigma} | \mathcal{E} \circ \overline{\sigma} \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)) \land \forall r, \mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\sigma}^\sharp)(r))\}
$$

This means that an environment $\overline{\sigma}$ is in the concretization of $\overline{\sigma}^\sharp$ if there exists $\sigma \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp))$ such that $\mathcal{E} \circ \overline{\sigma} = \sigma$ and such that $\mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\sigma}^\sharp)(r))$ for all location or variable r.

We now need to prove the correctness of the $\overline{\mathsf{assign}}^\sharp$, $\overline{\mathsf{store}}^\sharp$, $\overline{\mathsf{load}}^\sharp$ and $\overline{\mathsf{assert}}^\sharp$ operators:

Lemma 3.4.

$$\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e))$$
$$\overline{\mathsf{Store}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), e_1, e_2) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{store}}^\sharp(\overline{\sigma}^\sharp, e_1, e_2))$$
$$\overline{\mathsf{Load}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{load}}^\sharp(\overline{\sigma}^\sharp, x, e))$$
$$\overline{\mathsf{Assert}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), e) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assert}}^\sharp(\overline{\sigma}^\sharp, e))$$

Proof. We need to prove that for all $\overline{\sigma} \in \overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)$, $\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, e))$. We first define $\mathcal{E}(\overline{\Sigma}) = \{\mathcal{E} \circ \overline{\sigma} | \overline{\sigma} \in \overline{\Sigma}\}$ for all $\overline{\Sigma} \in \mathcal{P}(\overline{M})$. We then notice that $\mathcal{E}(\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)) = \mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)), x, e))$ by definitions.

Then, by correctness of $\mathsf{assign}^\sharp$, we have that $\mathsf{Assign}(\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\sigma}^\sharp)), x, e)) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e))$. And by definition of $\overline{\mathsf{assign}}^\sharp$, we have that $\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)) = \mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e$. Thus, $\gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e))) = \gamma_{\mathbb{M}^\sharp}(\mathsf{assign}^\sharp(\mathcal{E}(\overline{\sigma}^\sharp), x, e))$ which implies that $\mathcal{E}(\overline{\mathsf{Assign}}(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp), x, e)) \subseteq \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)))$ and therefore, there exists $\sigma \in \gamma_{\mathbb{M}^\sharp}(\mathcal{E}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}, x, e)))$ such that $\mathcal{E}(\overline{\sigma}) = \sigma$.

It is then left to prove that for all r, $\mathcal{T}(\overline{\sigma}(r)) \in \gamma_{\mathbb{T}^\sharp}(\mathcal{T}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e))(r))$. By definition of $\overline{\mathsf{assign}}^\sharp$, $\mathcal{T}(\overline{\mathsf{assign}}^\sharp(\overline{\sigma}^\sharp, x, e)) = \mathcal{T}(\overline{\sigma}^\sharp)[x \mapsto \overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e)]$. By definition of $\overline{\mathsf{Assign}}$, we know that there exists $\overline{\sigma}_1 \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)$ such that $\overline{\sigma} = \overline{\sigma}_1[x \mapsto (v, t)]$ with $\overline{\sigma}_1(e) = (v, t)$.

The correctness of $\overline{\mathsf{taint}}^\sharp$ can easily be proven by induction on e:

$$\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp) \implies \mathcal{T}(\overline{\sigma}(e)) \in \gamma_{\mathbb{T}^\sharp}(\overline{\mathsf{taint}}^\sharp(\overline{\sigma}^\sharp, e))$$

By exploiting the lemma, the correctness of $\overline{\mathsf{assign}}^\sharp$ is thus proven. The correctness of the other operators is similarly proven. □

The following theorem which states the correctness of the abstract analyzer with regards to the collecting semantics can now be proven.

Theorem 3.5. For all abstract environment $\overline{\sigma}^\sharp$ and program p,

$$\llbracket p \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p \rrbracket^\sharp(\overline{\sigma}^\sharp))$$

Proof. We first remark that $\llbracket p \rrbracket$ is a monotone function, i.e. $\Sigma_1 \subseteq \Sigma_2 \implies \llbracket p \rrbracket(\Sigma_1) \subseteq \llbracket p \rrbracket(\Sigma_2)$. The proof is by induction on p. The theorem is also proven by induction on p. We have that:

- if $p = \mathtt{skip}$, it is trivially true;
- if $p = *e_1 \leftarrow e_2$ or $p = x \leftarrow e$ or $p = x \leftarrow *e$, it is a direct consequence of the correctness of the corresponding operators;
- if $p = p1; p2$, we have $\llbracket p_1 \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_1 \rrbracket^\sharp(\overline{\sigma}^\sharp))$ by induction hypothesis on p_1 and $\llbracket p_2 \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_1 \rrbracket^\sharp(\overline{\sigma}^\sharp))) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_2 \rrbracket^\sharp(\llbracket p_1 \rrbracket^\sharp(\overline{\sigma}^\sharp)))$ on p_2. And by monotony of $\llbracket p_2 \rrbracket$, we have $\llbracket p_1; p_2 \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)) = \llbracket p_2 \rrbracket(\llbracket p_1 \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp))) \subseteq \llbracket p_2 \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_1 \rrbracket^\sharp(\overline{\sigma}^\sharp))) \subseteq \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_2 \rrbracket^\sharp(\llbracket p_1 \rrbracket^\sharp(\overline{\sigma}^\sharp))) = \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p_1; p_2 \rrbracket^\sharp(\overline{\sigma}^\sharp))$ which is what we needed to prove;
- if $p = \mathtt{if}\ e\ \mathtt{then}\ p_1\ \mathtt{else}\ p_2$, it is a consequence of the correctness of $\overline{\mathrm{assert}}^\sharp$;
- if $p = \mathtt{while}\ e\ \mathtt{do}\ p$, it is a consequence of the correctness of pfp with regards to the invariant, and the correctness of $\overline{\mathrm{assert}}^\sharp$.

The theorem is thus proven. □

This theorem intuitively means that the abstract analyzer is correct with regards to the collecting semantics since if $\gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p \rrbracket^\sharp(\overline{\sigma}^\sharp))$ is empty, $\llbracket p \rrbracket(\gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp))$ must necessarily be empty too, and thus the execution is stuck with regards to the collecting semantics.

Finally, combining Theorems 3.4 and 3.5, the following correctness theorem is a direct consequence:

Theorem 3.6. For all program p, environment $\overline{\sigma}$ and abstract environment $\overline{\sigma}^\sharp$ such that $\overline{\sigma} \in \gamma_{\overline{\mathbb{M}}^\sharp}(\overline{\sigma}^\sharp)$, if we have the execution $\langle \overline{\sigma}, p \rangle \rightsquigarrow^* \langle \overline{\sigma}', \mathtt{skip} \rangle$, then we also have $\overline{\sigma}' \in \gamma_{\overline{\mathbb{M}}^\sharp}(\llbracket p \rrbracket^\sharp(\overline{\sigma}^\sharp))$.

This is the main theorem of correctness of the abstract interpreter. It ensures that we compute correct over-approximations of reachable states in the tainting semantics. We can then safely perform abstract tests on the program to check that no tainting state may reach a stuck configuration. By that, we mean that the analyzer may fail or raise alarms during the analysis. For instance, when analyzing `if (x)`, it may raise an alarm to say that `x` may potentially depend on a secret if at this program point, it knows that its taint is $\mathcal{H}^\sharp$. Hence, we can conclude that if no alarm is raised, then the program is safe with regard to the tainting semantics and is thus constant-time.

3.6 Implementation and Experiments

Following the methodology presented previously, we have implemented a prototype leveraging the Verasco static analyzer. It necessitated to add a taint layer to Verasco to track the taint associated with variables and memory locations. This layer reused information already computed by Verasco to obtain the necessary points-to information to properly taint memory locations. The analyzer has then been modified to query the taint layer when an `if` instruction or a memory access is encountered in order to verify that they are harmless.

We have been able to evaluate our prototype by verifying multiple actual C code constant-time algorithms taken from a set of representative cryptographic libraries such as NaCl [BLS12], mbedTLS [mbe14],`curve25519-donna` [Lan08] and Open Quantum Safe [Bos+15].

Many analyzers for verifying constant-time security exist, such as [RBV17; TIS16; Alm+16; Bar+14], and we will compare more specifically with [Alm+16], a state of the art analyzer operating on LLVM bytecode. This comparison was chosen as their tool ct-verif provides a similar level of guarantee as ours, but instead relying on the semantic framework of relational verification and product programs. The other tools were not chosen for comparison for different reasons. For instance, [RBV17] has a statistical approach in which the analyzed program is run multiple times with different inputs to *test* its constant-timeness. It thus lacks any guarantee on the answer it provides compared to an approach like ours. As for [Bar+14], while also based on CompCert, it operates at the assembly level and is thus crippled by lack of precision due to the simple difficulty of operating at this level. A comparison would be unfair with a tool operating at source level as ours. [TIS16] is the tool most similar to ours but is a commercial tool and cannot thus be freely tested.

In order to use our tool, the user simply has to indicate which variables are to be considered as secrets and the prototype will either raise alarms indicating where secrets may leak, or indicate that the input program is constant time. The user can either indicate a whole global variable to be considered as secret at the start of the program, or use the `verasco_any_int_secret` built-in function to produce a random signed integer to be considered as secret.

The While language we presented has a few differences with the C#minor language of CompCert that we analyze using Verasco. First, C#minor allows more constructs such as `switch` and does not use `while` loops, but infinite loops that must be exited using a `break` statement. Secondly, C#minor expressions can contain memory reads whereas our While language models a memory load as a statement. However, this is only a slight difference as C#minor programs such as $x = *y + *z$ are already transformed into $x1 = *y; x2 = *z; x = x1 + x2$ by Verasco in order to improve the precision of the analysis.

3.6.1 Context Sensitivity

An inherent advantage of our methodology is that context sensitivity is preserved. Indeed, by combining Verasco's points-to analysis with a taint analysis, we inherit

Verasco's ability of interprocedural analysis. We thus obtain an analysis that is more precise than if the taint analysis was solely a client of a points-to analysis.

For instance, consider the following program where secret values are copied into an array and are then replaced with public ones.

```c
int table[64];
int pub[64] = { ... }; // public values
int secret[64] = { ... }; // secret values

int* memcpy(int* dst, int* src, signed len) {
  for (signed i = 0; i < len; ++i) {
    dst[i] = src[i];
  }
  return src;
}

int main(void) {
  memcpy(table, secret, 64);
  memcpy(table, pub, 64);
  return table[0];
}
```

If the points-to analysis is run first, `dst[i]` would be annotated with `table[0..63]` while `src[i]` would be annotated with both `pub[0..63]` and `secret[0..63]`. A taint analysis leveraging the points-to analysis would then need to conclude that the returned value `table[0]` is tainted. However, our methodology combines both analyses and thus manages to conclude that the return value is untainted as secret values are replaced by public ones.

3.6.2 Memory Separation

By leveraging Verasco, the prototype has no problem handling difficult problems such as memory separation, i.e., the taint of each cell in an array is tracked instead of tainting the whole array with the same taint as most standard analyzers do. For example, the small example of Figure 3.11 is easily proven as constant time. In this program, an array `t` is initialized with random values, such that the values in odd offsets are considered as secrets, contrary to values in even offsets. So, the analyzer needs to be precise enough to distinguish between the array cells and to take into account pointer arithmetic. The potential leak happens on line 6. However, the condition on line 5 constrains `i%2 == 0` to be true, and thus `i` must be even on line 6, so `t[i]` does not contain a secret. A naive

```
int main(void) {
  int t[4] = { verasco_any_int(), verasco_any_int_secret(),
       verasco_any_int(), verasco_any_int_secret() };
  for (int i = 0; i < 4; i++)
     if (i%2 == 0) {  // First if condition
       if (t[i])  t[i] = 0; } // Second if condition
  return 0; }
```

Figure 3.11: An example program that is analyzed as constant time

analyzer would taint the whole array as secret and would thus not be able to prove the program constant-time, however our prototype has no problem to prove it.

Interestingly, an illustration of the problem can be found in real-world programs. For example, the NaCl implementation of SHA-256 is not handled by [Alm+16] due to this. Indeed, in this program, the hashing function uses the following C `struct` as an internal state that contains both secret and public values during execution.

The struct and the hashing function are defined in Figure 3.12

The function first starts by initializing the internal state with some constant value and then updates it using the input value `in` which is considered secret as it can be a password that an user is trying to hash. Both fields `state` and `buf` may contain secret dependent values as a result of the update. Last, `crypto_hash_sha256_final` contains a conditional branching that depends on the `count` field of the internal state: `if ((state->count[1] += bitlen[1]) < bitlen[1])`. However, the whole internal state `struct` is allocated as a single memory block at low level (i.e., LLVM) and [Alm+16] does not manage to prove the memory separation and cannot thus ensure that the hashing function is secure.

3.6.3 Cryptographic Algorithms

We report in Table 3.1 our results on a set of representative cryptographic algorithms. All executions times reported were obtained on a 3.1GHz Intel i7 with 16GB of RAM. Sizes are reported in terms of numbers of C#minor statements (i.e., close to C statements), lines of code are measured with `cloc` and execution times are reported in seconds.

The first block of lines gathers test cases for the implementations of a representative set of cryptographic primitives including TEA [WN95], an implementation of sampling in a discrete Gaussian distribution by Bos et al. [Bos+15] (`rlwe_sample`) taken from the Open Quantum Safe library [Saf16], an implementation of elliptic curve arithmetic operations over Curve25519 [Ber06] by Langley [Lan08](`curve25519-donna`), and various primitives such as AES, DES, etc. The second block reports on implementations

```
typedef struct crypto_hash_sha256_state {
    uint32_t      state[8];
    uint32_t      count[2];
    unsigned char buf[64];          } crypto_hash_sha256_state;

int crypto_hash(unsigned char *out, const unsigned char *in,
                unsigned long long inlen)
{
    crypto_hash_sha256_state state;

    crypto_hash_sha256_init(&state);
    crypto_hash_sha256_update(&state, in, inlen);
    crypto_hash_sha256_final(&state, out);
    return 0;
}
```

Figure 3.12: SHA256 Example

Example	Size	Loc	Time
aes	1171	1399	41.39
curve25519-donna	1210	608	586.20
des	229	436	2.28
rlwe_sample	145	1142	30.76
salsa20	341	652	5.34
sha3	531	251	57.62
snow	871	460	4.37
tea	121	109	3.47
bear_aes_ct	803	766	1.97
bear_des_ct	454	560	2.54
bear_sha1	243	197	2.45
bear_sha256	259	329	2.83
nacl_chacha20	384	307	0.34
nacl_sha256	368	287	1.85
mbedtls_sha1	544	354	0.33
mbedtls_sha256	346	346	0.62
mbedtls_sha512	310	399	0.58
mee-cbc	1959	939	933.37

Table 3.1: Verification of cryptographic primitives

from the BearSSL library [Por16]. The third block reports on different implementations from the NaCl library [BLS12]. The fourth block reports on implementations from the mbedTLS [mbe14] library. Finally, the last result corresponds to an implementation of MAC-then-Encode-then-CBC-Encrypt (MEE-CBC).

All these examples are proven constant time, except for AES and DES which both make use of look-up tables. Our prototype rightfully reports memory accesses depending on secrets, so these two programs are not constant time. Similarly to [Alm+16], `rlwe_sample` is only proven constant time assuming that the core random generator it uses is also constant time, thus showing that it is the only possible source of leakage.

The last example `mee-cbc` is a full implementation of the MEE-CBC construction using low-level primitives taken from the NaCl library. Our prototype is able to verify the constant-time property of this example, showing that it scales to large code bases (1399 loc).

Our prototype is able to verify a similar amount of programs than [Alm+16], except for a constant-time fixed point operations library named `libfixedtimefixedpoint` [And+15] which unfortunately does not use standard C and is not handled by CompCert. The library uses extensively a GNU extension known as statement-expressions and would require heavy rewriting to be accepted by our tool.

On the other hand, our tool shows its agility with memory separation on the program SHA-256 that was out of reach for [Alm+16] and its restricted alias management. In terms of analysis time, our tool behaves similarly to [Alm+16]. On a similar experiment platform, we observe a speedup between 0.1 and 10. This is very encouraging for our tool whose efficiency is still in an upgradeable stage, compared to the tool of [Alm+16] that relies on decades of implementation efforts for the LLVM optimizer and the Boogie verifier.

3.7 Conclusion

In this chapter, we presented a methodology to ensure that cryptography software implementations respect the constant-time security paradigm. The approach is first presented on a small While language and is then adapted to C by leveraging the Verasco static analyzer. It is based on the observation that verifying constant-time security of a program can be reduced into verifying the safety of the program in a specific semantics, namely a “tainting” semantics. This observation is then used with the support of abstract interpretation to build a static analysis that can verify safety in the tainting semantics. The analysis is proven correct on the While language following the usual framework of abstract interpretation.

The static analysis has been implemented by leveraging the Verasco abstract interpreter. This has two advantages, first, Verasco analyzes code close to source level which allows us to give useful feedback indicating the location of the culprit instruction to the programmer that seeks to understand what error were made. Second, we benefit from the CompCert and Verasco framework which gives strong semantic guarantees. However, the modifications to Verasco are not yet proven in Coq. To finish this, we would need to adapt the current proofs in Verasco to take taints into account. This is quite a daunting task and a challenging proof engineering exercise, as the modifications cannot be done modularly and require to modify directly the memory abstraction of Verasco, which represents around 6,000 lines of Coq [Jou+15].

Finally, the prototype has been experimentally evaluated on a number of representative cryptography libraries and shown to be able to scale. Furthermore, difficult problems that were previously out of reach of state-of-the-art tools were solved by our prototype thanks to the usage of advanced abstract interpretation techniques. Unfortunately, our tool suffers from the same blight that affects all tools that operate on source code: "Is the security property preserved by compilation ?" The two following chapters present different methods to solve this issue.

CHAPTER 4

VERIFICATION AT THE ASM LEVEL

We previously presented an analysis at source level to verify whether a program respects the constant-time security property. A question remained, whether we can trust a compiler to preserve security properties. One simple solution is to not trust it at all and to verify at assembly level that the security policy is still respected. Such an analyzer at assembly level for CompCert had already been presented in [Bar+14]. However, this analyzer suffered many drawbacks due to the sheer complexity of analyzing assembly code. For instance, the authors had to manually rewrite the code they analyzed in order for the code to fit the constraints required by their tool. This included lifting local arrays of functions to global arrays in order to obtain artificial memory separation as the arrays aren't merged into the function' stack anymore. Other modifications involve inlining all functions in order to avoid inter-procedural analysis. Inlining all functions may render some programs impossible to analyze as they would become too large to analyze. This happens for instance for Adam Langley's implementation of curve25519.

In order to not completely redesign a constant-time analyzer from scratch, one solution is to reuse their tool and improve it. As their rewriting is due to the difficulty of obtaining useful alias information from assembly code, one way to improve their tool is to provide it more useful analyses. Considering the powerful analyses provided by Verasco, the matter is then to manage to transfer the alias information obtained by Verasco down to assembly.

We present in this chapter a method that follows this solution, it combines two ideas, namely defensive programming and relational verification. Defensive programming is a coding methodology to ensure that instructions can be safely executed, this is done by inserting defensive checks (assertions) in the code that make the programs abort if they fail. For instance, `z = x / y` is modified into `if (y != 0) z = x / y else abort()` in order to ensure that a division by zero cannot happen. Relational verification's goal is to check whether a program verifies a property relatively to another program. Relative safety is a particular instance of relational verification which considers the problem of verifying whether a program Q is safe knowing that program P is safe and P and Q are related by some relation R, where R describes the similarity between both programs, they can for instance be syntatically equal or differ only on variable names, etc.

How these two ideas are combined to transport information from source to target level will be explained in more details in the first section. The second section describes our particular instantiation of defensive programming while the third section details the implementation of a relative-safety checker. Section 4.4 details the analysis we implemented to take advantage of the information provided by Verasco and presents the experimental evaluation of our methodology, followed by conclusion in Section 4.5.

The work presented in this chapter has been presented at the 30th Computer Security Foundations Symposium (CSF) in [Bar+17]. Sections 4.3 and 4.4 are mainly the contributions of Vincent Laporte and are presented for the sake of completeness. The companion development is available at http://www.irisa.fr/celtique/ext/csf17/.

4.1 Methodology

Our approach relies on the combination of defensive programming and relational verification as well as clever usage of the properties of a *correct* compiler and *correct* static analyzer such as CompCert and Verasco. It is illustrated in Fig. 4.1.

Consider a compiler $[\cdot] : \texttt{Prog}_{\mathcal{S}} \longrightarrow \texttt{Prog}_{\mathcal{T}}$, a property ϕ over source programs $\texttt{Prog}_{\mathcal{S}}$ and its counterpart property ψ over target programs $\texttt{Prog}_{\mathcal{T}}$. The aim of the methodology is to provide a process to check, given a source program p that satisfies ϕ, whether the compiled program $[p]$ satisfies ψ. This works as follows.

Suppose that the compiler $[\cdot]$ preserves safety (which is a consequence of compiler correctness as demonstrated in Section 2.1.2.2), i.e., for every source program p, if $\texttt{safe}_{\mathcal{S}}(p)$ then $\texttt{safe}_{\mathcal{T}}([p])$. Further assume that there is a method to transform any source program p into a defensive program p_ϕ such that safety of program p_ϕ implies that p satisfies property ϕ. Similarly, we assume that there is an analogue method at the target level such that q_ψ is the defensive version of target program q with regards to ψ. Moreover, assume that we also have a static analyzer $\texttt{an} : \texttt{Prog}_{\mathcal{S}} \longrightarrow \mathbb{B}$ such that for every source program p, $\texttt{an}(p) = \texttt{true}$ implies that p is safe, i.e., $\texttt{safe}_{\mathcal{S}}(p)$. Finally, assume that we have a relative-safety checker $\texttt{relsafeC} : \texttt{Prog}_{\mathcal{T}} \times \texttt{Prog}_{\mathcal{T}} \longrightarrow \mathbb{B}$ such that for every target programs p and q, if $\texttt{relsafeC}(p, q) = \texttt{true}$ and p is safe, then so is q. The idea of the relative-safety checker is to verify that both programs are related by some relation R as described in the introduction. It is then left to prove that satisfying this relation R suffices to conclude relative safety.

Given these tools, we can verify that a source program p satisfies ϕ: if $\texttt{an}(p_\phi) = \texttt{true}$, then p_ϕ is safe, and therefore p satisfies ϕ by definition of p_ϕ. As we assume that the compiler preserves safety, we can also deduce that $[p_\phi]$ is safe. However, what we want to know is whether $[p]$ satisfies ψ, i.e., whether $[p]_\psi$ is safe. $[p_\phi]$ and $[p]_\psi$ are not necessarily equal, but may be similar as ψ is the target-level counterpart of

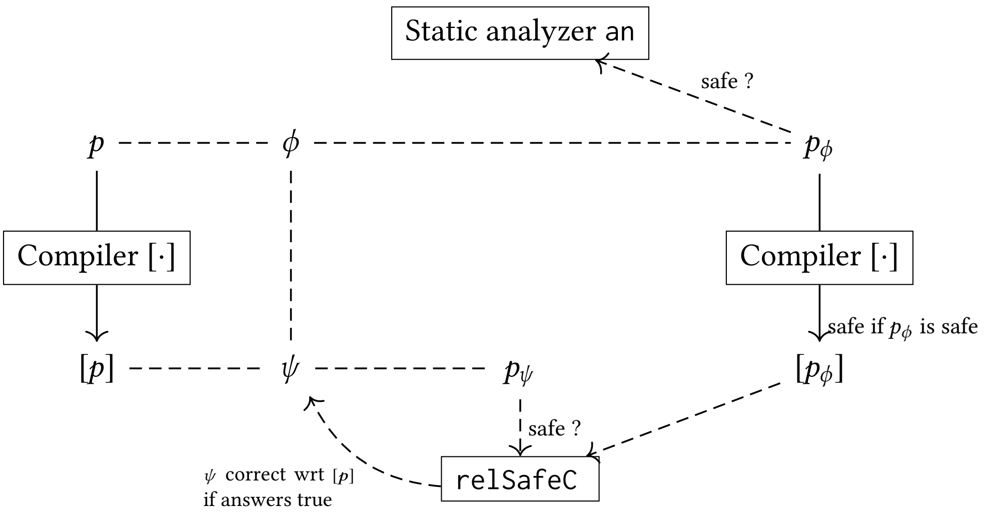

Figure 4.1: Overview of the methodology

ϕ. The crux of the matter is to prove that $[p]_\psi$ is safe knowing that $[p_\phi]$ is. This is exactly an instance of relational verification and can be solved by using `relsafeC`, i.e., if `relSafeC`$([p_\phi], [p]_\psi) =$ `true`, then $\texttt{safe}_{\mathcal{T}}([p]_\psi)$, and thus $[p]$ satisfies ψ.

We develop a verified instantiation of the methodology on top of the CompCert compiler and the Verasco static analyzer. The source language we consider is the C-like language C#minor which is the intermediate representation that is analyzed by Verasco. The target language is RTL which is the intermediate representation used for most optimizations in CompCert. This is a natural trade-off between engineering and proof effort. Indeed, stopping at RTL means that we have to manually prove the preservation of the property we consider down to assembly, which increases the proof effort. However, this is a relatively simple proof in our case and it has been done. Conversely, using RTL as the target language allows us to build the defensive form of target programs more easily than directly in assembly. This is in part due to having an infinite number of pseudo-registers at the RTL level, while at the assembly level, there are only a finite number of machine registers. The ϕ property we consider is satisfiability of *points-to* annotations, i.e., source programs are annotated with points-to information and the property is satisfied if the *actual* memory accesses occuring during the execution of the programs are within the range *denotated* by the points-to annotations. The relative-safety checker is detailed in Section 4.3. The defensive encoding of annotations is presented in the following section, as well as the proof that the encoding is "correct", i.e., for any program p annotated by ϕ, if its defensive form p_ϕ is safe, then p satisfies the annotations ϕ. We will also use Verasco to provide the points-to annotations but any other source of annotations could have been used.

```
char G[3], H;
int main(void) {
  int x;
  init(G, &x);
  return x; // 0.x: [0; 0]
}
```

```
void init(char *p, int *q) {
    p += any_int() % 3;
    *p = 0; // G: [0; 2]
    *q = 1; // 1.x: [0; 0]
}
```

Figure 4.2: A simple program

4.2 Defensive encoding of annotations

We present in this section how to generate defensive programs from programs annotated with points-to information with the aim that the encoding is *correct*; the defensive programs must fail when a defensive check (i.e, an assertion) is violated. We first describe its implementation and then its formal verification.

4.2.1 Annotation syntax

We focus on *points-to* annotations: each instruction that accesses the memory (i.e., every load and store) is annotated with an optional set of symbolic pointers. Moreover, during compilation, local variables of functions are forgotten and allocated in a single stack frame at different offsets during the compilation from C#minor to Cminor (i.e., before generating RTL code, on which our defensive transformation operates). Thus, we define a symbolic pointer as a symbolic block (either a global variable name or a depth in the call stack) together with a concrete range that denotes the pointer offset. Syntactically speaking, we use the annotation (d.x: [l; h]) to represent pointers to the variable x in the stack frame at relative depth d in the call stack and whose offsets are between l and h; and the annotation (G: [l; h]) to represent the pointers to the global variable G whose offsets are between l and h.

As an example, consider the program of Figure 4.2; it is shown using C syntax for easier reading but the annotation inference is done at the C#minor level. The three annotations that are automatically inferred by the Verasco static analyzer are shown as comments in the figure. There are three memory accesses in this program: the store through pointer p, the store through pointer q, and the load of x at the end of the main function. The first one writes global variable G at some offset between 0 and 2 (because of the `% 3` modulo computation); it can thus be annotated with (G: [0; 2]) in the `init` function. The second one writes the local variable x of the `main` function; when this store is run, the `main` function is at relative depth 1 in the call stack; therefore this store is annotated with (1.x: [0; 0]). The third memory access loads the local variable x of the `main` function (*i.e.*, at relative depth 0 in the call stack); it is thus annotated with (0.x:

[0; 0]).

4.2.2 Lowering of annotations

It is now necessary to lower the annotations through the compilation chain. However, we do not have to prove that the annotations stay valid after each compilation pass as these passes are mostly optimization passes that are prone to changes and tweaks, this would also require to modify the proofs that the annotations stay valid every time. Instead, we verify once and for all at the end of the compilation chain that the annotations stay valid.

During stack allocation, local variables of functions are forgotten and simply allocated in a single stack frame at different offsets during the compilation from C#minor to Cminor. For instance, if a function has two local integer variables x and y, after stack allocation, its memory layout becomes a single stack represented by an array where offset 0 represents x whereas y is found at offset 4 (since x occupies 32 bits, or 4 bytes). The annotations thus need to be transformed in order to stay correct.

However, this transformation pass demands more caution. For instance, consider the example in Figure 4.2. The (1.x: [0; 0]) annotation in the init function only tells us that *q points to the local variable x of the function that is at relative depth 1 which may be main or some other function foo. This information is crucial in order to know how to transform the annotations as x is at offset 0 for the main function, but may be allocated at offset 16 for foo.

One possible solution is to amend the annotations so that they also track the names of the function in which the local variable is found. For instance, (1.x: [0; 0]) could be modified into (1.main.x: [0; 0]) to indicate the variable x local to function main. Unfortunately, this solution would have necessitated to modify Verasco to also track the function names which would have been an extensive endeavour.

Our workaround is to artificially do the stack allocation directly at the C#minor level by adding a verified pass to merge all local variables into a single one prior to running the annotation inference. Lowering the annotations during the actual stack allocation becomes simply the identity.

A second slightly problematic compilation pass is register allocation. During this transformation pass, loads and stores of 64-bits chunks of memory are each split into two operations, as CompCert only handled 32-bits architectures at the time[1]. It is thus necessary to shift the offsets in the annotations by 4 bytes. For instance, consider a 64-bits memory load annotated with the (0.x: [8; 8]), it thus loads a memory chunk between byte 8 and 16 as 64 bits corresponds to 8 bytes. After register allocation, this 64-bits load operation is split into two 32-bits memory loads, the first one annotated

[1] Our work was based on CompCert 2.6, while support for 64-bits architectures was added starting CompCert 3.0.

with (0.x: [8; 8]) as it loads the memory chunk between byte 8 and 12, while the second one is annotated with (0.x: [12; 12]) as it loads the chunk between byte 12 and 16.

All other passes during the compilation do not modify the memory accesses (but may remove them) and thus have no impact on the annotations.

4.2.3 Annotation encoding

We now need to define how to produce a defensive program which dynamically checks the validity of the annotations, i.e., for every memory access to pointer `p` annotated with a set α of symbolic pointers, the program checks that `p` is actually one of the pointers in the set of concrete pointers represented by α.

There are two cases, depending on whether the block of the pointer is definitely known. For instance, suppose that a memory access through pointer `p` is annotated with (G: [0; 4]). In this case, the annotation can be encoded as `G <= p && p <= G + 4` as the annotation indicates that `p` is definitely within the block corresponding to global variable G.

On the other hand, if the annotation is { (G: [2; 3]) , (H: [1; 2]) }, then pointer `p` can either be within the block corresponding to global variable G or H. As inequality comparisons between pointers within two different blocks is undefined in the C semantics, it is not possible to simply encode the annotion as `(G + 2 <= p && p <= G + 3) || (H + 1 <= p && p <= H + 2)`. Fortunately, equality comparison is defined. The issue can thus be circumvented by enumerating all possible pointers. The annotation would then be encoded as `p == G + 2 || p == G + 3 || p == H + 1 || p == H + 2`.

This second encoding might seem very inefficient, but since the defensive program is not meant to ever be executed, it is not really important. The defensive program is only used as a proof artefact to witness the validity of the annotations.

In order to encode the annotations, it is necessary to compute the concrete pointers corresponding to the symbolic pointers denoted by the annotations. However, there is an issue when the annotation refers to a local variable of a suspended function. For instance, in Figure 4.2, there is no direct way to forge a pointer to `main`'s local variable `x` from within the `init` function as needed to encode the (1.x: [0; 0]) annotation.

To forge such a pointer is generally not possible without runtime support, therefore, we make each function *leak* a pointer to its own stack frame into a global variable (the so-called *shadow stack*).

The shadow stack `STACK` is a global array that stores a pointer to the stack frames of each currently running function. Its general structure is illustrated on Figure 4.3. The top of the shadow stack is represented by a second global variable `SIZE` such that the top `STACK[SIZE]` always holds a pointer to the stack frame of the current function. The shadow stack must thus need to maintain the invariant that there are as many pointers in the shadow stack as there are functions in the call stack and each pointer corresponds to the stack pointer of one of these functions as described in Figure 4.3.

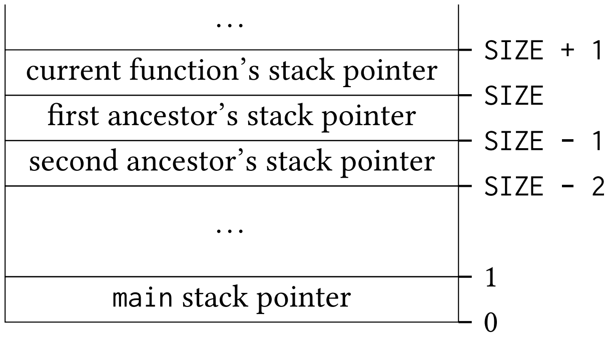
…
current function's stack pointer
first ancestor's stack pointer
second ancestor's stack pointer
…
main stack pointer
SIZE + 1
SIZE
SIZE - 1
SIZE - 2
1
0

Figure 4.3: General structure of the shadow stack

In order to maintain the invariant and the structure of the shadow stack, each function is given a *prologue* `SIZE = SIZE + 1; STACK[SIZE] = sp` that pushes its stack pointer atop the shadow stack and an *epilogue* `SIZE = SIZE - 1` that pops a value from the shadow stack. A function only has one entry point and thus only one prologue is needed, but it may have multiple exit points, therefore, an epilogue must be inserted before each `return` instruction.

4.2.4 Annotation semantics

The meaning of an annotation has already been informally described. However, in order to state and prove a correctness theorem, it is necessary to formally define the semantics of annotations. The global environment of a program allows us to statically compute the concrete addresses of its global variables, but the addresses of the stack frames depend on the actual execution state of the program.

At the RTL level, an execution state is defined as follows.

```
Inductive stackframe : Type :=
  | Stackframe:
      forall (res: reg) (* where to store the result *)
             (f: function) (* calling function *)
             (sp: val) (* stack pointer in calling function *)
             (pc: node) (* program point in calling function *)
             (rs: regset), (* register state in calling function *)
      stackframe.

Inductive state : Type :=
  | State:
      forall (stack: list stackframe) (* call stack *)
             (f: function) (* current function *)
             (sp: val) (* stack pointer *)
             (pc: node) (* current program point in c *)
             (rs: regset) (* register state *)
             (m: mem), (* memory state *)
      state
```

The state `State s f sp pc rs m` records the stack pointer `sp` of the current function and a list of the stack pointers of the suspended function within the list of stackframe `s`. Therefore, to dynamically interpret an annotation, we extract the list of stack pointers `sps` such that its first element is the current stack pointer, the second is the stack pointer of the caller function, and so on. Given a list of stack pointers `sps`, the pointer `p` is in the denotation of the annotation (d.x: [l; h]) if there exists `sp` such that `sp` is the d-th element of `sps` and there exists an integer `ofs` such that `p` is equal to `sp + ofs` and l ≤ `ofs` ≤ h. x in the annotation is not used as it is a legacy of the analysis at C#minor and corresponds to a variable name that no longer exist at the RTL level as it has been merged into a function's stack.

4.2.5 Correctness theorem

An execution state is said to be correctly annotated when either the next instruction to be executed is not an annotated memory access, or it is a memory access through a pointer `p` and it is annotated with a symbolic set of pointers α, such that pointer `p` belongs to the denotation of α.

The correctness theorem of the defensive encoding of a program ensures that the validity of the annotations is completely assessed by the safety of the defensive program.

Theorem 4.1 (Precision of the defensive form)**.** Given a *safe* annotated RTL program p, if the defensive version of p is also safe, then every reachable state in the execution of p is correctly annotated.

This theorem is only proven in Coq at the RTL level and not at the C#minor level as we do not need it for our methodology. Indeed, we only require the defensive program to be safe. In order to prove this theorem, we equip the original program p with a *blocking* semantics which refines the original RTL semantics to dynamically check, before every execution step that the current state is correctly annotated. This is simply defined in Coq as follows.

```
Definition step_block (s1: state) (t: trace) (s2: state) :=
 step s1 t s2 /\ annotations_correct s1.
```

A step in the blocking semantics is defined as the step being allowed in the regular semantics and the starting state being correctly annotated. Thus, proving that p is safe with regards to the blocking semantics entails that every reachable state of the program is correctly annotated.

The standard technique used throughout CompCert to prove that safety is preserved is to show a simulation between both programs. However, the corresponding compiler transformations need only to prove a forward simulation (i.e., that a safe original program results into a safe transformed program), while we need to prove the opposite direction (i.e., safety of the defensive program implies safety of the annotated original program in the blocking semantics). We thus have to directly show a backward simulation between the transformed program p' and the original program p. This cannot be obtained from a forward simulation as usually done in CompCert, as we would need to be able to match one step in the defensive program with steps in the original program, which is not possible for steps involved in the defensive checks. As always with such simulation proofs, the gist of our proof is to define the matching relation between execution states of both programs.

The relation must describe which sort of invariant holds that can explain why both programs exhibit the same behavior. The first invariant describes the shape of the transformed program with regards to the original program. In our case, the transformation adds two global variables to implement the shadow stack, it also adds a *prologue* and *epilogue* to each function in order to instrument the shadow stack, and finally each `load` and `store` operation is preceded by defensive checks to verify the correctness of the annotations.

The second and last invariant should concern the shadow stack in order to prove its correctness: there are as many pointers in the shadow stack as there are functions in the call stack, and each pointer corresponds to the stack pointer of one of these functions as described previously.

This invariant is obviously true during the initial state of the programs, as the shadow stack is empty and the `main` function is not yet called. As the shadow stack is never modified outside of the *prologue* and *epilogue* of each function, the invariant naturally holds. However, in both cases it is slightly tricky to prove that the invariant holds, especially in the case of the *prologue*. Indeed, we need to make sure that we do not go out of bounds of the shadow stack, as it is implemented by an array of finite size. This is ensured by the assumption that the defensive program is safe with regards to the regular semantics, which provides us the proof that there was no out-of-bounds access.

Finally, to prove a backward simulation between p' and p as defined in CompCert, we need to prove two additional lemmas, namely the *progress* lemma and the *simulation* lemma. The first one states that if s'_1 is a safe state of program p' (i.e., it is either a final state, or any state that can be reached from it is final or non-blocking) that matches with state s_1 of program p, then either s_1 is also a final state or it is non-blocking.

The main difficulty in proving this progress lemma resides in the case where s_1 is a state in which a `load` or a `store` is about to be executed. Then, according to the matching invariant, s'_1 is about to execute the assertions (i.e., defensive checks). The crux of the issue is to first prove that the assertions are valid because s'_1 is a safe state (the program is instrumented such that if an assertion fails then the program crashes), and that this entails that the annotations are correct.

Conversely, the simulation lemma states that if s_1 is a state of p that steps to s_2 and s_1 matches with safe state s'_1 of p', then there must exist state s'_2 such that s'_1 can reach s'_2 and s_2 and s'_2 are matched. This is similar to a forward simulation with the exception that we assume s'_1 is safe, which is not assumed in the standard forward simulation.

The main difficulty is also related to `load` and `store` instructions: we need to prove that the defensive checks are always successful. However, it is fairly easy since we assume that the defensive program is safe, and thus the defensive checks do not fail by assumption.

Finally, combining these two lemmas with the fact that initial states of both programs as well as their final states are matched enables us to prove that if p' is a safe program then p has the same behavior which entails that it must also be a safe program with regards to the blocking semantics, i.e., every reachable state in the execution of p is correctly annotated.

4.3 Relative-safety checking

The purpose of the relative-safety checker is to verify that a program R is safe provided that another program L is known to be safe. In our setting, these are two defensive programs at the RTL level as illustrated in Figure 4.1. They are, by construction, very similar. It is thus possible to directly prove another stronger property, namely behavior equivalence. This section describes the design of an equivalence checker and its formal verification. The contribution detailed in this section is mainly the work of Vincent Laporte and is presented for the sake of completeness.

4.3.1 Overview

In order to prove that program R has the same behavior as a program L, the equivalence checker will employ the same usual technique of simulations. This is separated into two tasks, first, we verify that the two programs are similar enough by constructing a *product program*. Second, we verify that the product is *valid*.

As both programs that we try to prove equivalent are very similar, i.e., both programs have the same control-flow, each time R branches, L also branches on the same condition, each time there is a function call in R, the same function call with the same exact arguments appears in L. This allows us to have a modular reasoning as we now only need to prove that functions on both sides are pairwise equivalent. As such, a product program is built from product functions which are themselves built by combining the functions of both programs. This is done by featuring non-*critical* instructions of both functions and assertions (not the same as the defensive checks as before) claiming that the critical instructions on both sides are the same. Critical instructions are instructions that may fail during execution such as memory accesses, as detailed in Section 2.1.2, and instructions that influence the control-flow such as function calls.

If the product function is valid, i.e., its assertions are valid, then the two functions are equivalent. If all product functions are valid, then the program product is valid and both programs are equivalent.

4.3.2 Program product

A program product is built by constructing the function products of functions that have the same names on both sides. When a register r appears in a function of program L, it is mirrored in the function product by a register r_l. Similarly, a register that appears in program R is mirrored by a register r_r in the function product.

In order to build the function product of functions f_l and f_r, we first start by assuming they have the same number of arguments and their arguments are pairwise equal, i.e., $\text{arg}_l^1 = \text{arg}_r^1$, $\text{arg}_l^2 = \text{arg}_r^2$, etc. We then mirror the instructions of f_l until a critical instruction is reached. For instance, if the function is `x = y + 1; return x`, the first critical instruction is `return x`. Thus, it is mirrored in the function product as $\mathtt{x}_l = \mathtt{y}_l + 1$. Similarly, we then mirror the instructions of f_r until a critical instruction is reached. Finally, we need to assert that the critical instructions on both sides are the same and use the same arguments. The process is continued until both functions are entirely visited. Obviously, the construction of the function product can fail if the critical instructions are not the same for instance. An example of the construction of a function product is given in Figure 4.4 where f_l is given on the left side, f_r on the right side and the function product is in the middle. f_l computes the absolute value of $x + x$ while f_r computes the absolute value of $2 \times x$.

The crux of the methodology lies in how the critical instructions serve as "synchronization" points between the two functions that we try to prove equivalent. Figure 4.5 details how the products of critical instructions are constructed. One particular element is the `havoc` operator which provides a non deterministic assignment that is useful to make our verification modular. Indeed, we do not need to track the memory state of the function product. As we can statically verify that the initial memory states of both programs L and R are the same by making sure that they have the same global variables and are initialized with the same values, we only need to make sure that both programs keep the same memory state after each memory write. This is ensured by the assertions as given by Figure 4.5. In the store case, the first assertion $p_l = q_r$ verifies that the memory accesses write at the same location and the second assertion $u_l = v_r$ verifies that the same value is written. This allows us to define the product of memory reads as only verifying that they access the same location by asserting that $p_l = q_r$.

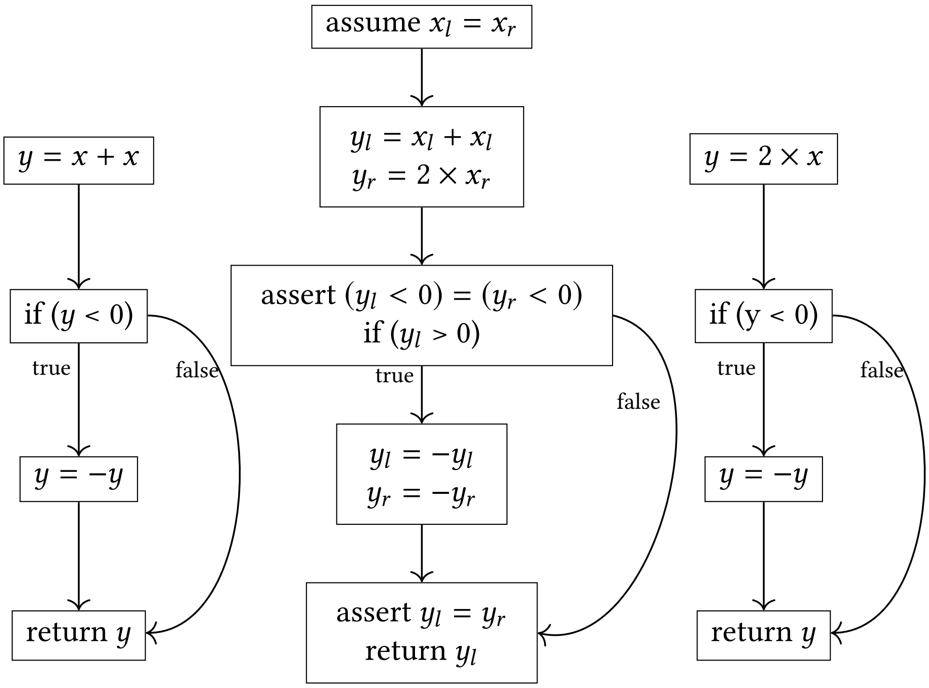

Figure 4.4: An example of function product

Whatever the value that is read, we do not care as illustrated by the usage of the havoc operator, but we are sure that both programs will read the same value and can thus safely write $y_r = x_l$.

Similarly, as we ensure that both programs call the same functions with the same arguments as illustrated in Figure 4.5 and functions are assumed equivalent, we do not care what results they return as they are equal, thus the use of havoc in the product of function calls.

4.3.3 Valid product

The validity of the assertions within the product program justifies the equivalence between the two programs L and R. It is thus crucial to verify that they are indeed valid.

To this end, a verification condition generator has been implemented. However, as the programs we analyze may contain loops, it is necessary to infer loop invariants. In our case, the loop invariants correspond to equality of the variables that are live at the loop headers of the initial programs. For instance, consider the following function that computes the factorial of parameter n.

```
int fact(int n) {
```

Left	**Right**	**Product**
$x = \mathrm{load}_\kappa\ p$	$y = \mathrm{load}_\kappa\ q$	assert $p_l = q_r$ $x_l = \mathsf{havoc}$ $y_r = x_l$
$\mathrm{store}_\kappa(p, u)$	$\mathrm{store}_\kappa(q, v)$	assert $p_l = q_r$ assert $u_l = v_r$
if (x)	if (y)	assert $\mathsf{cnz}(x_l) = \mathsf{cnz}(y_r)$ if (x_l)
$x = p(u^1, \ldots, u^n)$	$y = q(v^1, \ldots, v^n)$	assert $p_l = q_r$ assert $u_l^1 = v_r^1$ $\vdots$ assert $u_l^n = v_r^n$ $x_l = \mathsf{havoc}$ $y_r = x_l$
return	return	return
return x	return y	assert $x_l = y_r$ return x_l

Figure 4.5: Product of critical instructions

```
  int res = 1;
  while (n > 0) {
    res = res * n;
    n = n - 1;
  }
  return res;
}
```

The live variables at the loop header are `n` and `res`, the loop invariant that we thus need is $\mathtt{n}_l = \mathtt{n}_r \wedge \mathtt{res}_l = \mathtt{res}_r$. The liveness analysis provided by CompCert is used to automatically infer these invariants.

The verification condition generator also produces a verification condition for each assertion that appears in the function product. For instance, the verification condition corresponding to the first assertion $(y_l < 0) = (y_r < 0)$ in Figure 4.4 is $x_l = x_r \longrightarrow y_l = x_l + x_l \longrightarrow y_r = 2 \times x_r \longrightarrow (y_l < 0) = (y_r < 0)$ which can be discharged by a simplification procedure we implemented in Coq[2] as this reduces by rewriting to a tautology.

[2] This is not entirely true as we did not implement arithmetic procedures to prove that $x_l + x_l = 2 \times x_l$, but this is not an issue in practice for our use case.

The resolution of the verification condition entails the validity of its corresponding assertion. Thus, if all verification conditions can be discharged, then the program is valid.

4.3.4 Simulation

The construction of the program product between programs L and R as well as its validity entails the existence of a simulation between L and R, more precisely, a “star” simulation as described in Section 2.1.2. This simulation furthermore implies the relative safety of the two programs (i.e., if L is safe, then so is R).

The proof sketch of the simulation is as follows. Two states are related if they have the same memory states and same stack pointers, their program counters pc_l and pc_r must be related by a program point pc in the product such that pc is the program point of the product of the first critical instructions that are reached after pc_l and pc_r.

Thus, if L is at a non-critical instruction and advances, R is either already a the next critical instruction and waits or also executes a non-critical instruction. On the other hand, if L is at a critical instruction, R can directly advance at its next critical instruction that can be safely executed since the validity of the program product asserts that it is possible and furthermore preserves the relation between the states. For instance, if the next instruction of L is $\mathrm{store}_\kappa(p, u)$, the program product asserts that the next critical instruction of R is some $\mathrm{store}_\kappa(q, v)$ and the validity of the two assertions $p_l = q_r$ and $u_l = v_r$ ensures that the same value is written at the same location in both programs, thus both programs still have the same memory states.

4.4 Experimental results

The analysis presented in [Bar+14] operates on the Mach intermediate language of CompCert. However, as explained at the beginning of this chapter, the analysis relies on a weak points-to analysis that doesn’t handle memory separation well and thus requires implementations from standard cryptography libraries undergo manual rewriting in order to manage to analyze them. The rewriting is extensive, including lifting local variables to global variables and full inlining of the programs, and consequently making some of them impossible to analyze.

We developed a cryptographic constant-time analysis similar to the one presented in [Bar+14]. Each program point is given a “state” that associates each register and each memory location to a flow-sensitive security level High or Low. The points-to information derived from the Verasco analyzer is used in order to track the security level of values in memory. A type system then ensures that no conditional jump nor memory access depend on high values. We also consider another variant of constant-time security, namely stealth constant-time security, inspired by stealth memory [EA07;

Program	Size	Infer (s)	Check (s)	Equiv (s)	Result†
blowfish	177	29.2	32.4	0.01	✓⃝
des	230	2.8	4.9	0.84	✓⃝
donna	1214	515	∞	310	☑
RC4	94	4.6	5.1	0.02	✓⃝
salsa20	342	6.0	10.4	0.56	☑
snow	871	2.7	8.2	0.12	✓⃝
tea	121	3.43	3.9	0.01	☑
core (1)	166	0.05	0.29	0.03	☑
core (2)	142	0.04	0.28	0.03	☑
core (4)	198	0.06	0.35	0.04	☑
aes	1147	38.3	119	137	✓⃝
sha3	457	62.5	207	3.1	☑

† ✓⃝ = S-Constant-Time, ☑ = Constant-Time

Table 4.1: Timings

KPM12]. This variant assumes that some chosen variables may be stored into stealth memory where memory accesses are constant-time and thus cannot leak information through their usage.

Table 4.1 describes the execution time of some test C programs. The first block gathers results for various implementations of cryptographic primitives found in mBedTLS (previously PolarSSL) [mbe14]. The second block reports on test programs from the NaCl cryptography library [BLS12]. The third block lists the results for two cryptographic algorithms found in CompCert's benchmark suite. For each test program, we report its size in terms of number of C#minor instructions, the duration of inferring the points-to annotations by Verasco (first run), the duration of checking the high-level defensive program (second run of Verasco), and the duration for proving the equivalence of the two defensive programs. The last column reports whether the program has been proven constant-time or stealth constant-time. One cell in the "Check" column reports ∞: this means that the validation of the high-level defensive program was not possible due to limitations of Verasco. The issue is that the defensive transformation produces test conditions that are too complex due to aliasing as explained in Subsection 4.2.3. The constant-time analysis at the end only take a few milliseconds for all programs, except program donna whose analysis requires a few seconds.

The running time of the whole verification process (inferring, checking, equivalence checking and constant-time analysis) is rather affordable for most program taking at most a few seconds, but can become quite frustrating for some programs such as donna which comes close to 15 minutes. This happens when there is a lot of aliasing, i.e., when a memory access can point to different blocks, and all pointers symbolized by the annotation must be enumerated as explained in Subsection 4.2.3. This can happen when there is a "wrapper" function for memory accesses and is used from different functions. A way to improve this situation would be to duplicate the wrapper function so that each call site calls different functions.

4.5 Conclusion

In this chapter, we have proposed a method that cleverly combines defensive programs and relational verification to validate the translation of results from source level analyses to low-level programs. This method was instantiated with the CompCert compiler and the Verasco static analyzer. Thanks to the translation of points-to information, we managed to analyze more programs than and also programs that were previously out of reach by [Bar+14], hence providing a largely automatic way to check for constant-time security directly at low-level, in opposition with [Bar+14] where programs would need to undergo extensive manual rewriting.

The methodology presented provides a solution to verify that the code that is actually executed is cryptographically constant-time. However, what happens if it is rejected? Using the analysis presented in the previous chapter can tell whether your source code is secure. If the source code is secure, while the compiled code isn't, the problem then lies with the compiler. The programmer has thus not much more recourse than trying to tweak the compiler's options or trying to rewrite its code in a way that the compiler doesn't break its security. The solution presented in the following chapter provides an answer to this issue by showing how to prove that a compiler preserves constant-time security.

CHAPTER 5

PRESERVATION OF CRYPTOGRAPHIC CONSTANT-TIME SECURITY

A natural follow-up to verifying constant-time security at source level is to ask whether this security property is preserved by compilation. Indeed, optimizations can often hinder security. For example, we present three ways to write the same "selection" function that either returns the first or second parameter depending on the value of a boolean:

```
unsigned not_constant_time(unsigned x, unsigned y, bool b)
{
    if (b) { return y; }
    else { return x; }
}

unsigned constant_time_1(unsigned x, unsigned y, bool b)
{ return x + (y - x) * b; }

unsigned constant_time_2(unsigned x, unsigned y, bool b)
{ return x ^ ((y ^ x) & (-(unsigned) b)); }
```

The first version is self-explanatory, it returns `y` if `b` is true and `x` otherwise. The second version uses the fact that parameter `b` has type `bool`, which in C, is represented by unsigned integers 0 (false) or 1 (true)[1]. If its value is 1 (true), then the returned value is `x + (y - x)` which is equal to `y`. Otherwise, it returns simply `x` since `(y - x) * 0`

[1] More precisely, it is only true since C99 when `<stdbool.h>` is included.

`= 0`. The third version is more elaborate, it uses bitwise operator XOR `^` and bitwise operator AND `&`. It also exploits the wrap around behavior of unsigned integers and since `b` is either 0 or 1, `-(unsigned) b` becomes either -0 = 0 or the integer which has only 1 as bits ($2^{32} - 1$ for 32 bits architectures). The result of the bitwise AND operation `((y ^x) & (-(unsigned) b))` is thus `y ^ x` if `b` is true and 0 otherwise. Finally, since `x ^ (y ^ x) = y` and `x ^ 0 = x`, the function returns the expected result of `y` if `b` is true, and `x` otherwise.

If we consider the boolean parameter a secret, the first version is not constant-time as it branches on it, whereas the second and third version are constant-time. However, when compiled for older architectures that do not support conditional moves such as i386 or i486, the compiler Clang version 7.0.0[2] produces code that is not constant-time. The assembly code generated by the compiler is reproduced below in AT&T syntax.

```
not_constant_time: # not constant time
  movb 12(%esp), %al
  testb %al, %al
  jne .LBB0_1
  leal 4(%esp), %eax
  movl (%eax), %eax
  retl
.LBB0_1:
  leal 8(%esp), %eax
  movl (%eax), %eax
  retl
constant_time_1: # not constant time
  movb 12(%esp), %al
  testb %al, %al
  jne .LBB1_1
  leal 4(%esp), %eax
  movl (%eax), %eax
  retl
.LBB1_1:
  leal 8(%esp), %eax
  movl (%eax), %eax
  retl
constant_time_2: # not constant time
  movb 12(%esp), %al
  movl 4(%esp), %ecx
  testb %al, %al
```

[2] Tested on March 1st, 2018 using the Godbolt compiler explorer https://godbolt.org/g/dx4nzC.

```
27   jne .LBB2_1
28   xorl %eax, %eax
29   xorl %ecx, %eax
30   retl
31 .LBB2_1:
32   movl 8(%esp), %eax
33   xorl %ecx, %eax
34   xorl %ecx, %eax
35   retl
```

We first notice that `not_constant_time` and `constant_time_1` both compile to the exact same code except for the label names as the compiler manages to understand that the multiplication by the boolean `b` is equivalent to testing it. The code works as follows, the value at `esp + 12` represents the third parameter of the function which is the boolean `b` in the source code according to calling conventions and is moved into register `al`. The `testb` instruction then sets the ZF (Zero Flag) flag if `b` is false (i.e. 0) and clears the flag otherwise. If the flag is set, then the `jne` jump at line 4 is taken and the effective address `esp + 8` which represents `y` in the source code is computed and loaded into register `eax` before returning. Otherwise, the flag is cleared, and the jump is not taken, `esp + 4` which represents `x` is similarly computed and loaded into `eax` before returning.

The code is thus not constant-time, as the `jne` jumps at line 4 and 15 depend on whether the previous `testb` instructions set the ZF flag. This is however decided by the value of the secret `b`. Similarly, for `constant_time_2`, the `jne` jump at line 27 depends on the boolean `b` and the code is thus not constant-time. The code for `constant_time_2` is interesting as the compiler manages to optimize away the `&` operator and only uses XOR operations. In the case when `b` is false, the instruction at line 28 sets `eax` to zero as the compiler managed to conclude that the `((y ^x) & (-(unsigned) b))` operation would result in zero. `eax` is then XORed with `ecx` which contains variable `x`. In the other branch, the operation at line 32 moves `y` into `eax`, then stores the result of `y ^x` into `eax` at line 33. The AND operation was removed as it is redundant. However, a peephole optimization could have noticed that the operations at line 33 and 34 are redundant, as the result in `eax` is the same before and after the two operations.

One could argue that it is not really harmful as both branches contain the exact same number of operations for the compiled `constant_time_1` function. However, this does not protect the program from an attack. For instance, an attacker could manage to modify the cache so that the `leal` load instruction is faster in one of the branches. This would make an attacker be able to distinguish which branch was taken and thus leak the secret.

What's most worrying is that `constant_time_2` uses the style of code recommended by cryptographers[3] that abuses bitwise operators in the hope that compilers do not manage to optimize it and therefore not break constant-time security.

Another example can be found in [Kau+16] where the authors present a timing attack on a constant-time implementation of an elliptic curve by exploiting the MSVC compiler which transforms a constant-time 64-bit multiplication into a variable-time routine on architectures that do not natively support 64-bit integers.

This chapter is split into four parts, the first section presents a theoretical framework that can be used to prove that a correct compiler preserves constant-time security by taking advantage of the compiler's proof of correctness. The second part shows examples of how this framework could be applied to some selected compilation passes. The third part studies how this could be applied to CompCert. The last part concludes and details the main differences with [BGL18] which presents a work concurrent to ours with a similar approach.

5.1 Framework

Suppose that we have a compiler that compiles programs in a source language $\mathcal{S}$ to a target language $\mathcal{T}$ modeled by a *partial* function $\mathsf{compile} : \mathcal{S} \rightharpoonup \mathcal{T}$. We further assume that both languages are deterministic as it will make further reasoning easier and that the compiler is correct, i.e., it satisfies the following theorem:

Theorem 5.1 (Correctness of compilation)**.** For all source program p, if p is safe and compiles into program $\mathsf{compile}(p) = p'$, then p' has the same observable behavior as p.

As before, "safe" means no undefined behavior, the semantics of the program does not get stuck. Observable behavior corresponds to the trace of events that can be observed when executing the program, such as asking an input to an user on the command line or writing an integer to it. Whether the program terminates can also be observed.

The theorem only states that observable behavior is preserved, it has no relation with constant-time security. Therefore, a correct compiler does not give guarantee that security is preserved.

We assume that a program has a unique initial state that is determined by the initial values contained in the program. In C and in CompCert, this is determined by the `main` function and all global declarations, i.e., the global variables and the function definitions. A program may have no initial state if it is not well-formed, for instance if it does not contain a `main` function. Having a unique initial state allows to state constant-time security informally as if two programs are "similar" then they have "same leakage". We

[3] For example, it is recommended in page 9 of RFC7748 Elliptic Curves for Security https://tools.ietf.org/html/rfc7748#page-10.

will use a predicate $\phi(p, p')$ to say that both programs have the same values for some initial public variables that are defined by ϕ and that both the programs are syntactically equal otherwise. It reads as p and p' are ϕ-similar. Given a smallstep semantics with transition $\cdot \longrightarrow \cdot$, we use $s \xrightarrow{l} s'$ to say that the semantic step from state s to state s' produces the leak l.

As we previously assumed the languages to be deterministic, constant-time security can thus be defined as follows:

Definition 5.1 (Constant-time security)**.** A program p is ϕ-constant-time if for any program p' such that $\phi(p, p')$ then p and p' have same leakage, i.e., if s_0 and s'_0 are the initial states of respectively p and p', then for all $n \in \mathbb{N}$, s_1 and s'_1 such that $s_0 \longrightarrow^n s_1$ and $s'_0 \longrightarrow^n s'_1$, then either there exists a (possibly empty) leak l, s_2 and s'_2 such that $s_1 \xrightarrow{l} s_2$ and $s'_1 \xrightarrow{l} s'_2$ or both executions are stuck at s_1 and s'_1.

Constant-time security can be stated as a non-interference property as previously, but it can also be defined with a simulation-based view. This will be more useful as all compiler correctness proof are usually stated as a simulation, and thus constant-time security preservation amounts to proving that simulations can be composed in a certain way that we will detail later. Without determinacy, this property wouldn't be "strong" enough as it uses an existential quantifier which does not constrain the actual executions of the programs to follow the execution given by the quantifier.

In order to prove that a program p is ϕ-constant-time, it suffices to prove that p is safe and that for all program p' such that $\phi(p, p')$, there exists a leak-preserving lockstep simulation illustrated in Figure 5.1 similar to the simulations presented in Chapter 2 and defined as follows:

Definition 5.2 (Leak-preserving lockstep simulation)**.** A leak-preserving lockstep simulation between a program p and a program p' is defined by a relation $\cdot \sim \cdot$ between states of p and states of p' such that:

- If s_i is the initial state of p and s'_i is the initial state of p', then $s_i \sim s'_i$;
- For every step $s_1 \xrightarrow{l} s_2$ leaking information l of program p and state s'_1 of p' such that $s_1 \sim s'_1$, there exists a state s'_2 such that $s'_1 \xrightarrow{l} s'_2$ and $s_2 \sim s'_2$;
- For every state s and s' such that $s \sim s'$, if s is a final state, then so is s'.

Given a leak-preserving lockstep simulation, we prove that it implies same leakage in the following lemma.

Lemma 5.1. *If p is safe and there is a leak-preserving lockstep simulation $\cdot \sim \cdot$ between p and p', then they have same leakage.*

$$\begin{tikzcd}
s_1 \arrow[r, dash, "\sim"] \arrow[d, "l"'] & s'_1 \arrow[d, dashed, "l"] \\
s_2 \arrow[r, dash, dashed, "\sim"'] & s'_2
\end{tikzcd}$$

Figure 5.1: Leak-preserving lockstep diagram
(Hypotheses in plain lines, conclusion in dashed lines)

Proof. Both p and p' have an initial state, respectively s_0 and s'_0.

We first prove by induction on $n \in \mathbb{N}$ that if $s_0 \longrightarrow^n s_n$ and $s'_0 \longrightarrow^n s'_n$ then $s_n \sim s'_n$.

- For $n = 0$, we only need to prove that $s_0 \sim s'_0$ which is true by definition of a leak-preserving lockstep simulation;
- Let's now prove for $n + 1$ assuming that it is true for n. We have $s_0 \longrightarrow^n s_n \longrightarrow s_{n+1}$ and $s'_0 \longrightarrow^n s'_n \longrightarrow s'_{n+1}$. By induction hypothesis, we know that $s_n \sim s'_n$. Thus, by using the leak-preserving lockstep simulation, we have that there exists s''_{n+1} such that $s'_n \longrightarrow s''_{n+1}$and $s_{n+1} \sim s''_{n+1}$ (we omit the leak given by the simulation as we don't need it). However, since we assume the languages deterministic, we have that $s'_{n+1} = s''_{n+1}$, and thus, $s_{n+1} \sim s'_{n+1}$.

The property is thus proven by induction.

Now, we prove that both programs have same leakage, i.e., for all $n \in \mathbb{N}$, if $s_0 \longrightarrow^n s_n$ and $s'_0 \longrightarrow^n s'_n$, then either both executions are stuck at s_n and s'_n, or there exists a leak l_n and states s_{n+1} and s'_{n+1} such that $s_n \xrightarrow{l_n} s_{n+1}$ and $s'_n \xrightarrow{l_n} s'_{n+1}$.

This is true since for any such s_n and s'_n, we just proved that $s_n \sim s'_n$. And since we assume that p is safe, either s_n is a final state of p and therefore s'_n is also a final state of p' thanks to the simulation, or there exists a leak l_n and a state s_{n+1} such that $s_n \xrightarrow{l_n} s_{n+1}$, and again, by the leak-preserving lockstep simulation and by determinacy, there exists a unique s'_{n+1} such that $s'_n \xrightarrow{l_n} s'_{n+1}$.

Finally, we proved that both programs have same leakage. □

However, the converse is not generally true, if two programs have the same leakage, it does not mean that either of them is safe. It is not a problem as we assume a compiler correctness setting, i.e., we assume that the source program is safe. The following lemma can thus be considered the converse of the previous one.

Lemma 5.2. If p and p' have same leakage, then there exists a leak-preserving lockstep simulation between p and p'.

Proof. Let s_0 and s'_0 be the initial states of respectively p and p'. We define $s \sim s'$ as there exists $n \in \mathbb{N}$ such that $s_0 \longrightarrow^n s$ and $s'_0 \longrightarrow^n s'$.

- We have trivially $s_0 \sim s'_0$ by taking $n = 0$.

- If $s_1 \xrightarrow{l} s_2$ and $s_1 \sim s'_1$, we need to prove that there exists s'_2 such that $s'_1 \xrightarrow{l} s'_2$ and $s_2 \sim s'_2$. Such a s'_2 exists, since by definition of $s_1 \sim s'_1$, there exists a n such that $s_0 \rightarrow^n s_1$ and $s'_0 \rightarrow^n s'_1$. Since $s_1 \xrightarrow{l} s_2$, there exists s'_2 such that $s'_1 \xrightarrow{l} s'_2$ or p and p' wouldn't have same leakage. Furthermore, $s_2 \sim s'_2$ by definition.

- If s is the final state of p and $s \sim s'$, then s' is the final state of p', or there would be l and s'' such that $s' \xrightarrow{l} s''$ which is impossible since p and p' have same leakage.

The leak-preserving lockstep simulation is thus defined. □

Constant-time security is a symmetrical property in the sense that if p and p' have same leakage, then p' and p have same leakage. Thus, an equivalent definition would be that there exists a leak-preserving lockstep simulation between p and p' and another one between p' and p. However, we chose to trade the second simulation with the assumption that p is safe. This trade has a few advantages, in that we only need to prove one simulation instead of two to prove that a program is constant-time. Furthermore, assuming that the program given to the compiler is safe is a reasonable assumption that is also made when proving the correctness of the compiler.

We have shown that constant-time security implies existence of leak-preserving lockstep simulations, while safety and lockstep simulations are needed to prove constant-time security. Therefore, one possible way to prove the preservation of constant-time security through compilation is to 1. prove that safety is preserved through compilation, 2. the leak-preserving lockstep simulations are preserved through compilation and 3. assume that the initial program is safe. Preservation of safety is already a consequence of the correctness of the compiler.

We now have to solve the issue of how to preserve leak-preserving lockstep simulations. Compiler correctness can be stated as trace preservation and is proven through the usage of events preserving simulations. There are several kinds of such simulations, from the most constrained to the most general, they are the lockstep, plus and star simulations illustrated in Figure 5.2 and previously defined in Chapter 2. We remind the definition of the star simulation as it is the most general one.

An event preserving star simulation between a program p and a program p' is defined by a relation $\cdot \sim \cdot$ between states of p and states of p' such that:

- If s_i is the initial state of p and s'_i is the initial state of p', then $s_i \sim s'_i$;

- There exists a measure function $m : \mathbb{S} \rightarrow \mathbb{N}$ where $\mathbb{S}$ is the type of states of p;

- For every step $s_1 \xrightarrow{e} s'_1$ producing event e of program p and state s_2 of p' such that $s_1 \sim s_2$, either there exists a state s'_2 such that $s_2 \xrightarrow{e}^+ s'_2$ and $s'_1 \sim s'_2$, or e is a silent event (i.e., the step produces no event) and $m(s'_1) < m(s_1)$;

$$\begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d, "e"'] & s_2 \arrow[d, dashed, "e"] \\ s_1' \arrow[r, dashed, dash, "\sim"'] & s_2' \end{tikzcd}$$

(a) Lockstep diagram

$$\begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d, "e"'] & s_2 \arrow[d, dashed, "+e"] \\ s_1' \arrow[r, dashed, dash, "\sim"'] & s_2' \end{tikzcd}$$

(b) Plus diagram

$$\begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d] & s_2 \arrow[dl, dashed, dash, "\sim"', "m(s_1)>m(s_1')"] \\ s_1' & \end{tikzcd} \quad \text{or} \quad \begin{tikzcd} s_1 \arrow[r, dash, "\sim"] \arrow[d, "e"'] & s_2 \arrow[d, dashed, "+e"] \\ s_1' \arrow[r, dashed, dash, "\sim"'] & s_2' \end{tikzcd}$$

(c) Star diagram

Figure 5.2: Trace preserving simulations
(Hypotheses in plain lines, conclusion in dashed lines)

- For every state s and s' such that $s \sim s'$, if s is a final state, then so is s'.

The measure function used in the star simulation is to prevent p' from stuttering. Otherwise, a non-terminating program can be compiled into a terminating program and thus violates observable behavior preservation. For instance, suppose the source program is an infinite loop that does nothing, and it is compiled into a single instruction `skip`. Without the measure, the star simulation could be proven, even though behavior has not been preserved, since the source program is non-terminating while the compiled program is terminating.

Intuitively, we can see that the lockstep simulation used for constant-time security and the simulations used for compiler correctness can be composed. Suppose that we have two source programs p and p' such that p is ϕ-constant-time and $\phi(p, p')$. The leak-preserving lockstep simulation $\sim_S$ (S as in Source) tells us that if s_1 is a state of p and s_1' is a state of p' such that $s_1 \sim_S s_1'$ and s_1 advances to some state s_2 while leaking l, i.e., $s_1 \xrightarrow{l} s_2$, then there exists a state s_2' such that $s_1' \xrightarrow{l} s_2'$ and $s_1' \sim_S s_2'$. As we assume the compiler is correct, we know that there is some simulation $\sim_C$ (C as in Compile) to prove that p is correctly compiled, and similarly a simulation $\sim_C'$ for p'. The first simulation tells us that since $s_1 \xrightarrow{l} s_2$, for all σ_1 such that $s_1 \sim_C \sigma_1$, there exists a leak λ and a state σ_2 such that $\sigma_1 \xrightarrow{\lambda}{}^n \sigma_2$ where n is some unknown integer. Similarly for the second simulation, it tells us that since $s_1' \xrightarrow{l} s_2'$, for all σ_1' such that $s_1' \sim_C' \sigma_1'$, there exists a leak λ' and a state σ_2' such that $\sigma_1' \xrightarrow{\lambda'}{}^{n'} \sigma_2'$ where n' is some unknown integer.

This feels like the beginning of a simulation diagram, but still requires proving that $\lambda = \lambda'$ and $n = n'$. We do not need to prove that $\lambda = l$ as leaks are generally not preserved by compilation. For instance, some optimization may remove memory accesses if it deems them unnecessary, the leak due to the memory accesses at the source level is thus removed when compiled. What's important is that the compiled leaks stay the same, i.e., $\lambda = \lambda'$.

We define this as a 2-simulation diagram that is characterized by three relations ($\sim_S, \sim_T^{pre}, \sim_C$) and detailed below. The last relation $\sim_C$ corresponds to the relation used in proving that the source program is correctly compiled into the target program. There should be two such relations since there are two programs p and p', however, these two relations are morally the same as both programs have been compiled with the same transformation. We thus use only one relation $\sim_C$ for the sake of readability.

Definition 5.3 (2-simulation diagram)**.** ($\sim_S, \sim_T^{pre}, \sim_C$) is a 2-simulation diagram for programs p, p', ρ, ρ' if

- $\sim_S$ is a leak-preserving lockstep simulation at source level between p and p',
- $\sim_C$ is an event preserving star simulation between p and ρ that proves the correctness of compiling p into ρ,
- $\sim_C$ is an event preserving star simulation between p' and ρ' that proves the correctness of compiling p' into ρ',

and $\sim_T^{pre}$ is a target level relation between states of ρ and ρ' such that

- if σ_0 and σ_0' are respectively the initial states of ρ and ρ', then $\sigma_0 \sim_T^{pre} \sigma_0'$,
- for all states s_1, s_1', s_2, s_2', σ_1, σ_1' and leak l such that $s_1 \sim_S s_1'$, $s_1 \sim_C \sigma_1$, $s_1' \sim_C \sigma_1'$, $\sigma_1 \sim_T^{pre} \sigma_1'$, $s_1 \xrightarrow{l} s_2$, $s_1' \xrightarrow{l} s_2'$, then there exists an integer n, a leak λ and states σ_2, σ_2' such that $\sigma_1 \xrightarrow{\lambda}^n \sigma_2$, $\sigma_1' \xrightarrow{\lambda}^n \sigma_2'$, $s_2 \sim_C \sigma_2$, $s_2' \sim_C \sigma_2'$ and $\sigma_2 \sim_T^{pre} \sigma_2'$, this is illustrated in Figure 5.3
- for all states σ and σ', if $\sigma \sim_T^{pre} \sigma'$ and σ is a final state, then so is σ'.

Informally, the relation $\sim_T^{pre}$ defined in the 2-simulation diagram represents the fact that the two programs are at the exact same program point. How to define this is however dependent on the language, which is why we cannot abstract it away in the definition. Furthermore, the relation may not be a leak-preserving lockstep simulation relation as the diagram only tells us that there is some number of steps n between states that are related by $\sim_T^{pre}$, we are missing the lockstep part of the definition. We can however use it to build such a relation as proven by the following theorem, hence the *pre* in the symbol, as it can be seen as a pre-lockstep simulation.

We only consider program transformations that do not depend on the secrets. For instance, a transformation that would add n `skip` instructions at the beginning of the program is not allowed if n is secret. This is necessary in order to have transformations

$$\begin{tikzcd}
s_1 \arrow[r, "l"] \arrow[d, dash, "\sim_C"'] \arrow[rr, bend left, dash, "\sim_S"] & s_2 \arrow[d, dashed, dash, "\sim_C"] \arrow[rr, bend left, dash, "\sim_S"] & s'_1 \arrow[r, "l"] \arrow[d, dash, "\sim_C"'] & s'_2 \arrow[d, dashed, dash, "\sim_C"] \\
\sigma_1 \arrow[r, dashed, "\lambda"', "n"] \arrow[rr, bend right, dash, "\sim_T^{pre}"'] & \sigma_2 \arrow[rr, bend right, dashed, dash, "\sim_T^{pre}"'] & \sigma'_1 \arrow[r, dashed, "\lambda'"', "n"] & \sigma'_2
\end{tikzcd}$$

Figure 5.3: 2-simulation diagram
(Hypotheses in plain lines, conclusion in dashed lines)

that verify the property that if p is compiled into ρ and ρ and ρ' are ϕ-similar, then there exists p' such that p and p' are ϕ-similar.

Theorem 5.2 (Preservation of constant-time security). If program p is ϕ-constant-time, safe and there is a $(\sim_S, \sim_T^{pre}, \sim_C)$ 2-simulation diagram for all p' such that $\phi(p, p')$, then compile(p) is ϕ-constant-time.

Proof. Let ρ' be a program such that $\phi(\text{compile}(p), \rho')$, there exists a p' such that $\rho' = \text{compile}(p')$ by hypothesis. We first define the relation $\cdot \sim_T^n \cdot$ between states of compile(p) and compile(p') as follows:

$$\begin{aligned}\sigma \sim_T^n \sigma' \triangleq\, & \exists\lambda, \exists\sigma_1, \exists\sigma'_1, \exists s_1, \exists s'_1,\\ & \sigma \xrightarrow{\lambda}{}^n \sigma_1 \wedge \sigma' \xrightarrow{\lambda}{}^n \sigma'_1 \wedge \sigma_1 \sim_T^{pre} \sigma'_1 \wedge s_1 \sim_C \sigma_1 \wedge s'_1 \sim_C \sigma'_1 \wedge s_1 \sim_S s'_1\end{aligned}$$

We now define the lockstep simulation relation $\cdot \sim_T \cdot$ between states of compile(p) and compile(p') as $\sigma \sim_T \sigma' \triangleq \exists n, \sigma \sim_T^n \sigma'$.

Informally, this means that $\sigma \sim_T \sigma'$ if there exists some states σ_1 and σ'_1 such that σ and σ' can both respectively reach σ_1 and σ'_1 in the same number of steps while leaking the same information. Furthermore, there must exist some states s_1 and s'_1 in the source programs such that $s_1 \sim_C \sigma_1$ and $s'_1 \sim_C \sigma'_1$ and $s_1 \sim_S s'_1$.

We first show a lemma that for all n, σ_1 and σ'_1, if $n > 0$ and $\sigma_1 \sim_T^n \sigma'_1$, there exists λ, σ_2 and σ'_2 such that $\sigma_1 \xrightarrow{\lambda} \sigma_2$, $\sigma'_1 \xrightarrow{\lambda} \sigma'_2$ and $\sigma_2 \sim_T^{n-1} \sigma'_2$.

By definition of $\sim_T^n$, there exists λ, σ_3, σ'_3, s_3, s'_3 such that $\sigma_1 \xrightarrow{\lambda}{}^n \sigma_3$, $\sigma'_1 \xrightarrow{\lambda}{}^n \sigma'_3$, $\sigma_3 \sim_T^{pre} \sigma'_3$, $s_3 \sim_c \sigma_3$, $s'_3 \sim_C \sigma'_3$ and $s_3 \sim_S s'_3$. Thus, there exists σ_2, σ'_2, λ_1 and λ_2 such that $\sigma_1 \xrightarrow{\lambda_1} \sigma_2 \xrightarrow{\lambda_2}{}^{n-1} \sigma_3$, $\sigma'_1 \xrightarrow{\lambda_1} \sigma'_2 \xrightarrow{\lambda_2}{}^{n-1} \sigma'_3$ and $\lambda = \lambda_1 \cdot \lambda_2$. Hence, we can conclude that $\sigma_2 \sim_T^{n-1} \sigma'_2$ by definition.

We now show that $\cdot \sim_T \cdot$ is indeed a lockstep simulation:

- If σ_i is an initial state of compile(p) and σ'_i is the initial state of compile(p'), by safety of p and p', there exists s_i and s'_i respectively initial states of p and p'. By definition of $\sim_C$, we have $s_i \sim_C \sigma_i$ and $s'_i \sim_C \sigma'_i$, thus $\sigma_i \sim_T \sigma'_i$ with $n = 0$.

- If σ_f is a final state and $\sigma_f \sim_T \sigma'_f$, by definition of $\sim_T$, there exists some states s and s' such that $s \sim_C \sigma_f$, $s' \sim_C \sigma'_f$ and $s \sim_S s'$.

  By definition of a star simulation and since σ_f is a final state, there exists a state s^1 such that $s \longrightarrow s^1$, $s^1 \sim_C \sigma_f$ and $m(s^1) < m(s)$. By iterating this process, we build a finite maximal sequence $s^1, \ldots, s^k$ of states such that $s \longrightarrow s^1 \longrightarrow \ldots \longrightarrow s^k$ and $s^k \sim_C \sigma_f$. The sequence is finite because we have $m(s^1) > \ldots > m(s^k)$ and this cannot infinitely decrease as $\mathbb{N}$ is well-founded. $s_f = s^k$ is a final state, since otherwise there would be a state s^{k+1} such that $s^k \longrightarrow s^{k+1}$ and the sequence wouldn't be maximal.

  And by exploiting the lockstep simulation $\sim_S$, we can build a sequence of states $s'^1, \ldots, s'^k$ such that $s' \longrightarrow s'^1 \ldots \longrightarrow s'^k$, $s^1 \sim_S s'^1, \ldots, s^k \sim_S s'^k$. Since $s_f = s^k$ is a final state, then so is $s'_f = s'^k$ thanks to the lockstep simulation $\sim_S$.

  By definition of the 2-simulation diagram, $s'_f \sim_C \sigma'_f$, thus σ'_f is also a final state.

- If $\sigma_1 \sim_T \sigma'_1$ and $\sigma_1 \xrightarrow{\lambda} \sigma_2$, by definition of $\sim_T$, there exists n, σ_3, σ'_3, λ' such that $\sigma_1 \xrightarrow{\lambda'}{}^n \sigma_3$ and $\sigma'_1 \xrightarrow{\lambda'}{}^n \sigma'_3$ and there exists s, s' such that $s \sim_C \sigma_3$, $s' \sim_C \sigma'_3$ and $s \sim_S s'$. We need to prove there exists σ'_2 such that $\sigma'_1 \xrightarrow{\lambda} \sigma'_2$ and $\sigma_2 \sim_T \sigma'_2$.
  - If $n > 0$, we use the lemma, and therefore there exists λ_{bis}, σ_{2bis}, σ'_2 such that $\sigma_1 \xrightarrow{\lambda_{bis}} \sigma_{2bis}$, $\sigma'_1 \xrightarrow{\lambda_{bis}} \sigma'_2$ and $\sigma_{2bis} \sim_T \sigma'_2$. By determinism of the semantics, we have that $\sigma_2 = \sigma_{2bis}$ and $\lambda = \lambda_{bis}$. Thus we have $\sigma'_1 \xrightarrow{\lambda} \sigma'_2$ and $\sigma_2 \sim_T \sigma'_2$.
  - However, if $n = 0$, we have $\sigma_3 = \sigma_1$ and $\sigma'_3 = \sigma'_1$. Thus, we obtain $s \sim_C \sigma_1$ and $s' \sim_C \sigma'_1$. s cannot be a final state, because σ_1 would be a final state due to $\sim_C$ which is impossible since $\sigma_1 \xrightarrow{\lambda} \sigma_2$. Hence, by safety of p, there exists a state s_2 such that $s \xrightarrow{l} s_2$.

    By the definition of lockstep simulation with $\sim_S$, there exists some s'_2 such that $s' \xrightarrow{l} s'_2$ and $s_2 \sim_S s'_2$.

    By using the 2-simulation diagram, there exists $k \in \mathbb{N}$, σ_4, σ'_4 and λ^1 such that $\sigma_1 \xrightarrow{\lambda^1}{}^k \sigma_4$ and $\sigma'_1 \xrightarrow{\lambda^1}{}^k \sigma'_4$ with $s_2 \sim_C \sigma_4$, $s'_2 \sim_C \sigma'_4$ and $\sigma_4 \sim_T^{pre} \sigma'_4$. Therefore, by definition, $\sigma_1 \sim_T^k \sigma'_1$.

    If $k > 0$, we use again the previous lemma to conclude.

> Otherwise $k = 0$, and we know that $m(s_2) < m(s)$ by definition of $\sim_C$. Thus, we can reiterate the previous process until we obtain a new "k" that is strictly positive. This iteration process is finite because the measure strictly decrease until we obtain such a new k and it cannot decrease infinitely. The conclusion is hence the same as before.

We proved that $\sim_T$ is a lockstep simulation, thus $\mathsf{compile}(p)$ is ϕ-constant-time thanks to Lemma 5.1 and the theorem is proven. □

We proved that if the 2-simulation diagram is satisfied, then constant-time security is preserved. However, it is still left to prove that the simulation diagram can be satisfied by a compiler. Intuitively, we only know that given a star simulation $\sim_C$, when the states of the two high level programs advance, the lower level states will advance some number of steps n and n' which are not necessarily equal. However, the high level programs are in a lockstep simulation and thus follow *a fortiori* the same control flow, it makes sense that the lower level states advance similarly.

5.2 Examples

We present in this section two examples of transformation passes, one on which we can apply our framework, and a second example of transformation pass that doesn't generally preserve constant-time security, we will show how trying to unsuccessfully apply our framework can help understand why the pass does not preserve security.

5.2.1 Stack allocation

In the early passes of CompCert, each local variable of a function that has its address taken (i.e., non scalar variable) are allocated separately in the memory. One of the compilation passes allocates all these variables in a single stack for each function. Thus, instead of accessing for instance `&x` or `&y`, it becomes `stack + ofsx` or `stack + ofsy` where `stack` is a pointer to the stack and `ofsx` and `ofsy` are some integer offsets that are computed at compile-time.

The crux of the correctness proof of the stack allocation pass lies in the fact that each local variable of a function is associated a *constant* offset of the stack of the function during compilation. Hence, the offset does not depend on secret information; the resulting program does not leak *more* than the source program.

Theorem 5.3. Stack allocation preserves constant-time security.

Proof. Let p be a ϕ-constant-time program. We have to prove that $\rho = \mathsf{stack\text{-}allocate}(p)$ is constant-time. Let ρ' be a program such that $\phi(\rho, \rho')$, there is a program p' such that $\rho' = \mathsf{stack\text{-}allocate}(p')$ and $\phi(p, p')$ as such a program can be obtained by modifying p.

As p is ϕ-constant-time, there exists a $\sim_S$ leak-preserving lockstep simulation between p and p'. Moreover, since stack-allocate is correct, there exists simulation relations $\sim_C$ between p and ρ and between p' and ρ'. We also further assume that this simulation is lockstep, i.e., if $s \sim_C \sigma$ and $s \xrightarrow{l} s'$, then there exists σ' and λ such that $\sigma \xrightarrow{\lambda} \sigma'$ and $s' \sim_c \sigma'$. This is a reasonable assumption as stack allocation does not modify instructions, only the addresses that are used in memory accesses.

We define $\sim_T^{pre}$ as $\sigma \sim_T^{pre} \sigma'$ if σ and σ' are exactly the same except for the contents of their memories and their registers which are allowed to differ.

We prove that $(\sim_S, \sim_T^{pre}, \sim_C)$ is a 2-simulation.

- If σ_0 and σ_0' are respectively the initial states of ρ and ρ', since $\phi(\rho, \rho')$, σ_0 and σ_0' are the same except for the initial values of the "secret" variables, thus $\sigma_0 \sim_T^{pre} \sigma_0'$.

- If $s_1 \sim_C \sigma_1$, $s_1 \sim_S s_1'$, $s_1' \sim_C \sigma_1'$, $\sigma_1 \sim_T^{pre} \sigma_1'$, $s_1 \xrightarrow{l} s_2$ and $s_1' \xrightarrow{l} s_2'$, then there exists λ, λ', σ_2 and σ_2' such that $\sigma_1 \xrightarrow{\lambda} \sigma_2$ and $\sigma_1' \xrightarrow{\lambda'} \sigma_2'$ because of the correctness of the transformation.

  We first notice that since we have non-empty leaks only if we execute a branching instruction or a memory access and since stack allocation does not modify the instructions of the program, if l is an empty leak, then so are λ and λ'.

  Otherwise, l may be a leak due to a conditional, it is thus a boolean value and is preserved through compilation, hence $\lambda = l = \lambda'$. Or l is a leak due to a memory access, it is a pointer value. Either the location accessed is a global variable, then the same pointer is kept in the target executions since stack allocation does not touch global variables, hence $\lambda = l = \lambda'$. Or the l has the form `&x + ofs` where `x` is a variable local to a function `f`. Consequently, λ has the form `stack_f + ofsx + ofs` where `stack_f` is the address of the stack of function `f` and `ofsx` is the offset for `x`. Similarly, λ' has the form `stack_f' + ofsx' + ofs`. `ofsx` and `ofsx'` only depend on the definition of the function `f`, and since both programs p and p' have syntactically equal functions `f`, we have `ofsx = ofsx'`. Finally, since `stack_f` and `stack_f'` only depend on the control-flow and as by definition of $\sim_T^{pre}$, both σ_1 and σ_1' are at the same program point, we have `stack_f = stack_f'`, hence $\lambda = l = \lambda'$.

- If $\sigma \sim_T^{pre} \sigma'$ and σ is the final state of ρ, then, by definition of $\sim_T^{pre}$, σ and σ' are at the same program point, therefore σ' is the final state of ρ'.

This is thus a 2-simulation and ρ is constant-time. The theorem is hence proven. □

5.2.2 Memoization

Memoization is a technique to store the results of expensive computations so that when the same computations occur again, the results can be retrieved quickly instead of recomputing the results. This technique does not preserve constant-time security in general. Indeed, it transforms computations into memory accesses.

When trying to apply our framework, the proof would be stuck at function calls in our diagram. Indeed, suppose that in the source execution, a function call happen, either the function is not memoized, and the target execution keeps the function call. Or, the function is memoized, then the corresponding instructions in the target execution are a test to verify whether the inputs have been used before and if it is the case, an additional memory access. If the input does not depend on secret information, all is fine, however, if it is not the case, the proof is stuck since we have to prove that the results of the tests in the two target executions are equal, which is not possible.

5.3 Application to CompCert

We study in this section how the method presented previously can be adapted to CompCert. We first need to define our models by first defining what it means for programs to be similar, then what are the leaks we consider and finally how to augment each semantics with leaks.

In CompCert, a program is represented by the identifier of its `main` and a list of declarations which are global variables and function definitions. Thus, we can define similarity of programs `p1` and `p2` with regard to a set of identifiers that represent secret variables as `p1` and `p2` have the same `main` identifiers and the same function definitions, global variables are only allowed to differ if their identifiers are in the set of secret variables and are otherwise equal. This can be defined as follows in Coq where `match_except secret` is a predicate that says that the program definitions are similar except for variables in `secret` and `list_forall2 p l1 l2` means that for every element $a_1, a_2, \ldots$ of `l1` and $b_1, b_2, \ldots$ of `l2`, `p` a_i b_i holds.

```
Definition similar_programs (secret: list ident) (p1 p2: program): Prop :=
        p1.(prog_main) = p2.(prog_main) /\
        list_forall2 (match_except secret) p1.(prog_defs) p2.(prog_defs)
```

We then need to instantiate our model of leaks. For constant-time security, the leaks are either `Guard b` where `b` is a boolean due to the evaluation of the guard clause in a conditional, a memory access `MemAccess block ptrofs` or the leak is `Silent`.

Finally, in order for leaks to appear in semantics, we can rewrite each semantics to incorporate them but this would require extensive changes at all levels of the compiler.

A more modular way is to define an observation predicate observe for each semantics and define a "leaky" step as

```
Definition lstep (sem: semantics) (observe: state sem -> leak -> Prop)
                 (s1: state sem) (l: leak) (s2: state sem) :=
           exists e, step sem s1 e s2 /\
           observe s1 l.
```

observe s1 l means that when advancing from state s1, l will be leaked. s2 is not needed as the leak is entirely determined by what's executed which is contained in s1. We can now state constant-time security.

```
Definition secure (secret: list ident) (p: program): Prop :=
        forall (p': program),
           similar_programs secret p p' ->
           forall s0 s0',
              initial_state (semantics p) s0 ->
              initial_state (semantics p') s0' ->
              forall n s1 s1' t t',
                 StarN (semantics p) n s0 t s1 ->
                 StarN (semantics p') n s0' t' s1' ->
                 (exists l s2 s2',
                      lstep (semantics p) observe s1 l s2 /\
                      lstep (semantics p') observe s1' l s2') \/
                 (~ exists e s2, step (semantics p) s1 e s2 /\
                  ~ exists e' s2', step (semantics p') s1' e' s2').
```

This is exactly Definition 5.1 written in Coq, a program p is secure if for all programs p' that are similar with p with regards to secret, then if s0 and s0' are respectively their initial states, then for all states s1 and s1' such that s0 $\longrightarrow^n$ s1 and s0' $\longrightarrow^n$ s1', either both states s1 and s1' can take a leaky step with same leak l, or both executions are stuck.

A leak-preserving lockstep simulation is defined as a record in Coq.

```
Record lp_sim_properties (match_states: state -> state -> Prop): Prop :=
  Build_lp_sim_properties {
      lp_match_initial_states:
        forall s1,
          initial_state sem1 s1 ->
          exists s2, initial_state sem2 s2 /\ match_states s1 s2;
      lp_match_final_states:
        forall s1 s2 r,
          match_states s1 s2 ->
          final_state sem1 s1 r ->
          final_state sem2 s2 r;
      lp_simulation: forall s1 l s1',
          lstep sem1 s1 l s1' ->
          forall s2,
            match_states s1 s2 ->
            exists s2',
              lstep sem2 s2 l s2' /\
              match_states s1' s2' }.
```

The definition in Coq follows exactly Definition 5.2 but renames the ∼ relation into `match_states`.

The next step is to define the framework for 2-simulations. However, its definition relies on stating that the two executions at the target level (bottom part of Figure 5.3) advance the *same* number of steps. This number of steps is not random but is the number of steps prescribed by the event preserving simulation used for proving the correctness of the compiler. Yet, this number of steps does not appear explicitly in the theorem statement in CompCert as shown below.

```
fsim_simulation:
      forall s1 t s1', Step L1 s1 t s1' ->
      forall i s2, match_states i s1 s2 ->
      exists i', exists s2',
         (Plus L2 s2 t s2' \/ (Star L2 s2 t s2' /\ order i' i))
      /\ match_states i' s1' s2'.
```

This proposition states that if a state `s1` of `L1` advances to `s1’` while producing event `t` and it is related with state `s2` such that `match_states i s1 s2`, then there exists an index `i’` and a state `s2’` such that `s2` advances to `s2’` while producing event `t` and `s1’` and `s2’` are related, if the number of steps is not strictly positive (`Star` case), then `i’` must be less than `i` (i.e., `order i’ i`); the indexes `i` and `i’` represent the decreasing measure that we used in the previous section.

The number of steps does not appear at all, but it is a crucial part of our framework. Furthermore, we cannot only just state that there *exists* some number of steps as it would then be impossible to relate it to the number of steps taken by the “second” execution and make it impossible to reason with. One observation that can be made is that this number of steps already appears in the *proof* of the statement as the steps taken by `s2` are described inside of the proof. Moreover, as this number of steps only depends on how are `s1` and `s2` related, i.e., `match_states i s1 s2`, the simulation statement can be amended this way into a “counting” simulation.

```
counting_fsim_simulation:
      forall s1 t s1', Step L1 s1 t s1' ->
      forall n i s2, match_states n i s1 s2 ->
      exists s2', exists i', exists n',
         (StarN L2 n s2 t s2' /\ (n = 0 -> order i i'))
      /\ match_states n' i' s1' s2'.
```

The `match_states` relation is modified in order to take an additional parameter `n` which is a natural number that represents the number of steps taken by `s2` to reach `s2’`, if `n` is zero then the index must decrease. From our experiments on a few passes in CompCert, the necessary modifications to the proofs seem fairly minor.

Finally, it is time to study whether it is possible to prove that CompCert’s compilation passes verify our framework. We started our experiments by studying the constant propagation `Constprop` pass. This pass is interesting as it is one of the passes that modify the leaks. For instance, this pass can remove a memory load if the analysis manages to prove that it is redundant, `x = *p; y = *p` can be rewritten into `x = *p; y = x`.

In order to prove that this pass preserves constant-time security, we need to define the $\cdot \sim_T^{pre} \cdot$ relation presented in the previous section. As explained earlier, $\sigma \sim_T^{pre} \sigma'$ intuitively tells that both states σ and σ' are at the exact same program point. We define this in Coq as an “indistinguishability” relation. We first recall the RTL intermediate language that is used for most optimizations in CompCert such as `Constprop`.

An execution state in RTL is either a `Callstate`, a `Returnstate` or a regular `State`. They all record a list of stackframes `Stackframe res f sp pc rs` which contains a caller function `f`, its corresponding stack pointer `sp` and the program point where it was left at `pc`, its register state `rs` and the register `res` where the return value must be stored.

A `Callstate stk f args m` represents a state with the list of stackframes `stk` and memory `m` about to call the function `f` with arguments `args`. A `Returnstate stk v m` represents a state with list of stackframes `stk` and memory `m` that returns the value `v`. A `State stk f sp pc rs m` represents a state with list of stackframes `stk`, register state `rs`, memory `m`, current function `f`, stack pointer `sp` and program counter `pc`.

$$\frac{}{\texttt{Stackframe res f sp pc rs} \simeq \texttt{Stackframe res f sp pc rs'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{State stk f sp pc rs m} \simeq \texttt{State stk' f sp pc rs' m'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{Callstate stk f args m} \simeq \texttt{Callstate stk' f args' m'}}$$

$$\frac{\texttt{stk} \simeq \texttt{stk'}}{\texttt{Returnstate stk v m} \simeq \texttt{Returnstate stk' v' m'}}$$

Figure 5.4: Indistinguishability definition

We define the indistinguishability ≃ for stackframes and states in Figure 5.4. Two stackframes are indistinguishable if they are equal except for their register states that are allowed to differ. Two states are indistinguishable if their stackframes are indistinguishable and they are at the same program point.

The first property to prove for our 2-simulation is the following one: given programs p, p', ρ and ρ' such that p and p' are respectively transformed into ρ and ρ' after `Constprop`, the initial states of ρ and ρ' must be indistinguishable. The initial state of a program is `Callstate nil f nil m` where `f` is the function corresponding to the `main` function of the program, the memory `m` is just initialized with the global variables. Thus, proving that two initial states are indistinguishable comes down to proving that the two `main` functions are equal as we do not need to prove anything on the memory part. This is trivial as by definition of program similarity, the functions of both ρ and ρ' are pairwise equal, hence their `main` are equal.

The next step is to fulfill the diagram, part of what needs to be proven is that if two indistinguishable states in the target programs advance the same number of steps, then they both arrive at indistinguishable states. Let's have a closer look to function calls. The semantics for calls at RTL level is defined as follows in CompCert.

```
exec_Icall:
forall s f sp pc rs m sig ros args res pc' fd,
      (fn_code f)!pc = Some(Icall sig ros args res pc') ->
      find_function ros rs = Some fd ->
      funsig fd = sig ->
      step (State s f sp pc rs m)
        E0 (Callstate (Stackframe res f sp pc' rs :: s) fd rs##args m)
```

The rule says that if the instruction to be executed at program point `pc` is a call instruction `Icall sig ros args res pc'` and that given the register state `rs` and the register or symbol ros, the function called is fd, then the next state is a Callstate about to enter fd.

Now, suppose that both indistinguishable states of our target programs are at Icall instructions. They thus both arrive at Callstates. In order to prove them indistinguishable, we need to prove that the functions that are called are equal. There are two cases, either they are both called by name, i.e. ros is a symbol, or by pointer, i.e. ros is a register that contains the pointer value. In the first case, it is easy as both programs are similar, thus the symbol is associated to the same function in both programs. In the second case, it is not that simple. Indeed, we do not know the contents of the register states nor the memory, and cannot thus conclude that both function calls use the same pointer value, and even then we do not know whether the memory layout is different between the two programs.

The first idea one would have is to make use of the fact that in the diagram, there are source states s and s' such that $s \sim_C \sigma$ and $s' \sim_C \sigma'$ in order to exploit the correctness proof of compilation. The proof tells us that the function call in the transformed program corresponds to a call to the transformed form of the function called in the source program. By hypothesis, we know that the two function calls in the source program are equal. We need to be able to deduce from it that the functions called at the target level are equal. This reasoning would work for most passes, but unfortunately not for Constprop as it is one of the few program transformations that relies on an external analysis, i.e., the transformation depends on the results of the analysis.

This might not seem a difficult issue, as we could just think that since both programs are similar, then their analyses must be the same. This is true, but it is not that easy in presence of separate compilation which is supported by CompCert. Indeed, a user could compile multiple compilation units separately using CompCert and then link them together afterwards. Thus, the transformation of a function does not depend on the analysis of the *whole* program, but only on the compilation unit that it is in. This is where the issue lies as illustrated below.

```
Lemma functions_translated:
  forall (v: val) (f: fundef),
  Genv.find_funct ge v = Some f ->
  exists cunit,
         Genv.find_funct tge v = Some (transf_fundef (romem_for cunit) f)
                              /\ linkorder cunit prog.
```

The lemma states that for each function f in the initial program (represented by its global environment ge), the corresponding function in the transformed program is transf_fundef (romem_for cunit) f where cunit is a compilation unit contained in the whole program prog. This is problematic as the lemma states only that there exists a compilation unit but does not give enough constraint on it in order to relate the two compilation units we obtain from our two target states.

A possible solution is to not use the high-level lemmas provided by CompCert, but use a lower-level reasoning. The solution relies on the way global definitions are allocated during the initialization process. In CompCert, each global variable and function definition is associated a pointer, and this process is determined entirely by the order of the definitions. As we consider our two target programs to be similar, the order of definitions is the same. The global environment (the association table between definitions and pointers in CompCert parlance) is thus the same. Next, the correctness proof tells us that the target program uses the same pointer as in the source program. We thus only have to prove that the two pointers at the source level are the same to prove that they are also the same at the target level. The same pointers are used at the source level because we know that both programs called the same function and thus necessarily used the same pointer.

This shows some of the difficulties besides those inherent to our framework, but are due to the characteristics of adapting to a realistic compiler such as CompCert. Only the definitions given in this section have been formalized in Coq, the proofs presented in 5.1 have not been mechanized yet and are left as future work. Furthermore, we presented the troubles we encountered while trying to prove that the constant propagation pass preserves security, the proof has not been finished yet however.

5.4 Related Work and Conclusion

Concurrenly to our work, Barthe et al. [BGL18] have also studied the issue of preserving constant-time security through compilation and have developed an approach very similar to ours. Their paper presents their approach on a While language with a compiler built from scratch. This allows them to avoid pitfalls due to design choices of a preexisting compiler such as CompCert.

One notable difference in our methodology is that their methodology requires that when `match_states s1 s2` and `s1` advances one step, the number of steps advanced by `s2` must be computable by a function `num_steps` such that the number is `num_steps(s1, s2)`. For their example on the constant propagation pass, this necessitated to enrich the syntax of programs with annotations and thus modify the compilation pass to properly produce these annotations. For instance, in order to define their `num_steps` function, they need to statically know whether a branch is removed, they have to produce an annotated version of the source program with boolean flags telling whether the branch is removed to accomplish this. Thus, applying their method on CompCert would require to modify the syntax of the language and its semantics, which impacts all compilation passes that uses this language. It is preferable to avoid modifications if possible. Our method involves modifying the `match_states` relation so that it contains the expected

$$\begin{tikzcd}
& s_1' \arrow[rr, "l"] \arrow[dd, dash, "\approx"'] & & s_2' \arrow[dd, dash, "\approx"] \\
s_1 \arrow[ur, dash, "\equiv_S"] \arrow[rr, "l"] \arrow[dd, dash, "\approx"'] & & s_2 \arrow[ur, dash, dashed, "\equiv_S"'] \arrow[dd, dash, "\approx"] & \\
& \sigma_1' \arrow[rr, "\lambda'", "n'"' near end] & & \sigma_2' \\
\sigma_1 \arrow[ur, dash, "\equiv_C"'] \arrow[rr, "\lambda", "n"' near end] & & \sigma_2 \arrow[ur, dash, dashed, "\equiv_C"'] &
\end{tikzcd}$$

Figure 5.5: 2-simulation diagram from [BGL18]
(Hypotheses in plain lines, conclusion in dashed lines)

number of steps, thus we only have to modify the proofs and not the compilation passes.

A second difference is that their diagram (illustrated in Figure 5.5 using their notations) is slightly different from ours (in Figure 5.3). They directly assume for instance that there exists λ, n and σ_2 such that $\sigma_1 \xrightarrow{\lambda}^{n} \sigma_2$ and $s_2 \approx \sigma_2$ where $\approx$ is the relation used in the simulation for proving correctness of the compiler pass, while we ask to prove that such objects exist. They only ask to prove that $\lambda = \lambda'$, $n = n'$ and the dashed lines in the diagram. They are thus asking less things to prove than us. However, it seems intuitive that in order to prove that $\lambda = \lambda'$ in the diagram, it is necessary to be able to relate λ and λ' with l. We conjecture that they use the fact that $s_1 \approx \sigma_1$ and determinacy of the semantics in order to relate l and λ for instance. This is similar to unfolding the correctness proof of the transformation in order to relate the source and target leaks which is what our methodology imposes. Thus, in our opinion, the amount of work needed by both methodologies is similar.

As we have not finished mechanizing our development and the authors of [BGL18] only applied their approach on a toy compiler, we are planning to combine our efforts to apply our methodology to CompCert.

In this thesis, we started by presenting a method to verify constant-time security on source code. However, nothing guarantees that a secure code at source level stays secure when compiled as shown by the example in the beginning of this chapter. We thus provided two solutions, a first one in the previous chapter by presenting a method to verify whether assembly code respects constant-time security. This has the advantage that we now can be sure that the code that is *actually* executed is secure, but the tool can only report errors about the mangled code produced by a compiler which are of little use to a programmer. The second method presented in this chapter is the natural follow-up to Chapter 3 in which we presented a way to verify source code, “how do we prove that this security property is preserved by the compiler ?” As CompCert makes extensive use of simulations to prove its correctness, it was a logical step to exploit them in order to prove the property we wanted. Constant-time security is a property that can be stated as non-interference property or a simulation based property as shown in this chapter. This second way of stating constant-time security makes it easier to reason with the simulations used in CompCert. We presented in this chapter a framework to prove the preservation of constant-time security through compilation, this framework stems out from the definitions of the multiple simulations that are used and trying to assemble them together. This framework seems the most natural method, we will show in the next chapter other potential methods.

CHAPTER 6

CONCLUSION

6.1 Summary

Electronic communications have become more and more prevalent in our world through the democratization of the Internet, smartphones, contactless payments, etc. In order to ensure that communications are secure and private, cryptography has become especially crucial. The use of formal methods is thus natural in order to attain the highest degree of assurance possible as illustrated by the emergence of *high-assurance cryptography* and the plethora of recent publications in this area.

However, functional correctness is not sufficient. Exploitation of side-channels has recently become quite popular and in particular timing attacks due to the ease with which they can be remotely executed to recover secrets. Cryptographers have adopted the use of constant-time programming in order to avoid timing attacks. This thesis takes place in this context and applies formal verification to constant-time security.

In our work, we have given answers to different challenges pertaining to the verification of constant-time security. In Chapter 3, we provide a sound methodology to improve an abstract interpreter in order to verify that imperative programs are correctly written in the constant-time programming style. This can be used to help the programmer understand where the errors are if there are any.

However, there remains the question of whether the code that is *actually* executed satisfies the constant-time security policy. Indeed, compilers are known to often not respect the *intent* of the programmer. It is even more so the case when it is not even considered in the C standard as it is the case for side-channels. All bets are off and the compiler is free to remove all security countermeasures. We have proposed two possible answers to this challenge. The first one is to verify again that the compiled code is secure. However, due to the complexity of analyzing low-level code such as assembly code, it is extremely difficult to design a precise enough analysis at this level. In Chapter 4, we propose a way to avoid this issue by designing a methodology to transfer useful information that can be inferred at source level down to assembly in order to improve the analysis as this level. The second possibility is to directly prove that the compiler preserves constant-time security. Chapter 5 proposes a proof methodology based on the standard simulation framework used for verifying compiler correctness to prove that constant-time security is preserved.

6.2 Perspective

We report in this section various possible places of improvement for the different works we presented, as well as few ideas for extending our work.

6.2.1 Constant-Time Security Preservation Again

We presented one solution to the issue of preserving constant-time security through compilation in Chapter 5, but it is not yet completely mechanized. One obvious future work would thus be to finish it. There are currently discussions with the authors of [BGL18] in order to combine our efforts.

The solution presented in Chapter 5 involves modifying the standard simulations used for semantic preservation by reasoning on 2 different executions. Directly using the proposed simulation for each program transformation in CompCert could prove relatively cumbersome. We surmise it is possible to simplify it for a large proportion of the transformations. Indeed, only the CSE and constant propagation optimizations remove branchings, while the other compilation passes do not remove nor add branchings. Preservation of constant-time security can thus be split into two parts: preservation of branchings and another property concerning memory accesses.

This splitting has a significant advantage, namely that preservation of branchings only needs to reason about one execution and not two, which is much simpler. We still need to prove another property concerning memory accesses, preferably a property that also only reasons about one execution. We surmise that it is possible by proving that every location (pointer) accessed by a memory operation at the source execution is related to its corresponding location accessed in the target execution by a function that only depends on the program, or the function indicates that the corresponding memory access has been removed. In CompCert parlance, this corresponds to proving that the memory injection used in the semantic preservation proof can be entirely statically determined.

Thus, preservation of constant-time security can be reduced to proving the two simulations in Figure 6.1. Indeed, this suffices to prove the 2-simulation presented in Figure 5.3. If we reuse the notations of Figure 5.3, if l is a boolean leak (conditional guard) b, then λ and λ' are also b by Figure 6.1a. If l is a location loc, then λ and λ' are both equal to some location $f(loc)$ by Figure 6.1b. The only issue left is with compilation

$$\begin{tikzcd}
s_1 \arrow[r, "b"] \arrow[d, no head, "\sim_C"'] & s_2 \arrow[d, dashed, no head, "\sim_c"] \\
\sigma_1 \arrow[r, dashed, "b"] & \sigma_2
\end{tikzcd}$$

(a) Preservation of branching values

$$\begin{tikzcd}
s_1 \arrow[r, "loc"] \arrow[d, no head, "\sim_C"'] & s_2 \arrow[d, dashed, no head, "\sim_c"] \\
\sigma_1 \arrow[r, dashed, "f(loc)"] & \sigma_2
\end{tikzcd}$$

(b) Memory accesses are related by a function f

Figure 6.1: Simplification of constant-time security preservation

passes that *add* memory accesses such as register allocation which may spill some registers. It may be necessary to use the general 2-simulation in these cases.

One possible different way to tackle the issue is to verify a posteriori that the compilation did preserve constant-time security. This can be achieved by using the methodology described in Chapter 4. Indeed, we only need a program transformation such that the transformed program is safe if the original program is constant-time. We can take inspiration from program transformations for non-interference monitoring such as [Ass+13; Alm+16] for instance. Dynamic checks are added before each branching and memory access to verify that they do not depend on secrets. One way to do that is to add variables or registers to *shadow* the existing ones. These shadow variables and registers will be used to track the taint of the variables and registers they shadow, 1 is used if the taint is high and 0 otherwise.

For instance, `x = 4 * z + y` can be transformed into `shadow_x = shadow_z || shadow_y; x = 4 * z + y`, the taint of `x` is high if either of the taints of `z` and `y` is high. Similarly, `x = 42` is transformed into `shadow_x = 0; x = 42` as `x` is assigned a constant that does not depend on secrets. For conditional branching, it is only needed to check that the conditional guard does not depend on secrets. However, it becomes tricky when memory accesses are involved as the usual problem of aliasing comes into play. The solution would be to reuse the points-to annotations presented in Chapter 4. For instance, if we have `*p = 3 * x + y` where the memory access `*p` is annotated by Verasco with (T: [0; 2]), then we first need to check that the memory access might not leak secret, `assert (!shadow_p)` checks that `p` has a low taint. Next, the taints are updated as follows, each of the possible locations accessed is updated.

```
shadow_T[0] = shadow_x || shadow_y || shadow_T[0];
shadow_T[1] = shadow_x || shadow_y || shadow_T[1];
shadow_T[2] = shadow_x || shadow_y || shadow_T[2];
```

A weak update is used here as the annotation does not indicate exactly which location is accessed, therefore it is necessary to overapproximate the taint by keeping the previous one. This transformation could be used to verify that a program is constant-time, it is uncertain however whether the relative safety checker presented in Chapter 4 would cope with it.

6.2.2 Timing Attack Mitigations

Another possible axis of further work is to design program transformations to make programs constant-time as presented in [Mol+06] and [Cop+09]. Indeed, using the verifier presented in Chapter 3 to pinpoint the particular places in code that may leak secrets, it would be possible to automatically transform the code in these locations in order to remove leaks.

There are two sources of leaks, branchings and memory accesses that depend on secrets. In order to remove the branchings, one possible solution is to execute both branches in sequence and only keep the relevant computations. For instance, code such as `if (b) { x = A; } else { x = B; }`, where `A` and `B` are arbitrary computations, can be transformed into `x = b * A + (1 - b) * B`. If `b` is 1, then only the computation `A` is kept, otherwise `b` is 0, and only `B` is kept.

The second issue is memory accesses. One possible solution is to replace a single memory access to the index of an array to accessing the whole array. For instance, `x = t[pos]` can be replaced with `x = 0; for (i=0;i<N;i++) {x|=(i==pos)*t[i];}` where `N` is the length of the array `t`.

However, this solution is very slow as the whole array must be accessed instead of only one index. Another possibility is bitslicing. The essence of bitslicing is to consider a n-bit piece of data as n 1-bit pieces of data. For instance, instead of storing 32 bits of data into one register, it can be stored as 1 bit of data over 32 registers as the first bit of all these registers. A second 32-bit piece of data can then be stored as the second bit of the 32 registers, etc. Then, by using bitwise operators such as AND or XOR, it allows to parallelize 32 operations instead.

Bitslicing was historically first used by Biham [Bih97] in 1997 to replace the S-boxes of DES. These S-boxes are functions that take 6 bits of inputs and produce 4 bits of inputs, this can be implemented by using a lookup table with 64 (2^6) entries which is not secure as the inputs can depend on secrets. The bitsliced version of the first S-box of DES can be rewritten using 56 bitwise operations. This seems a lot more than one single table lookup, but if you consider 64-bit registers, the bitsliced version actually executes 64 different instances of the first S-box at the same time, which means that one instance of the first S-box costs less than one operation (56/64)[1].

The issue with bitslicing is that it seems that finding a bitsliced version of an algorithm is manually done by experts and seems difficult to automatize. However, as bitslicing is similar to software implementation of hardware circuits, it might be interesting to

[1] More information can be found on https://www.bearssl.org/constanttime.html#bitslicing

look in that direction to find optimization solutions, in particular circuit minimization. Another possible axis of research is dataflow languages as done by [Mer+18] which provides a domain specific language for implementing bitsliced algorithm and compiling them into C. According to their paper, the compiler validates *a posteriori* that the generated C code is correct with regards to the source code, but does not give more details. It would be interesting to know if it can be connected with CompCert.

6.2.3 A different security model

Constant-time security only considers branchings and memory accesses, but there are also other instructions that may be variable-time. For instance, on some older architecture, a multiplication may take different times to execute depending on the value of its operands. For instance, a multiplication might be faster when one of the operands is 0. This leaks information on the operands which may depend on secret values. Another source of leakage is floating point operations as illustrated by [And+15] which presents an attack that exploits variable-time floating point operations.

Unfortunately, no current chip vendor provides precise information on the timings of the processor's operations[2], only experimental studies exist[3]. One notable exception is the AVR microcontroller, for which the maker provides cycle precise information for instruction timings. This allowed [DMW17] to build a timing sensitive analysis for the 8-bit AVR microprocessor. This approach can work for simple architectures such as the AVR, but for more complex architectures, it would be interesting to modify the analyzer presented in Chapter 3 to be able to indicate whether some operations that may be variable time depend on secrets.

Another possible focus point is energy consumption side-channels as they become increasingly dangerous. Indeed, it was for a long time required to have physical access to the targeted machine in order to mount an attack. However, recently, remote power attacks have appeared. For instance, [ZS18] presents a remote attack on Field Programmable Gate Arrays (FPGAs) which are integrated circuits that can be programmed. They have recently been widely adopted in large scale datacenters. For instance, Amazon offers FPGA instances with its cloud services. As a user might not use all the available ressources (logical gates) of an FPGA, multiple users might use the same physical FPGA. [ZS18] presents a method to build a power monitor that can observe the power consumption of other modules on the FPGA, this allowed them to

[2] ARM had announced in November 2017 that their ARMv8.4-A chips would provide a new flag to indicate the use of constant-time operations, but this has since been removed from the announcement. A copy of the original announcement can be found at `http://web.archive.org/web/20171108050216/https://community.arm.com/processors/b/blog/posts/introducing-2017s-extensions-to-the-arm-architecture`.

[3] For instance, `https://www.bearssl.org/ctmul.html` provides a detailed study of which architecture supports constant-time multiplications.

mount a successful power analysis attack against an RSA cryptomodule.

Another example of remote power attack is [Man+18]. With the advent of green IT, CPU vendors have started to introduce software based ways to monitor power consumption. For instance, Intel has introduced the RAPL (Running Average Power Limit) feature which exposes the power consumption of the processor in a specific register. [Man+18] shows that it is actually quite simple to mount an attack against the RSA implementation of the Bouncy Castle cryptography library using the RAPL feature.

A popular security model to prove security against power analyses attacks is the probing security model [ISW03]. The idea of this security model is that the hardware can be thought of as a circuit with wires. The attacker is considered to only be able to observe a bounded number of those wires which also represent variables in programs. As the attack can only observe a bounded number of wires, a popular method of protection is *masking*, which consists in splitting secret variables into multiple shares. The higher the number of shares, the less information the attacker can obtain, but the less efficient the program is. This method is reminiscent of variable splitting in software obfuscation.

Final remarks In this thesis, we tackled *practical* challenges of securing cryptographic implementations against the timing side-channel. Obviously, there still exist other vulnerabilities, but the methods we presented show that formal verification *can* be used to ensure that complex cryptographic implementations satisfy certain security properties. There is no doubt that formal methods and security will become increasingly intertwined.

AUTHOR'S CONTRIBUTIONS

[Bar+17] Gilles Barthe, Sandrine Blazy, Vincent Laporte, David Pichardie, and Alix Trieu. “Verified Translation Validation of Static Analyses”. In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF).* 30th IEEE Computer Security Foundations Symposium. Santa-Barbara, United States, Aug. 2017, pp. 405–419. DOI: 10.1109/CSF.2017.16. URL: http://www.irisa.fr/celtique/ext/csf17/.

[BPT17] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation”. In: *European Symposium on Research in Computer Security.* 22nd European Symposium on Research in Computer Security. Oslo, Norway, Sept. 2017. URL: http://www.irisa.fr/celtique/ext/esorics17/.

[BPT18] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation (Extended version)”. In: *Journal of Computer Security* (2018). (Accepted for publication, to appear).

[BT16] Sandrine Blazy and Alix Trieu. “Formal Verification of Control-flow Graph Flattening”. In: *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs.* CPP 2016. St. Petersburg, FL, USA: ACM, 2016, pp. 176–187. ISBN: 978-1-4503-4127-1. DOI: 10.1145/2854065.2854082. URL: http://www.irisa.fr/celtique/ext/cfg-flatten/.

BIBLIOGRAPHY

[Alm+16] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations". In: *25th USENIX Security Symposium, USENIX Security 16, August 10-12, 2016.* 2016, pp. 53–70 (cit. on pp. 28, 29, 35, 53, 55, 57, 99).

[Alm+17] José Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "Jasmin: High-Assurance and High-Speed Cryptography". In: *CCS 2017-Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 2017 (cit. on p. 28).

[And+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "On Subnormal Floating Point and Abnormal Timing". In: *Proceedings of the 2015 IEEE Symposium on Security and Privacy.* SP '15. IEEE Computer Society, 2015, pp. 623–639 (cit. on pp. 57, 101).

[AP16] Martin R. Albrecht and Kenneth G. Paterson. "Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS". In: *Advances in Cryptology – EUROCRYPT 2016.* Ed. by Marc Fischlin and Jean-Sébastien Coron. Springer Berlin Heidelberg, 2016, pp. 622–643 (cit. on pp. xii, 5).

[App11] Andrew W. Appel. "Verified Software Toolchain - (Invited Talk)". In: *ESOP.* Vol. 6602. Lecture Notes in Computer Science. Springer, 2011, pp. 1–17 (cit. on p. 29).

[App15] Andrew W Appel. "Verification of a cryptographic primitive: SHA-256". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37.2 (2015), p. 7 (cit. on p. 29).

[Arz+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '14. ACM, 2014, pp. 259–269 (cit. on p. 27).

[Ass+13] Mounir Assaf, Julien Signoles, Frédéric Tronel, and Eric Totel. “Program Transformation for Non-interference Verification on Programs with Pointers”. In: *28th Security and Privacy Protection in Information Processing Systems (SEC)*. Ed. by Lech J. Janczewski, Henry B. Wolfe, and Sujeet Shenoi. Vol. AICT-405. Security and Privacy Protection in Information Processing Systems. Springer Berlin Heidelberg, July 2013, pp. 231–244 (cit. on p. 99).

[Bar+14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. “System-level Non-interference for Constant-time Cryptography”. In: *ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1267–1279 (cit. on pp. 28, 53, 59, 72, 74).

[Bar+17] Gilles Barthe, Sandrine Blazy, Vincent Laporte, David Pichardie, and Alix Trieu. “Verified Translation Validation of Static Analyses”. In: *Computer Security Foundations Symposium*. 30th IEEE Computer Security Foundations Symposium. Aug. 2017 (cit. on pp. xiv, 9, 60).

[Bar15] Gilles Barthe. “High-assurance cryptography: Cryptographic software we can trust”. In: *IEEE Security & Privacy* 13.5 (2015), pp. 86–89 (cit. on p. 28).

[Ber+15] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. “Verified Correctness and Security of OpenSSL HMAC.” In: *USENIX Security Symposium*. 2015, pp. 207–221 (cit. on p. 29).

[Ber05a] Daniel J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. 2005 (cit. on pp. ix, 3).

[Ber05b] Daniel J. Bernstein. *qhasm*. https://cr.yp.to/qhasm.html. 2005 (cit. on p. 28).

[Ber06] Daniel J. Bernstein. “Curve25519: New Diffie-Hellman Speed Records”. In: *Public Key Cryptography - PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Springer Berlin Heidelberg, 2006, pp. 207–228 (cit. on p. 55).

[BGL18] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. “Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"”. In: *CSF*. IEEE Computer Society, 2018, pp. 328–343 (cit. on pp. 78, 94, 95, 98).

[Bih97] Eli Biham. “A Fast New DES Implementation in Software”. In: *Proceedings of the 4th International Workshop on Fast Software Encryption*. FSE ’97. Springer-Verlag, 1997, pp. 260–272 (cit. on p. 100).

[Bla+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. “A static analyzer for large safety-critical software”. In: *PLDI*. ACM, 2003, pp. 196–207 (cit. on p. 26).

[BLS12] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. “The security impact of a new cryptographic library”. In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2012, pp. 159–176 (cit. on pp. 53, 57, 73).

[Bon+17] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. “Vale: Verifying high-performance cryptographic assembly code”. In: *Proceedings of the USENIX Security Symposium*. 2017 (cit. on p. 28).

[Bos+15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. “Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem”. In: *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 2015, pp. 553–570 (cit. on pp. 53, 55).

[BPT17] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying constant-time implementations by abstract interpretation”. In: *European Symposium on Research in Computer Security*. Springer. 2017, pp. 260–277 (cit. on pp. xiv, 8, 31).

[BPT18] Sandrine Blazy, David Pichardie, and Alix Trieu. “Verifying Constant-Time Implementations by Abstract Interpretation (Extended version)”. In: *Journal of Computer Security* (2018). (Accepted for publication, to appear) (cit. on pp. xiv, 8, 31).

[Cau+17] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. “FaCT: A Flexible, Constant-Time Programming Language”. In: *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE. 2017, pp. 69–76 (cit. on p. 29).

[CC76] P. Cousot and R. Cousot. “Static determination of dynamic properties of programs”. In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130 (cit. on p. 47).

[Cop+09] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. “Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors”. In: *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 2009, pp. 45–60 (cit. on p. 100).

[DB08] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on p. 28).

[DMW17] Florian Dewald, Heiko Mantel, and Alexandra Weber. "AVR Processors as a Platform for Language-Based Security". In: *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I.* 2017, pp. 427–445 (cit. on p. 101).

[EA07] U. Erlingsson and M. Abadi. *Operating system protection against side-channel attacks that exploit memory latency.* Tech. rep. MSR-TR-2007-117. Microsoft Research, 2007 (cit. on p. 72).

[Enc+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* OSDI'10. USENIX Association, 2010, pp. 393–407 (cit. on p. 27).

[Erb+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises". In: *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19).* May 2019 (cit. on p. 29).

[Flo67] Robert W. Floyd. "Assigning Meanings to Programs". In: 19 (Jan. 1967) (cit. on p. 14).

[Geu09] H. Geuvers. "Proof assistants: History, ideas and future". In: *Sadhana* 34.1 (Feb. 2009), pp. 3–25 (cit. on p. 12).

[GM82] J. A. Goguen and J. Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy.* Apr. 1982, pp. 11–11 (cit. on p. 26).

[Gon+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. "A Machine-Checked Proof of the Odd Order Theorem". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings.* 2013, pp. 163–179 (cit. on p. 6).

[Hoa69] Charles Antony Richard Hoare. "An axiomatic basis for computer programming". In: *Communications of the ACM* 12.10 (1969), pp. 576–580 (cit. on p. 14).

[HS06] Sebastian Hunt and David Sands. "On Flow-sensitive Security Types". In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '06. ACM, 2006, pp. 79–90 (cit. on p. 27).

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private circuits: Securing hardware against probing attacks". In: *Annual International Cryptology Conference*. Springer. 2003, pp. 463–481 (cit. on p. 102).

[Jou+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. "A Formally-Verified C Static Analyzer". In: *Proc. of the 42th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. ACM, 2015, pp. 247–259 (cit. on pp. 7, 25, 58).

[Jou16] Jacques-Henri Jourdan. "Verasco: a Formally Verified C Static Analyzer". Ph.D Thesis. Université Paris Diderot-Paris VII, May 2016 (cit. on p. 25).

[Kau+16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. "When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015". In: *Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Springer International Publishing, 2016, pp. 573–582 (cit. on p. 78).

[Koc+11] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. "Introduction to differential power analysis". In: *Journal of Cryptographic Engineering* 1.1 (Apr. 2011), pp. 5–27 (cit. on pp. viii, ix, 2, 3).

[Koc96] Paul Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology — CRYPTO '96*. Ed. by Springer. Vol. 1109. LNCS. 1996, pp. 104–113 (cit. on pp. ix, 3).

[KPM12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. "STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud". In: *USENIX Security 2012*. USENIX Association, 2012, pp. 11–11 (cit. on p. 73).

[Lan08] Adam Langley. *donna*. https://code.google.com/archive/p/curve25519-donna. 2008 (cit. on pp. 53, 55).

[Lap15] Vincent Laporte. "Verified static analyzes for low-level languages". Ph.D Thesis. Université Rennes 1, Nov. 2015 (cit. on p. 25).

[Lei08] Rustan Leino. "This is Boogie 2". In: Microsoft Research, June 2008 (cit. on p. 28).

[Lei10] K Rustan M Leino. "Dafny: An automatic program verifier for functional correctness". In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2010, pp. 348–370 (cit. on p. 28).

[Ler06] Xavier Leroy. “Formal certification of a compiler back-end, or: programming a compiler with a proof assistant”. In: *33rd symposium Principles of Programming Languages*. ACM Press, 2006, pp. 42–54 (cit. on pp. xiii, 6).

[Ler09] Xavier Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446 (cit. on p. 15).

[Li+17] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. “Static analysis of android apps: A systematic literature review”. In: *Information and Software Technology* 88 (2017), pp. 67–95 (cit. on p. 27).

[Man+18] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. “How Secure Is Green IT? The Case of Software-Based Energy Side Channels”. In: *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*. 2018, pp. 218–239 (cit. on p. 102).

[mbe14] mbedTLS. *mbedTLS (formerly known as PolarSSL)*. https://tls.mbed.org/. 2014 (cit. on pp. 53, 57, 73).

[Mer+18] Darius Mercadier, Pierre-Évariste Dagand, Lionel Lacassagne, and Gilles Muller. “Usuba, Optimizing & Trustworthy Bitslicing Compiler”. In: *Workshop on Programming Models for SIMD/Vector Processing*. Feb. 2018 (cit. on p. 101).

[Mol+06] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. “The Program Counter Security Model: Automatic Detection and Removal of Control-flow Side Channel Attacks”. In: *Proceedings of the 8th International Conference on Information Security and Cryptology*. ICISC’05. Springer-Verlag, 2006, pp. 156–168 (cit. on p. 100).

[Plo81] Gordon D Plotkin. “A structural approach to operational semantics”. In: (1981) (cit. on p. 14).

[Por16] Thomas Pornin. *BearSSL*. https://www.bearssl.org/. 2016 (cit. on p. 57).

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. “Dude, is my code constant time”. In: *Proc. of DATE 2017*. 2017 (cit. on p. 53).

[Ric53] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on p. 7).

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. “All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)”. In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP ’10. IEEE Computer Society, 2010, pp. 317–331 (cit. on p. 27).

[Saf16] Open Quantum Safe. *Open Quantum Safe*. https://openquantumsafe.org/. 2016 (cit. on p. 55).

[Sch+16] Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. “Let’s Face It: Faceted Values for Taint Tracking”. In: *Computer Security – ESORICS 2016*. Ed. by Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows. Springer International Publishing, 2016, pp. 561–580 (cit. on p. 27).

[SS71] Dana Scott and Christopher Strachey. *Towards a Mathematical Semantics for Computer Languages*. 1971 (cit. on p. 14).

[Swa+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. “Secure distributed programming with value-dependent types”. In: *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*. Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy. ACM, 2011, pp. 266–278 (cit. on p. 28).

[TIS16] TIS-CT. *TIS-CT*. http://trust-in-soft.com/tis-ct/. 2016 (cit. on p. 53).

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. “A Sound Type System for Secure Flow Analysis”. In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187 (cit. on pp. 26, 27).

[VS97] Dennis Volpano and Geoffrey Smith. “Eliminating Covert Flows with Minimum Typings”. In: *Proceedings of the 10th IEEE Workshop on Computer Security Foundations*. CSFW ’97. IEEE Computer Society, 1997, pp. 156– (cit. on p. 27).

[WN95] David J. Wheeler and Roger M. Needham. “TEA, a tiny encryption algorithm”. In: *Fast Software Encryption: Second International Workshop Leuven, Belgium, December 14–16, 1994 Proceedings*. Ed. by Bart Preneel. Springer Berlin Heidelberg, 1995, pp. 363–366 (cit. on p. 55).

[Yan+11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. ACM, 2011, pp. 283–294 (cit. on pp. xiii, 6).

[Ye+17] Katherine Q. Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W. Appel. "Verified Correctness and Security of mbedTLS HMAC-DRBG". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017, pp. 2007–2020 (cit. on p. 29).

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: a timing attack on OpenSSL constant-time RSA". In: *Journal of Cryptographic Engineering* 7.2 (June 2017), pp. 99–112 (cit. on pp. ix, 3).

[Zin+17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. "HACL*: A Verified Modern Cryptographic Library". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017, pp. 1789–1806 (cit. on p. 28).

[ZS18] M. Zhao and G. E. Suh. "FPGA-Based Remote Power Side-Channel Attacks". In: *2018 IEEE Symposium on Security and Privacy (SP)*. Vol. 00. 2018, pp. 839–854. DOI: `10.1109/SP.2018.00049` (cit. on pp. x, 5, 101).

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre: Vérification d'implémentations constant-time dans une chaîne de compilation vérifiée

Mot clés : Vérification formelle, compilation, canaux cachés, Coq, CompCert, Verasco, constant-time, analyse statique

Resumé : Les attaques par canaux cachés sont une forme d'attaque particulièrement dangereuse. Dans cette thèse, nous nous intéressons au canal caché temporel. Un programme est dit "constant-time" lorsqu'il n'est pas vulnérable aux attaques par canal caché temporel. Nous présentons dans ce manuscrit deux méthodes reposant sur l'analyse statique afin de s'assurer qu'un programme est constant-time. Ces méthodes se placent dans le cadre de vérification formelle afin d'obtenir le plus haut niveau d'assurance possible en s'appuyant sur une chaîne de compilation vérifiée composée du compilateur CompCert et de l'analyseur statique Verasco. Nous proposons aussi une méthode de preuve afin de s'assurer qu'un compilateur préserve la propriété de constant-time lors de la compilation d'un programme.

Title: Verifying constant-time implementations in a verified compilation toolchain

Keywords : Formal verification, compilation, side-channels, Coq, CompCert, Verasco, constant-time, static analysis

Abstract : Side-channel attacks are an especially dangerous form of attack. In this thesis, we focus on the timing side-channel. A program is said to be constant-time if it is not vulnerable to timing attacks. We present in this thesis two methods relying on static analysis in order to ensure that a program is constant-time. These methods use formal verification in order to gain the highest possible level of assurance by relying on a verified compilation toolchain made up of the CompCert compiler and the Verasco static analyzer. We also propose a proof methodology in order to ensure that a compiler preserves constant-time security during compilation.