Secure compilation for memory protection

Alexandre Dang

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *(voir liste des spécialités)*

Par

Alexandre DANG

Compilation Sécurisée pour la Protection de la Mémoire

Thèse présentée et soutenue à Rennes, le 10 Décembre 2020
Unité de recherche : Celtique
Thèse N° :

Rapporteurs avant soutenance :

Tamara Rezk	Inria Sofia Antipolis
Alejandro Russo	Chalmers University of Technology

Composition du Jury :

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

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Frédéric Besson	Inria Rennes
	Tamara Rezk	Inria Sofia Antipolis
	Alejandro Russo	Chalmers University of Technology
	Heydemann Karine	Université Pierre et Marie Curie
	Viet Triem Tong Valérie	CentraleSupélec
Dir. de thèse :	Thomas JENSEN	Inria Rennes

COMPILATION POUR LA PROTECTION DE LA MÉMOIRE

Nos vies sont de plus en plus dépendantes des systèmes informatiques et cette tendance devrait s'accentuer dans les années à venir. Que ce soit dans la santé, les transports ou l'économie, ces différents domaines ne sont plus capables de fonctionner correctement sans l'utilisation de ces systèmes. De ce fait, l'intérêt porté sur la sécurité informatique a également pris de l'importance, en conséquence des retombées désastreuses que peuvent provoquer les cyber attaques.

Cette thèse a pour but d'améliorer la sécurité des systèmes informatiques en s'intéressant à la compilation sécurisée des programmes. La compilation est le processus de traduction d'un programme source écrit par des humains vers du code machine lisible par nos processeurs. Il existe deux manière de faire de la compilation sécurisée: par application ou par préservation d'une politique de sécurité. Appliquer une politique de sécurité à un programme durant la compilation consiste à prendre n'importe quel programme en entrée et de toujours produire un programme sécurisé en sortie. Un tel exemple est le compilateur gcc patché avec StackGuard [29] qui produit des exécutables résistants aux attaques de type *buffer overflow* [70]. Dans ce cas précis, la politique de sécurité appliquée est choisie par le compilateur ce qui peut poser problème lorsqu'elle diffère du choix de l'utilisateur. Un compilateur préservant la sécurité des programmes suit un objectif différent, celui-ci s'assure que le programme compilé soit au moins aussi sécurisé que le programme source. Par exemple, Jasmin [3] est un compilateur dédié aux implémentations cryptographiques et garantit qu'un programme compilé respecte les mêmes propriétés de sécurité que sa version source. Le choix de compilateur sécurisé dépend du degré de confiance accordé au programme source qui va être compilé. Dans le cas où le programme source est d'origine inconnue ou malveillante, on utilisera l'application de sécurité pour être sûr d'obtenir un programme sécurisé en sortie. Au contraire, lorsque la politique de sécurité est mise en place par les développeurs dans le code source d'un programme, on utilisera plutôt à un compilateur qui s'assurera que la sécurité du programme source soit préservée jusqu'au

programme exécutable.

Les deux types de compilation sécurisée présentés seront abordés dans cette thèse. Tout d'abord nous avons développé le compilateur CompCertSFI qui applique les techniques de *Software Fault Isolation* (SFI) sur les programmes transformés. La deuxième partie de la thèse se concentre sur la préservation de propriétés de sécurité durant la compilation. Nous avons défini une propriété appelée *Information Flow Preserving* qui garantit qu'une transformation produise des programmes aussi sécurisés que que leurs version source.

CompCertSFI

Software Fault Isolation (SFI) est une technique d'isolation qui essaye de maximiser les temps d'exécution. Les principes fondamentaux de SFI ont été établis par Wahbe *et al.* [114] et les travaux qui suivent tels que NaCl [117] ou Pittsfield [73] sont tous basés sur ces principes. Nous commençons par présenter les fondamentaux de SFI pour ensuite nous pencher sur notre compilateur CompCertSFI.

Software Fault Isolation (SFI) Différentes méthode d'isolation existent comme les machines virtuelles ou la mémoire virtuelle entre processus. SFI est la technique qui réduit au maximum les interactions entre un programme hôte et différents modules qui peuvent être malveillants. Pour cela, dans SFI, le programme hôte accueille dans sa propre mémoire, les différents modules avec lesquels il veut interagir. Pour éviter que ces modules corrompent la mémoire du programme hôte, ils sont placés dans des zones mémoires attitrées, appelées *sandbox*. La politique de sécurité de SFI décrite par Wahbe *et al.* est la suivante:

— **Sûreté de la mémoire**: un module ne peut ni lire, ni écrire ni sauter hors de sa sandbox.

— **Communication sécurisée**: toute communication entre l'hôte et ses modules passent à travers une interface qui vérifie leur légitimité.

— **Vérification du code**: tout code exécuté par un module a été au préalable contrôlé de manière à ce qu'il satisfasse les deux règles précédentes

SFI est composée de deux éléments majeurs: un générateur et un vérifieur. Le générateur est intégré durant la compilation et transforme le code d'un module de manière à

ce qu'il satisfasse les règles de sûreté de la mémoire et de communication sécurisée. Ensuite, avant de charger le code d'un module en mémoire, il sera contrôlé par le vérifieur qui s'assure que le code respecte la politique de sécurité de SFI.

De nombreux travaux ont été effectué sur SFI tels que MiSFIt [107], Pittsfield [73] ou NaCl [117, 98, 6] qui a été utilisé dans le navigateur Google Chrome. Le principal but de ces travaux a été de diminuer au maximum, les ralentissements introduits par les transformations de programme de SFI.

COMPCERTSFI COMPCERTSFI est un compilateur basé sur COMPCERT [66] qui produit des programmes respectant la politique de sécurité de SFI. COMPCERT est un compilateur certifié, pour le langage C, prouvé avec l'assistant de preuve Coq [112], de respecter un théorème de préservation des programmes compilés. Grâce à ce théorème de préservation, COMPCERTSFI peut retirer le vérifieur SFI des outils de SFI. En effet, COMPCERTSFI possède seulement un générateur de code pour SFI mais pas de vérifieur. Cette approche a d'abord été théorisé par PSFI [60] et COMPCERTSFI en est une implémentation compétitive et complète. L'idée est d'utiliser un générateur SFI assez tôt dans la chaîne de compilation, puis de profiter du théorème de préservation de COMPCERT pour préserver la sécurité de SFI dans le binaire produit. Les théorèmes prouvés sur le générateur SFI de COMPCERTSFI sont les suivants:

— *Sécurité*: le code produit par le générateur ne peut accéder à de la mémoire situées en dehors de sa sandbox et toute interaction hôte-module est identifiée et contrôlée par une bibliothèque de confiance.

— *Sûreté*: le code produit ne contient pas de comportements indéfinis

Le théorème de sécurité correspond aux règles de sûreté de la mémoire et de communication sécurisée des principes de SFI et garantissent l'isolation des modules. Le théorème de sûreté est nécessaire pour pouvoir appliquer le théorème de préservation de COMPCERT qui cite: si un programme p ne contient pas de comportements indéfinis alors le programme compilé p' a le même comportement que p. Grâce à ce théorème de préservation, nous avons la garantie que les transformations faites auparavant par le générateur SFI sont conservées durant la compilation et que le programme compilé est bien isolé dans sa sandbox.

Ne pas posséder de vérifieur donne plusieurs avantages. Tout d'abord, puisque les transformations SFI se font à haut niveau, COMPCERTSFI supporte naturellement

toutes les architectures supportés par CompCert. Avec l'approche traditionnelle de SFI, nous sommes obligés de développer un générateur et un vérifieur pour chaque architecture. Un autre avantage est que les transformations SFI se font à haut niveau ce qui permet aux constructions SFI d'être optimisés par le compilateur. En effet, dans l'approche classique le vérifieur doit être capable de contrôler que du code respecte les principes de SFI et doit être donc facilement vérifiable. De ce fait, les transformations SFI ne sont pas optimisés afin d'être visible par le vérifieur SFI.

Nous avons effectué des mesures des performances de CompCertSFI et les avons comparés avec NaCl. En moyenne, notre compilateur a des performances similaires à NaCl et nous avons observé que les différences résident principalement dans les performances du compilateur CompCert pour les temps d'exécution. En effet, CompCert est plus lent que gcc ou Clang, utilisés par NaCl, et nos expériences montrent que l'impact de nos transformations SFI est modeste. Plusieurs pistes sont possibles pour améliorer CompCertSFI. La première est de tout simplement améliorer les performance de CompCert en utilisant des variantes plus rapides telle que CompCertSSA [10] qui possède des optimisations supplémentaires. Une autre idée est de créer des optimisations spécifiques à SFI pour diminuer encore plus le ralentissement provoqué par les constructions SFI rajoutées dans le code.

Préservation du flot d'information

Dans la seconde partie de la thèse nous nous intéressons à la préservation de la sécurité par les compilateurs. Nous expliquons tout d'abord pourquoi ce n'est pas le cas avec les compilateurs actuels, et nous présentons ensuite nos travaux qui permettent de pallier à ce problème.

Pourquoi les compilateurs ne préservent ils pas la sécurité? Une personne non initiée à la sécurité informatique pourrait être surprise qu'un compilateur ne préserve pas la sécurité des programmes. En pratique, c'est un problème récurent qui ne possède pas de solutions adéquates aujourd'hui. Par exemple, certaines instructions qui servent à effacer des secrets de la mémoire ne sont pas comprises par le compilateur et peuvent être retiré par ce dernier pour améliorer les performances du programme [116, 104]. Cela introduit alors une faille de sécurité dans le programme compilé malgré le sécurité du programme source.

Les compilateurs sont conçus pour préserver le comportement des programmes durant les différentes transformations. Ces comportement sont définis par les standards des langages sources utilisés. Cependant, qu'en est-il des comportements qui ne peuvent être exprimés par ces standards? Par exemple, l'instruction `x = 0` peut être utilisé pour effacer un secret qui était stocké dans `x`. Pour le standard d'un langage, cette instruction signifie juste que la variable `x` doit contenir la valeur `0` mais rien n'est précisé sur l'effacement de l'ancienne valeur. De ce fait, le compilateur peut mal interprété l'intention du développeur et introduire des failles de sécurité dans les programmes.

Ce phénomène est encore plus accentué aujourd'hui avec l'apparition de nouvelles attaques de type *canaux cachés*. Ces attaques utilisent le temps d'exécution [55], la consommation d'énergie [56] ou d'autres médiums physiques pour attaquer les systèmes informatiques. Ces comportements physiques ne sont pas définis dans les standards de langage et ne sont donc pas modélisés dans les programmes sources. De ce fait, même si un code est écrit de manière à s'exécuter en temps constant, rien ne garanti que ça soit le cas avec le code compilé.

Dans cette thèse, nous nous focalisons sur cette problématique et nous proposons une propriété sur les transformations de programme qui s'assurent qu'un programme transformé soit au moins aussi sécurisé que sa version source.

Information Flow Preserving (IFP) transformation Nous avons défini une propriété de sécurité appelée IFP, qui s'assure qu'un programme transformé soit au moins aussi sécurisé que sa version source, contre un attaquant capable d'observer la mémoire de manière arbitraire, durant une exécution. Ce modèle d'attaquant est générique et peut facilement être utilisé pour modéliser des attaques de type canaux cachés. En plus de cette définition, nous proposons une technique de preuve pour prouver qu'une transformation est IFP. Des algorithmes sécurisé pour les passes de compilation *Dead Store Elimination* (DSE) et *Register Allocation* (RA) sont également présentés, dans lesquels, nous utilisons notre technique de preuve pour montrer qu'elles sont IFP. Un allocateur de registre utilisant notre algorithme a aussi été implémenté, et testé pour évaluer le coût de notre sécurité sur les performances des programmes.

Nous modélisons un attaquant capable de sonder l'intégralité de la mémoire durant l'exécution d'un programme. Le but de notre propriété est de s'assurer qu'un tel attaquant n'est pas capable d'apprendre plus d'information en observant une exécu-

tion du programme transformé que du programme source. Pour pouvoir catégoriser les fuites d'information en plusieurs niveaux, nous rajoutons des *attaquants partiels* qui sont seulement capable d'observer un nombre limité de bits dans la mémoire. Notre propriété de sécurité cite qu'une transformation est IFP si: pour n'importe quel attaquant partiel, pour chaque observation possible d'une exécution du programme transformé, il existe une observation sur l'exécution du programme source qui donne au moins autant d'informations sur les données utilisé par les programmes. Cette propriété nous assure que pour toute information qu'on peut trouver dans le programme transformé alors cette information est aussi disponible dans le programme source.

De ce fait, nous avons trouver une condition suffisante pour prouver notre propriété de transformation IFP basée sur le même principe. En effet, l'idée est que, si nous sommes capables de corréler chaque emplacement mémoire du programme transformé à un emplacement mémoire du programme source de manière à que ces emplacements contiennent les mêmes valeurs; alors la transformation est IFP. Cette condition est basée sur le fait que toute information disponible dans le programme transformé doit aussi être disponible dans le source.

Finalement, nous proposons des versions sécurisées de DSE et RA et prouvons qu'elles sont IFP en montrant qu'il est toujours possible de créer cette corrélation entre emplacements mémoire depuis le programme transformé vers le source. Les expériences nous montrent que notre allocateur de registre sécurisé apporte un ralentissement modeste sur les temps d'exécution (15% dans le pire des cas) ce qui conforte la viabilité de notre approche. Le but final de notre travail est d'avoir un compilateur IFP complet et les futurs efforts doivent se concentrer à prouver notre propriété sur toutes les passes de compilation.

TABLE OF CONTENTS

Introduction **11**

1 Background **19**
1.1 Memory threats . . . 19
1.1.1 Common attacks against memory . . . 19
1.1.2 Side-channels attacks . . . 22
1.2 CompCert . . . 25

2 Protecting Memory from External Modules **29**
2.1 *Software Fault Isolation (SFI)* . . . 30
2.1.1 Principles of SFI . . . 31
2.1.2 Adapting and extending SFI to other architectures . . . 38
2.1.3 Extending SFI . . . 42
2.1.4 Optimisations . . . 45
2.1.5 Register management . . . 47
2.1.6 Range analysis . . . 49
2.1.7 Verification techniques . . . 51
2.1.8 Conclusion . . . 54
2.2 CompCertSfi . . . 57
2.2.1 Background . . . 60
2.2.2 A Thread-aware Sandbox . . . 64
2.2.3 Memory-safe Masking . . . 66
2.2.4 Enforcement of Control-Flow Integrity . . . 70
2.2.5 Safety and Security Proofs . . . 72
2.2.6 SFI Runtime and Library . . . 75
2.2.7 Experiments . . . 77
2.2.8 Related Work . . . 83
2.2.9 Conclusion . . . 84

3 Protecting Memory against Probing Attacks 87
3.1 Motivations . . . 88
3.1.1 Memory probing attacker . . . 88
3.1.2 Information Flow Preservation by Examples . . . 89
3.1.3 The “Full Information Flow” Paradox . . . 91
3.1.4 Some security properties against information leakage . . . 92
3.2 Related works . . . 96
3.2.1 Secure refinement . . . 96
3.2.2 Power Analysis . . . 97
3.2.3 Preservation of Constant-Time implementations . . . 98
3.2.4 Information-Flow preservation . . . 100
3.3 Information-Flow Preserving Transformations . . . 100
3.3.1 Information-Flow Preservation . . . 101
3.3.2 Sufficient Condition for IFP and its Proof Principle . . . 107
3.3.3 Securing Dead Store Elimination . . . 110
3.3.4 Translation Validation for Register Allocation . . . 113
3.3.5 Experiments . . . 125
3.3.6 Results and analysis . . . 125
3.4 Discussion and Perspectives . . . 127
3.4.1 Which properties can I preserve? . . . 128
3.4.2 Securing others compiler passes . . . 131
3.4.3 Observation points . . . 132

Conclusion 134

Bibliography 141

INTRODUCTION

The first full-fledged compiler was written in 1957 for the FORTRAN language. Compilers transform programs written in high-level languages into machine code. These high-level languages such as Python, C or Java use abstractions that mimic the behaviour of actual machines. This has the first advantage to abstract away low-level details which improves clarity and readability of the code and let developers focus on programs logic. Another good point of high-level languages is that programs generally get to be portable across multiple architectures. FORTRAN compiler written in 1957 also included the first compiler optimisations to produce more efficient machine code. Since then, compilers optimisations have improved by leaps and bounds such that they have become compulsory in our systems. However, issues arose with the popularization of the C language which contains numerous undefined behaviours [115] [62]. Indeed, instructions like division by zero are categorised as undefined behaviours by the C standard and choice of implementation is left to the compilers. While this gives compiled C programs better speed, it can also lead to unexpected program behaviours [96]. A field of research called *Certified Compilation* tackles such issue and focuses on the semantics preservation of programs during compilation. Successful projects such as CompCert [66], Vellvm [119] or CakeML [61] were born but semantics preservation was soon found to be lacking when it comes to security [31]. It may come as a surprise for non-cybersecurity specialist, but compilers are known to be unreliable on this aspect [32]. Surely, with the growing importance of security in our computer systems, secure compilers have become an active field of research during the recent years. This thesis focuses on this topic and shows two aspects of secure compilation. The first one is *enforcement* of security properties by the compilers which produces secure executables regardless of the source program. The second is *preservation* of security properties of programs during compilation which makes sure that transformed programs are at least as secure as their source programs.

Current State of Compilation

Compilers are tools that transform programs written in high-level languages down to binaries readable by a computer. Compiled languages (C, Rust) are often opposed to interpreted languages (JavaScript, Python). Multiple differences between these two categories of languages exist even though the distinction is not so clear nowadays with the usage of *Just In Time* (JIT) compilers which compiles code at runtime.

— *Self-contained*, compilers produce executable files which can be read directly by the machine, whereas interpreted programs need the corresponding interpreter to run.

— *Efficiency*, compiled programs run generally faster than interpreted programs. Compilers have the advantages to have more time to analyse the source program, which enables optimisations while interpreters only run program's code line by line. This difference is blurred by the use of JIT compilers which can do optimisations at runtime.

— *Portability*, compilers were used for portability in the past where most programming languages were architecture-dependent. Nowadays while compilers need to support multiple architecture this is also true for interpreters.

Nowadays numerous compilers exist and each puts the emphasis on different aspects of the code. Production of efficient compiled code had been an active field of development for compilers since its creation. With the rise of portable devices and *Internet of Things*, minimization of memory and power consumption have also become highly desirable criteria. Actually, the optimizing compilers GCC and Clang, mostly known for compiling C, possess numerous options to customize the compilation of programs. For instance, one can choose to prioritize speed, size of the code or debugging experience depending on the context.

Another type of compilers called *Just In Time* (JIT) compilers brings the benefits of compilation to interpreted language. A main advantage is that JIT compilers can detect parts of a program which are executed intensively and optimize this hot code. Most famous JIT compilers are used in browsers to speed up the execution of JavaScript like Chrome V8 for Google Chrome or SpiderMonkey for Firefox.

Certified compilers

Another branch of compilation uses formal methods to prove the semantics preservation of compilers. These formal methods are used to prove a semantics preservation theorem for the different compiler transformations. The theorem states that the behaviour of the transformed program should be included in the source program possible behaviours.

Formal methods The most common way to see if a program is working as it is supposed to be is through program testing. The program will be executed with different inputs and the tests check if the outputs have the expected value. An issue of testing is that it is not possible to test all the possible inputs. Formal methods are another technique that can be used in addition to testing, to increase the trust we place in a program's correctness.

Formal methods use mathematical techniques to improve the conception and verification of programs in accordance to their specifications. Specifications in the industry are usually expressed in natural languages, like English, where words or expressions may have multiple meanings. Therefore, specifications describing the functionality of a program may be interpreted differently depending on the person or context. To certify the correctness of a program, formal methods require a description of its behaviour using precise mathematical terms. With a mathematical language, the specified behaviours are described such that there is a unique possible interpretation of the different rules. Using formal methods, one can have stronger guarantees that a system design matches the requirements imposed. For example, when creating security protocols, certain properties like confidentiality of the data or authentication of the parties involved must be enforced. Thus, using a formal description of a protocol, one can make a proof that the design logic will ensure the desired security properties. Such techniques are used extensively to secure the future protocols like 5G [13], electronic voting systems [28]. . . Other than improving the designs of systems, formal methods can also be used to prove that the implementation of a system respects its formal specification. For the line 14 of Paris subway, B-method was used to minimize the gap between the specification and the concrete system. Proofs of formal methods can be written by hand and checked by humans but there also exist proof assistants, like Coq [15] or Isabelle [83], to write machine-checked proofs that offer higher trust.

Semantic Preservation Certified compilers are proven to respect a semantic correctness property which says that compiled programs behave as intended by the source program according to the language specifications. A core goal of compilers is to ensure that source languages abstractions are preserved down to the binary code. Nevertheless, some compiler transformations alter the behaviour of the source program accidentally or for various reasons like speed. This may introduce bugs in compiled programs leading to unexpected results and even security vulnerabilities when executing the binaries [115] [62]. To get stronger guarantees about semantic correctness, formal methods have been used for compilers by the research community.
In the case of certified compilers, formal methods define with mathematical terms the semantic correctness property described previously: the compiled program behaves as the source program according to the languages specifications. Secondly, the developed compiler will be proven to satisfy the semantic correctness property. Examples of certified compilers include CompCert [66], Vellvm [119] or CakeML [61] which are successful projects created over the past years. CompCert is used repeatedly in this thesis and is thoroughly presented in Section 1.2. Certified compilers preserve the behaviour of programs described by the source language standards. However, programs run on concrete machine which are abstracted in language standards. This gap between high-level abstractions and concrete machine make certified compilers unable to preserve certain security properties [31] which are related to low-level details of the machine.

Why do compilers not preserve security?

Certified compilers ensure that the compiled programs behave as intended by the source programs according to the high-level language specifications. Then what about concrete behaviours which cannot be expressed in the high-level language? Power consumption, execution time or cache behaviour are not defined in the abstractions of language standards. A new category of attacks called *side-channel attacks* [55] use physical behaviours of computer systems to perform their deeds. Protections against these attacks minimize the leakage from these physical channels. Thus, these security mechanisms cannot be understood by standard compilers, and may be removed or compromised during the program transformations. We illustrate this shortcoming with the example of Figure 1.

The case of DSE On the left side a toy cypher function called `crypt` computes a cypher `c` from a key `key` and some text `t`. Attacks like data remanence are able to probe into systems memory and steal sensitive data. To mitigate this kind of attack it is good practice to reduce as much as possible the lifespan of sensitive variables like `key` in our program. Line 3, `key` is set to `0` in order to erase its value from the memory. Thus, an attacker which would be able to probe the memory at the end of `crypt` would have no way to get the key of the program. Unfortunately a compilation pass called *Dead Store Elimination* (DSE) is known to break this kind of security mechanism [116][30][2]. For the compiler, the erasure instruction `key = 0` has no utility in the computation of the cypher `c`. Therefore, to improve the performance of the compiled code, DSE will optimize away the erasure instruction and produce the program on the right-side. In this situation, the compiler introduced a vulnerability in the compiled code due to not understanding the purpose of the erasure instruction.

Occurrences of such cases can be easily found by browsing through compiler projects [85] or crypto libraries [91]. Multiple workarounds exist to avoid having erasure instruction removed, however those are either constrained to some special cases or specific to a compiler thus not widely adopted.

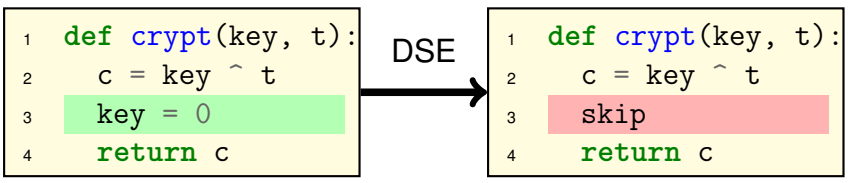


Figure 1 – *Dead Store Elimination* introduces vulnerabilities

Examples of vulnerabilities introduced by the compiler is not limited to DSE [108] [113]. Common programming languages do not have notions of time or power consumption in their specifications. This implies that any countermeasures against these side-channels are difficultly understood by compilers and might be compromised by the different compilation passes. This is not acceptable when one is running code manipulating sensitive data. Compilers, when used for critical code, should be able to preserve the security properties that were enforced at the source level. The principle behind preserving security properties is to make sure that the compiled program is no less secure than the source one.

Compilation for Security

Programs lifecycle can be divided into multiple layers from software to execution. In our situation where we focus on compilation we will consider: development of the software, compilation and execution on the hardware. Numerous layers can be included like operating systems or virtual machine, but for the simplicity of our argument we restrain ourselves to these three steps. A general question when doing security is to ask ourselves at which layer should one enforce security of programs. The semantic of a program depends on the layer of abstraction we are working on. At source level, the semantic of a program is defined by the standards of the source language used. The compiler translates this program until obtaining a binary program which semantic is defined by the processor targeted. The earlier in the program lifecycle the more abstract is the view of a program. Checking and maintaining security on abstract representations of programs has several advantages. First, abstractions of programs are in general more understandable which allow us to define clearer and more readable security policies. Second, it will also be more practical to enforce these security policies using program analysis on the abstractions. Lastly, the abstractions are in general portable to multiple architectures, meaning that the analysis are reusable for multiple implementations. On the other hand, one must make sure that security properties which hold at the abstract level still hold at the concrete level. Furthermore, certain security properties cannot be expressed with abstractions and can only be implemented at lower levels. For example, it is not natural to write programs with low power consumption at source level since there is no notion of power consumption in high-level programming languages.

Hardware Security The end goal of security is to ensure that the security properties we want to enforce are effective when the programs are executed by the software. To be sure that we have secure executions of programs, a simple solution is to solely use security-dedicated hardware for critical code like secure enclave [100] or hardware hypervisors [39]. Another advantage is that it does not require developers to be able to produce secure source code. While this approach gives strong guarantees about security, the main downside of this idea is flexibility and practicality. Indeed, it takes a fairly long time to develop new hardware whereas new attacks are discovered every day. Therefore, in case of a zero-day attack, it is unsuitable to solely use hardware protections to defend our systems. Furthermore, with the popularization of smartphones and

Internet of Things, personal data are stored and used by numerous devices. Protecting all these devices using dedicated hardware does not sound reasonable or would be too costly.

Enforcement of Security during Compilation The idea here is to use compilation to make sure that transformed programs are secure. Regardless of the input programs the compiler will produce a program which satisfies a defined security policy. This is is more flexible than hardware security since we only need to update the compiler and recompile the programs to prevail against new attacks. Furthermore, we do not need to trust the source program to have security at a lower level. This is ideal in the situation where the source provider is untrusted but too strict in the case where we let the developers define their own security policy. For example, a source program may already be secure against buffer overflows but a secure compiler will still enforce its own countermeasures which add redundant protections. Examples of compilers enforcing security properties can easily be found. gcc can enforce security protections against buffer overflows by inserting *canaries* in the stack of the compiled program [29]. To mitigate the problem of erasure caused by DSE, Simon *et al.* [104] propose a variant of the Clang compiler that erases the registers and the stack frame after returning from a sensitive function. Enforcement of security properties is used in this thesis in Chapter 2 to ensure that compiled programs satisfy the *Software Fault Isolation* policy.

Preservation of Security during Compilation Preservation of security is most suited to the situation where the security of programs is devised by the developers This is the most flexible solution, since the developer can define precisely the security policy that it wants. Indeed, using a secure hardware (resp. compiler) only guarantees that the hardware (resp. compiler) security policy holds but it may not be the one desired by the developer. For example, let assume a password checker written such that the last character typed is visible to the user. A secure compiler may deem that this is a breach of security and change the password checker such that no characters are visible. While this is more secure, this is also not the intent of the developer which may be considered as a compiler bug. Several works focus on the preservation of security during compilation. Jasmin [3] is a low-level compiler which focuses on preserving the security of cryptographic implementations. Barthe *et al.* [11] present a compiler prototype to preserve constant-time properties across transformations. The issue we tackle in

Chapter 3 also corresponds to the preservation of security properties during program transformation.

Content of the thesis

This thesis is divided into three chapters. Additional background information is given in Chapter 1 where we first present multiple memory attacks that we want to address in Section 1.1. Then, since the certified compiler CompCert is used multiple times in the thesis, Section 1.2 gives an overview of CompCert.

Chapter 2 deals with *Software Fault Isolation* (SFI). SFI is an isolation technique which allows a program to safely host untrusted modules into its own memory by placing them into *sandboxes*. The principles and current techniques of SFI will first be detailed in Section 2.1. Then we will present our compiler CompCertSFI, which takes a C program as input and produces a binary which execution is confined into a sandbox. CompCertSFI has been derived from CompCert presented previously and has been proven to produce SFI secure programs.

Staying on the theme of compilation and security, Chapter 3 deals with the issue of preserving security of programs during compilation. In our work, we focus on attackers which are able to probe the memory during an execution (like power analysis attacks). We define a notion called *Information-Flow Preserving* (IFP) transformation which makes sure that a probing attacker cannot learn more information from the transformed program than from the source program. An implementation of an IFP *Register Allocation* is also presented to show the viability of our approach.

CHAPTER 1

BACKGROUND

1.1 Memory threats

In this thesis, we want to use compilation to prevail against attacks targeting the memory of programs. Such attacks can lead to privileged access on a server or leakage of sensitive information manipulated by the program. We divide these attacks into two categories: algorithmic and side-channel attacks. Algorithmic attacks such as *buffer overflow* target program vulnerabilities which allow an attacker to meddle with memory areas that should not be accessible. On the other hand, side-channel attacks target computer systems directly and use physical information such as time or power consumption to carry out the attack.

1.1.1 Common attacks against memory

In Chapter 2 we work with an isolation technique called *Software Fault Isolation* which protects a host program from being corrupted by external untrusted modules. The two vulnerabilities we present: *Buffer Overflow* and *Format String attacks* can be mitigated using such isolation techniques. These two attacks have been discovered more than twenty years ago but are still ongoing threats. They both appear in programs written with low-level programming languages like C/C++ or Assembly. In these low-level languages, multiple tasks are left to the developer like memory management or bound checking which may lead to serious vulnerabilities if done incorrectly.

1.1.1.1 Buffer overflow

Buffer overflows are one of the most common vulnerability used by malicious parties [70]. The vulnerability was first studied and published in 1972 by *Computer Security Technology Planning Study*. Despite knowing and dealing with this vulnerability

for more than forty years, attacks using buffer overflows are still used nowadays. For example, in 2014, OpenSSL was diagnosed with its most impactful security flaw called Heartbleed [36]. A buffer overflow vulnerability was detected in their implementation of the TLS protocol which could lead to leakage of private keys used during a communication. This would allow the attackers to decrypt the targeted exchange but also past messages of the involved parties. The typical case of buffer overflow is to access memory locations which are out of an array bounds. Attackers are then able to read or write on memory locations which should be not accessible. For example, *Return-Oriented Programming* (ROP) seeks to overwrite the return address stored in the stack of a function and is illustrated in Figure 1.1. An array is stored in the stack frame of a function. Using a buffer overflow, an attacker is able to overwrite memory which does not belong to the array. The attacker manages to reach a stack location containing a return address and overwrite it with an address of its choice. This is usually the address of some malicious code injected beforehand by the attacker called *payload*. The attacker successfully manages to execute its own code and gets control of the process. One

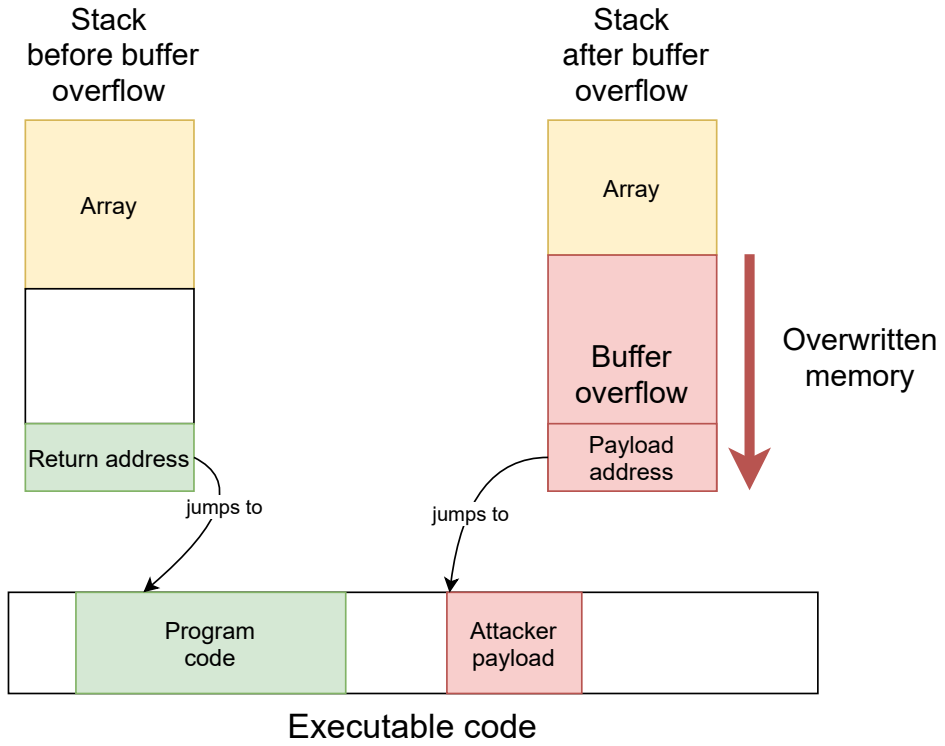


Figure 1.1 – ROP exploit

solution to mitigate ROP attacks is to make the stack or the heap non-executable. In

this situation the attacker is unable to leave a payload in the process memory and ROP attacks do not have a target to redirect the control-flow. A more advanced version bypasses this protection and is called *Return-To-Libc* [93]. Instead of jumping to a malicious payload, the overwritten return address points to instructions of the C standard library which is loaded in the executable memory. The usual target is the `system` libc function which can be used to execute shell commands. Multiple countermeasures exist such as protecting the stack by detecting any illicit overwriting of the stack using so-called stack *canaries* [29]. Another solution is to use *Address Space Layout Randomization*, which randomizes the memory layout at the creation of the process, making it more difficult, but not impossible, for an attacker to redirect the control-flow to its advantage.

1.1.1.2 Format string attack

Format string vulnerability was first discovered in 1989 [74] while fuzz testing UNIX operating systems but actual exploits did not occur before the early 2000s [23]. New vulnerabilities are still publicized by the MITRE in their CVE lists. While most of them are limited to leaking content of the memory, some exploits have been shown to give root access to the attacker [111]. The attack targets the C `printf`-family functions which have the peculiarity to be *variadic* functions, meaning that the number of parameters they expect is variable. An example is illustrated in Figure 1.2. `printf` role is to output

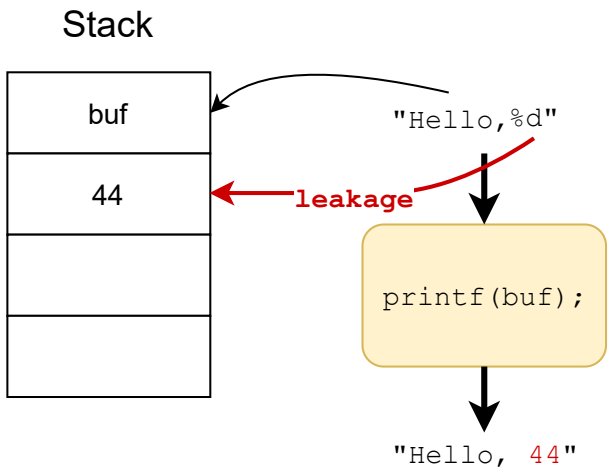


Figure 1.2 – Format String exploit

the string parameter given in the first position (`buf` in the example). If the string contains

a format (pattern starting with a %, %d in Figure 1.2), then `printf` expect additional parameters to replace the formats. Hence `printf` number of parameters depends on the number of formats found in the first string parameter. In our example, `printf` was called with a single argument, therefore the developer expected the users to give a string without any format. In the situation where a malicious user still gives a string containing a format such as “Hello, %d”, the program will look at the neighbouring memory for the non-existing additional parameters. Here the memory adjacent to `buf` contains the integer 44 which is a random value of the stack but will get printed on the screen. This way, a user can inspect the content of the memory using this format string vulnerability. Similar to buffer overflows, format string attacks can also alter the control-flow of the program by writing values on the return address using the %n format. Fortunately, most format string vulnerabilities can be detected using static analysis. However hard cases where the first string parameter of the `printf`-family function is generated at runtime are still difficult to prevent.

1.1.2 Side-channels attacks

Side-channel attacks have the particularity to target a characteristic of an implementation using physical mediums such as time, sound, power consumption... Paul Kocher first published an attack on cryptographic algorithms using *Timing Attacks* in 1996 [55] and then presented the concept of *Differential Power Analysis* in 1999 [56]. New channels are still discovered, recently, an attack using smartphones accelerometer to uncover the content of a discussion through the vibrations was published [90]. While many of these attacks may be unpractical to carry out, some of them are effectively used in the real world such as the infamous Meltdown [71] and Spectre [57] which use timing attacks to recover data leaked by the target. In this section, we present three side-channel attacks: cold boot, timing and power analysis attacks. Numerous other side-channels exist but these three have been chosen due to their closeness to the attacker model we use in Chapter 3.

1.1.2.1 Cold boot attacks

Contrary to common beliefs, after being shut down, RAM preserves the content of its memory for a period of time, which lasts from few seconds up to few minutes at low temperature. The goal of cold boot attacks is to reset a system and execute a

script that starts on boot which dumps the content of the RAM. This allows attackers to bypass any disk encryption systems by getting the cryptographic keys directly from the unprotected RAM. Even though warnings against this attack can be traced back from the 1990s, Halderman *et al.* [45] were the first to propose a practical way to mount cold boot attacks. Simmons [103] proposes a software system to store encryption keys in CPU registers instead of the RAM at the cost of significant overhead. Other solutions advise to have the BIOS directly clear the RAM on boot, to wipe RAM on sudden temperature drop or to lock the boot process. However, Gruhn and Müller [43] show that these solutions can be circumvented by transferring the RAM modules to another computer. Lastly, they also claim that, empirically, DDR3 RAM seems to be immune to this kind of attack.

1.1.2.2 Power Analysis

Power analysis uses the power consumption to track the behaviour of a program. This can be done easily by connecting an oscilloscope to the power cable of the targeted device. Similar to power analysis, electromagnetic analysis detects the magnetic waves produced by a device to perform an attack. These two side-channels follow the same principles and both the attacks and the mitigations are similar. Therefore, most of what we present on power analysis in this thesis is also applicable to electromagnetic analysis.

Simple Power Analysis (SPA) SPA monitors the variation in power consumption of a system caused by the different instructions that are executed. Since instructions have different power consumptions, attackers can infer the inner computations of a processor and guess the secret values used during the execution. Kocher *et al.* [56] advice to avoid using certain instructions which have distinct power consumptions like branching. Another solution is to use hard-wired implementations which have low power consumption variations. In general, the goal of these countermeasures is to either reduce the emission of useful signal or to increase the noise around to disturb the measurements.

Differential Power Analysis (DPA) DPA exploits the variations in power consumption of an execution caused by the value of certain bits during the executions. The influence of a bit on power consumption is small but using a statistical attack on nu-

merous executions, DPA is able to extract a fingerprint of whether a bit is set to 0 or 1. DPA is troublesome for security since popular countermeasures that seek to reduce the signal to noise ratio are not effective. Indeed, given a sufficient number of observations, an attacker is still able to differentiate the signal from the noise and get sensitive data. One of the most effective solution is to use masking [42] which split secrets into multiple random shares. Computations are then done independently on these shares. Since the shares are changed for every execution, random noise is added to the signal and prevent DPA. Masking is further detailed in the thesis in Section 3.1.4.2. A more advanced version of DPA exists called *Higher-Order Differential Power Analysis* which samples the power consumption multiple times within the same execution. The attack is difficult to perform but in theory an attacker can retrieve bits of information with a single execution.

1.1.2.3 Timing attacks

Timing is one of the first side-channel researched [55] and one the most infamous one. This is first caused by the existence of large scale attacks using such channel (Spectre [57] and Meltdown [71]) and also due to the practicality of the attack. The goal of the attackers is to correlate the timing behaviours of a program with the different inputs they have submitted. Using this correlation, an attacker retrieves information on the internal state of the program which can lead to data theft. A simple example is the password checker. A timing vulnerable password checker would give the result of the checking as soon as it finds a wrong digit or if all the digits are correct. Therefore, an attacker can use the variations in response time to have additional information on the correct password. A quick failure from the checker may corresponds to the first digit being false and a longer failure would correspond to a false latter digit. Therefore, the attacker is successful since it is able to reduce the cost of mounting an attack using the checker response time. A first solution to timing attacks was inputs blinding [55]. The idea is to transform the inputs with a random value, do the computations and untransform the result before returning. The attacker is then unable to correlate the time measured with the inputs it gave. Unfortunately, Backes and Köpf [9] show that while blinding reduces the leakage, an attacker with enough observations is able to overcome this countermeasure. Another proposition is to improve blinding by combining it with bucketing [58]. Bucketing consists in imposing several fixed execution times to programs. Time is split into intervals separated by these fixed times and when an exe-

cution terminates in an interval, it is delayed until the time reaches the upper bound of the interval. The previous countermeasure yields meaningful security but an attacker is still able to get information from the timing channels. In theory, *Constant-time programming* is a complete solution against timing attacks. The principle of constant-time programming is that duration of executions should not depend on secrets. More precisely, instructions with specific timing footprint like branching or memory accesses should not depend on secrets. Constant-time gives theoretical complete protection against timing attacks since attackers should not be able to guess any information on secrets using timing information. The downsides of this approach resides in the fact it may be difficult to implement in practice or comes at a large performance penalty [14].

1.2 CompCert

This thesis uses the CompCert compiler in both projects on *Software Fault Isolation* and *Information Flow Preservation*. Therefore, in this section we give an overview of CompCert.

CompCert [66] is a C compiler written by Xavier Leroy which have been machine-checked by the Coq proof assistant [112]. It compiles C programs down to assembly code through a succession of compiler passes which are shown to be semantics preserving. One of the most important point concerning CompCert is that it is a complete, realistic and certified compiler. Complete and realistic refer to its capability to transform high level C language all the way down to assembly for four common architectures: x86, ARM, PowerPC and RiscV. Previously certified transformations were mostly developed for specific optimisations using static analysis [64] or for custom languages [75]. Coupled with satisfying performance for compiled code (similar to gcc -O1), CompCert is an attractive tool to tackle real world problems using formal methods and compilation.

CompCert is divided into multiple passes shown in Figure 1.3. Compiler frontend of CompCert starts from the CompCert C to Cminor which is basically untyped C without side-effects. Afterwards a Control-Flow Graph (CFG) is built in RTL from the program and will be optimised by different compilation passes. Up to here the program is still architecture independent and will now go through multiple transformations which takes into account the target architecture (number of registers, calling conventions, . . .) until reaching assembly code. The compilation chain from assembly to binary uses gcc assembler and linker by default and is not certified.

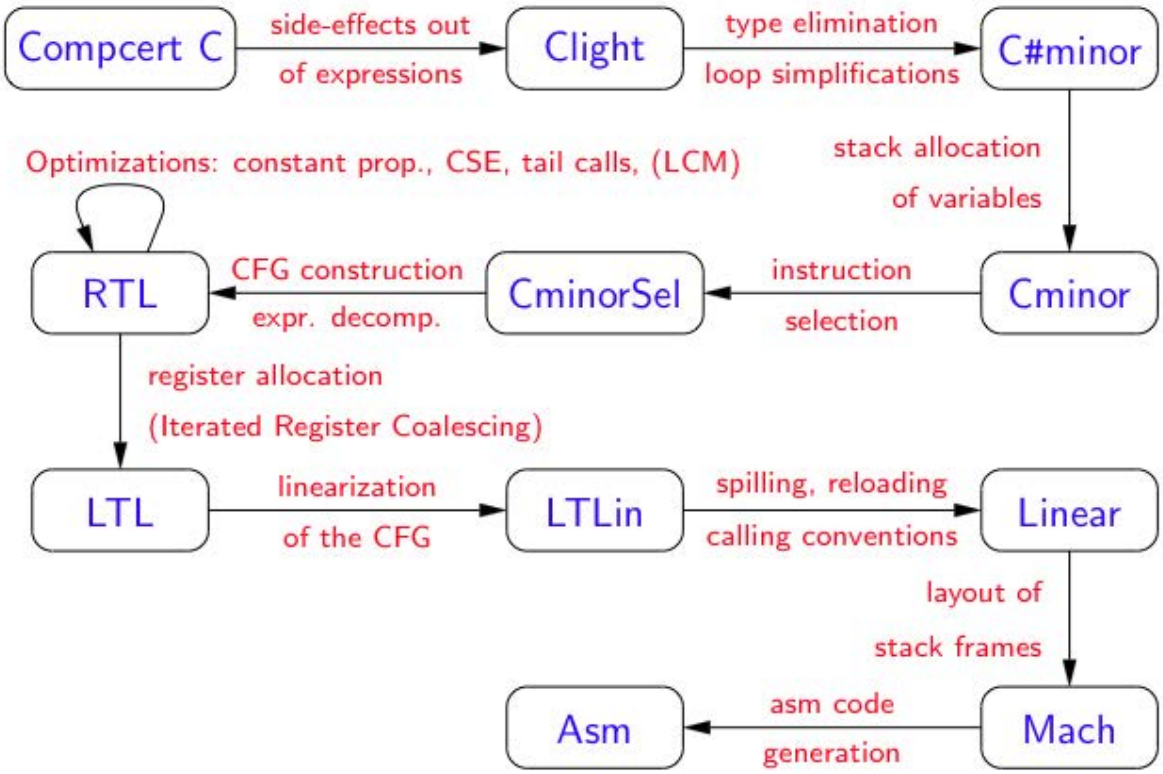


Figure 1.3 – CompCert compilation chain

COMPCERT Soundness Theorem.

Each compiler pass is proved to be semantics preserving using a simulation argument. Theorem 1 states semantics preservation.

Theorem 1 (Semantics Preservation)**.** *If the compilation of program p succeeds and generates a target program p', then for any observable behaviour beh' of program p' there exists an observable behaviour of p, beh, such that beh'* improves *beh.*

In this statement, an observable behaviour is a trace of observable events which can be:

— whether a program *terminates* normally, *diverges* or *goes wrong*. A program diverges if it runs forever. A *goes wrong* behaviour corresponds to a situation where the program semantics gets stuck (i.e., has an undefined behaviour). In this situation, the compiler has the liberty to generate a program with an *improved* behaviour i.e., the semantics of the transformed program may be more defined (i.e., it may not get stuck at all or may get stuck later on).

— calls to external functions

— reads or writes to *volatile* variables. These volatiles can correspond to mapped memory and are hence treated as input/output operations

Two points can be noted from Theorem 1. First, COMPCERT is allowed to fail during compilation in case of errors. Secondly, since C is a non-deterministic language, the compiler is free to pick one behaviour of its liking.

If a source program p respects a specification $Spec$ which corresponds to a set of acceptable traces if for any of its observable behaviour beh we have $beh \in Spec$. We also say that $Spec$ is a safety property, which defines a set of acceptable behaviours. A goal of COMPCERT is to make sure if source program respects safety property $Spec$ then for any observable behaviour beh' of compiled program p' then we also have $beh' \in Spec$. However, Theorem 1 is not sufficient to preserve a trace property because the target program p' may have behaviours that are not accounted for in the program p and could therefore violate the property. Corollary 1 states that in the absence of going-wrong behaviour, the behaviours of the target program are a subset of the behaviours of the source program.

Corollary 1 (Safety preservation)**.** *Let p be a program and p' be a target program. Consider that none of the behaviours of p is a going-wrong behaviour. If the compilation*

of p succeeds and generates a target program p', then any behaviour of program p' is a behaviour of p.

As a consequence, any (safety) property of the behaviours of p is preserved by the target program p'. While vanilla CompCert is suited to preserve safety properties it might be lacking for some security properties. One issue is that CompCert is only able to preserve behaviours that are defined as "observable". Security properties which depends on events that are not observable by CompCert may not be preserved during compilation. A second point, is that CompCert preserves safety properties but no guarantees have been made concerning information-flow properties. Contrary to safety properties, information-Flow properties are defined over a set of executions and includes multiple useful security properties such as *non-interference* [41] or constant-time [14].

CHAPTER 2

SOFTWARE FAULT ISOLATION

Because big software systems are complex to maintain and customize, many popular projects can be extended by third-party modules. They can be operating system kernel's modules such as Linux's .ko modules, web-browser extensions, native libraries, or any kind of plugin. They extend the possibility of their host software. For the Linux kernel, the .ko modules that can be loaded using the `modprobe` command. For a web browser, these are native plugins such as Adobe's Flash player. For interpreters, these are the native libraries of the system that can be loaded and used from the interpreted language. In Java, this is a `native` method.

We fix some terms that we will use continuously in this chapter: "host" refers to a trusted program that uses external untrusted "modules" for computations.

Cooperation between a host program and third-party modules has lots of advantages such as separation of tasks, reuse of efficient code, customization for the users... However, while the host software is usually of good quality, carefully developed and extensively tested, its modules can lack such quality. Such modules may provide backdoors to attackers who wish to corrupt our host program. For instance, Adobe's Flash player is a complex software with a long history of security issues. Because it is run as a standard process without any restriction by the web browser, attackers can use a vulnerability in this extension to remotely control the computer running it. Similarly, a bug in a kernel extension can lead to privilege escalation by corrupting another, unrelated part of the kernel, or by using the extension privileges to corrupt other parts of the system. Furthermore, in some cases the module may be malicious itself and can target the host program directly. Numerous techniques already exist to prevent a corrupted module from spreading to other entities and are called isolation techniques.

With the progress of computers, requirements on speed have been constantly increasing. The main challenge that isolation techniques face nowadays is to ensure security of the host program while limiting the impact on speed of the computations. Techniques including the use of a virtual machine or a monitor [40] provide strong

security guarantees. These techniques give more control to the host, because it can specify a fine-grained security policy and enforce it. Unfortunately, such techniques imply a lot of overhead because of frequent and complex dynamic verifications. Operating system processes is another a solution. They provide weaker isolation than the previous techniques but have faster communication channels between processes. The weaker isolation guarantees come from the fact that, although processes have separate virtual memories, they still share multiple components like actual memory, cache, CPU... However, the cost of context switching between processes is still high and is not acceptable if communication between host program and modules. The isolation technique we present in this chapter is called *Software Fault Isolation* (SFI). SFI is an isolation technique which allows a main program to work with untrusted module safely and with maximum speed for host-module communications. The main program will host the untrusted modules in delimited "sandboxes" directly in its own memory space to allow faster exchanges. Modules are not able to write or jump outside of their sandboxes and communication is constrained to a thoroughly checked interface. Such limitations are enforced by code rewriting of the modules. These modifications take effect at runtime and cause slight overhead for the modules in exchange of maximum speed for host-module communications. Situations where modules only contain a small amount of code but are called frequently are perfect for SFI.

In this chapter on SFI we first present some common isolation techniques and compare their advantages and disadvantages with SFI. Then we will explain SFI in detail and show the current state of SFI, the state-of-the-art techniques and how it is used nowadays. Finally we will introduce CompCertSfi, our implementation of SFI that use the certified compiler CompCert to ensure the security of our approach.

2.1 *Software Fault Isolation (SFI)*

In this section, we present the state of the art related to SFI. First of all, we introduce in details the principles of SFI, which have been greatly explained in the work of Wahbe *et al.* [114]. We will see that in general SFI can be divided into two parts: the generation of secure code and the verification of the executables before loading. Hence, we will begin by presenting the different implementations for the generation of SFI compliant code. They usually aim to obtain both security and speed which are the strong points of SFI. Therefore, optimisations to obtain faster SFI have also been a main point of

research and are detailed Section 2.1.4. Finally, we will also talk about the verification step which may look simple but covers the difficult issue of reliable disassembly of binaries.

2.1.1 Principles of SFI

The goal of isolation is to make sure any bug or vulnerability in a module cannot affect the integrity of the host. In this section, we introduce the main ideas of *Software Fault Isolation*. All recent implementations are derived from these principles. Those concepts are presented in a simple setting here, some issues and implementations specific details are presented later in following sections. The goal of SFI is to allow a host program to execute safely potentially dangerous modules in its own address space. To accomplish that, these modules are isolated in delimited areas of the host memory called *sandboxes*. In most implementations of SFI each module possesses two sandboxes. One of the sandboxes is located in the code segment (where the code is stored) and the second in the data segment. Jump from a module needs to target an address of the sandbox of the code segment. The same applies for any writes with the sandbox of the data segment. We will use this model (1 module, 2 sandboxes) to explain the principles of SFI.

The SFI approach can be divided into two core steps. The first one is the rewriting of the untrusted module to prevent it from accessing any memory out of its sandbox. The second one is the verification of the rewritten code before loading it into memory. This step checks if the rewriting done in the previous part is still present and valid in the code.

2.1.1.1 Foundations of SFI

The main principle behind SFI was first presented in the work of Wahbe *et al.* [114]. The implementation described in the paper was destined for a RISC architecture like MIPS or Alpha.

SFI declares that a module is effectively contained in the sandboxes if the following three security properties are true:

— **Verified code**: only instructions that have been checked by the verifier will be executed

— **Memory safety**: the module will not write or jump out of its sandboxes

— **Secured communication**: every control flow transfer from the module to the host program pass through a secure interface. All the calls to the interface are identified and checked against security rules defined by the host.

The first property makes sure that all the code executed by a module will be checked. This also means that if the module contains self-modifying code the code produced at runtime also needs to be verified to be able to run. In [114] self-modifying code is forbidden but this feature is required in NaCl [117] (an implementation of SFI). *Memory safety* prevents any illegal access to the memory of the host program. In this case we forbid any write and jumps but stricter policies also forbid modules to read data located outside of the code sandbox. The last property allows only licit interactions between the host and the module. Hence, the control flow cannot be compromised avoiding unexpected behaviour from the host program.

2.1.1.2 Toolchain

The SFI toolchain of [114] is presented in Figure 2.1. The SFI generator transforms the assembly code of the untrusted modules so that they respect the security properties presented before. The generator is integrated into the compiler which will create a sandboxed executable. Afterwards this executable is checked by the verifier before being loaded in memory. The verifier checks that the transformations introduced by the generator are present and valid. If the verification fails the module is rejected and is not executed. When evaluating security implementations one important criteria is the size of the *Trusted Computing Base* (TCB). This corresponds to the core code that needs to be correct so that the security properties of the implementations still hold. With SFI, as long as the verifier is correct, it is not possible to execute modules that do not satisfy the SFI policy. This is one advantage of SFI: only the verifier needs to be in the TCB.

2.1.1.3 Code transformation

To protect a program from its modules, the generator will modify every write and jump instruction of the modules so that they can only target addresses of their sandboxes. The generator has to face three issues to do so. The first one is to introduce protection mechanisms before every dangerous instruction in order to confine the modules within the sandbox. Secondly, we have to make sure that these protection mechanisms

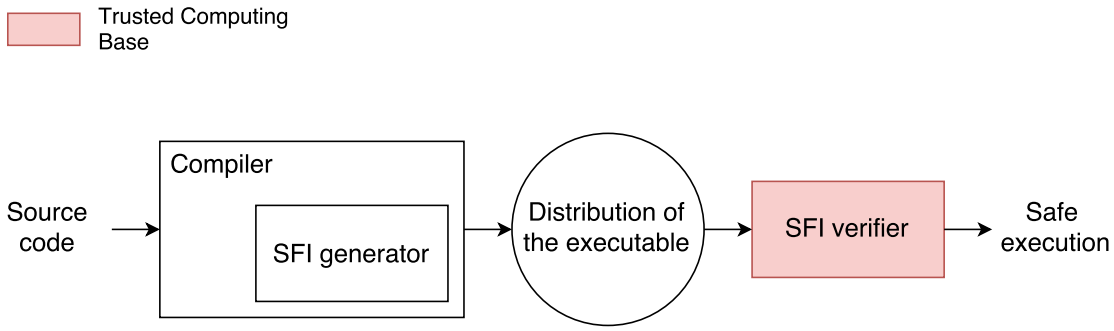


Figure 2.1 – SFI toolchain

cannot be circumvented by the modules. Finally, we only allow the modules to interact with the host program through a secured interface specified by the latter. For example, a browser only allows its modules to use web APIs to interact with its resources. This way, untrusted modules can only act within the boundaries of the web APIs which is secure if correctly designed and implemented.

Confining memory accesses. For jumps or writes using direct addressing mode (the address is hard-coded in the code) simple static analysis can check if the target address is authorized. The issue lies in indirect addressing mode where the target address is computed at runtime. For these cases, using security enforcements that take effect at runtime is more suited to ensure that the module stay within the sandboxes. One choice would be to add dynamic checks in the module code to verify whether the computed address is licit or not. This gives more control when the SFI policy is violated allowing to pinpoint and handle faults easily. The choice taken in SFI is to use a transformation called *sandboxing*. The idea is that regardless of the target address it will be rewritten to be within the boundaries of the concerned sandbox. Obviously if the address was already within the correct sandbox, the sandboxing operation should be idempotent and should only modify illegal addresses. This principle is called *transparency*, SFI transformations do not change the execution of safe programs. Sandboxing has the advantage of having lower overhead than checking addresses, given certain requirements that we detail next.

The sandbox needs to be a contiguous memory area whose size is a power of two. These requirements allow easier and faster sandboxing by allowing the use of bit arithmetic. In these conditions, we only need to verify that the most significant bits of the

targeted address match those of the concerned sandbox. For example, if we allocate the memory area `[0xda000000 - 0xdaffffff]` to a sandbox, then all the addresses whose most significant bits match `0xda` are located in this sandbox. Thus, all the SFI transformations will restrain the memory writes and jumps to the addresses of the sandbox area. In the future examples, we will keep using this sandbox which most significant bits are `0xda`, this sequence of bits is also called a *tag*. Each tag is specific to a unique sandbox and this term will be used repeatedly in our report.

The sandboxing operation is simple: the targeted address has its most significant bits replaced by the tag of concerned sandbox.

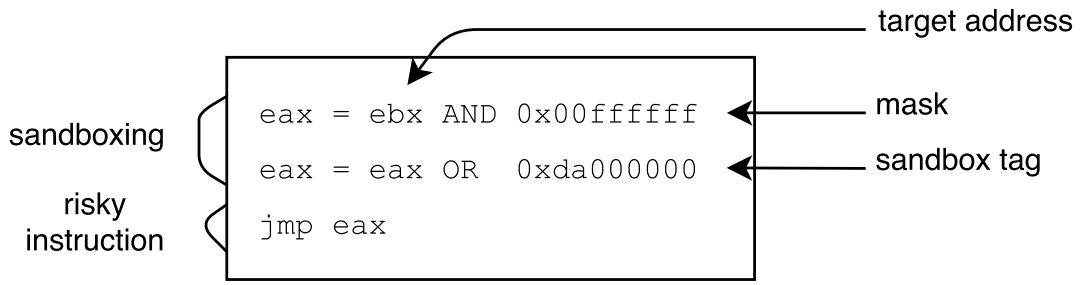


Figure 2.2 – Possible code for the sandboxing operation

Figure 2.2 represents an example of the sandboxing operation. The sandboxing starts with a masking operation which sets the most significant bits of the address stored in the register `ebx` to zero. Afterwards the second instruction writes the tag of the sandbox on the bits it just reinitialised before. Hence, we are sure that the `jmp` instruction will target a location in

the sandbox of the untrusted module. Note that as mentioned previously, the sandboxing does not change the behaviour of the module if the targeted address is already in the sandbox. For the write instructions, the principle is the same. We inject the sandboxing instructions before every write that can endanger our host program.

Protection of the sandboxing mechanism. We made sure in the previous section that a module cannot jump or write to a location out of its sandbox. Now we also want to protect the sandboxing operations to prevent any malicious code to bypass the runtime checks inserted by SFI. Using the example in Figure 2.2, we could imagine code which directly jumps to the `jmp eax` instruction. To protect the sandboxing, the solution is to reserve dedicated registers exclusively used for sandboxing. Masks and tags

are always the same and by storing them in dedicated registers we make sure that they cannot be used with wrong values. Furthermore, the dedicated registers used for jumps and writes will always contain addresses within the sandboxes which prevent any illegal memory accesses. These registers will not be available anymore for the rest of the code. Sandboxing requires three dedicated registers for each sandbox. These dedicated registers were not represented in Figure 2.2 for simplicity but are necessary for the sandboxing operation described. A first register is used to keep the mask value (`0x00ffffff` in Figure 2.2). A second register is reserved to store the tag of the concerned sandbox (`0xda000000` in Figure 2.2). Those two registers are necessary in the original implementation of SFI [114] because the size and value of a sandbox tag is unknown at compile time. However, these two registers can be replaced by constants (like in Figure 2.2) reducing the number of dedicated registers by using code rewriting at load time. The third dedicated register is used to manage the operations contained in the sandboxing, in our example in Figure 2.2 it would be the register `eax`. This way during the whole execution of the module `eax` only stores addresses of its sandbox. Then even if malicious code can jump directly to the instruction `jmp eax` we are sure that it stays in the sandbox. Worst case scenario would be that the value stored in `eax` was wrong and the dangerous module crash or has unexpected behaviour within the sandboxes. As long as the host program is not compromised by the behaviour of its modules the it is considered successful.

These dedicated registers are never used by the rest of the code and their values cannot change except during sandboxing operations. Since we have two sandboxes, one for the data and one for the code we then have a total of six dedicated registers. However, SFI manages to reduce the number of dedicated registers to five by sharing the same mask for both sandboxes. We can question ourselves on the efficiency of SFI when we remove five registers for the execution of the module code. [114] targets modern RISC architecture like MIPS or Alpha where there are generally 32 general-purpose registers. Moreover, the experiments show that removing five registers for the gcc compiler impacts insignificantly the efficiency of the programs tested. For architectures like x86 or ARM where the number of registers is smaller, specific techniques are used to lower the number of dedicated registers and will be discussed in Section 2.1.5.

Controlled interactions with the protected program. It is necessary for SFI to also control the different interactions between the module and the host program. Without

restrictions, malicious modules could, for example, make function calls with wrong parameters which could compromise the state of the host program. To avoid such a situation, SFI needs the host to define an interface which describes all the authorized entry points with the allowed values for parameters. SFI then transforms the module so that every call to the host program passes through connectors called *stubs* at runtime. These stubs make sure that calls from the module to the host program are identified and authorized. Furthermore, the stubs make sure that the parameters supplied are within the range of authorized values defined by the interface. If a call to the host does not respect the interface, the call will be rejected in a way defined by the implementation (for example the module will crash). These stubs are part of the trusted library of SFI which also setup the sandbox at runtime. The trusted library is part of the Trusted Computing Base (TCB) of SFI with the verifier. System calls are controlled the same way. The stub first checks the call, and if it is authorized, it transmits the system call to the host. Then the host executes the system call to the kernel and returns the results to the module.

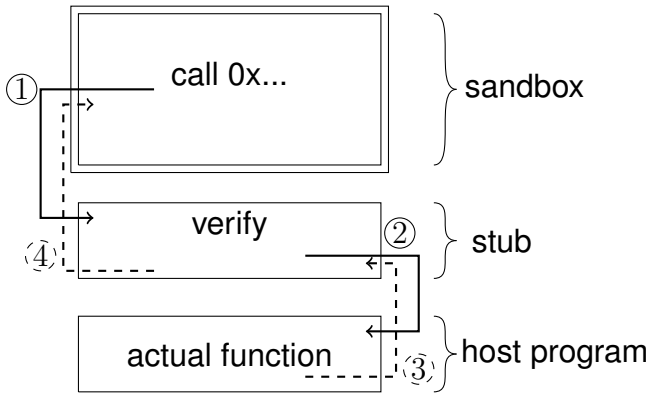


Figure 2.3 – Overview of the work of a stub.

Figure 2.3 summarizes the role of the stub:

— **1**: Direct call to the stub;

— **2**: After verifying the arguments, the stub calls the function;

— **3**, **4**: Return to the sandbox.

All implementations of SFI share this concept albeit with different names.

2.1.1.4 Verification

The verifier is the last element of the SFI chain. Consequently, it is necessary for it to be part of the TCB contrary to the code generator. As long as the verifier is not flawed only modules that respect the security policies of SFI will run in the host memory. Therefore, if the code generator is flawed then the verifier will reject all these potentially dangerous modules. Even if this is not useful for computations this protect our host program against untrusted modules. Nevertheless, the verifier relies on the generator work to do the checking. It is important for the verifier to be sound (all modules accepted by the verifier are secure) but it is difficult for it to be complete (all secure modules are accepted by the verifier). Usually, verifiers are tailored to check modules that have been rewritten by a specific code generator. Indeed, most verifiers know the SFI constructs employed by specific generators and are only able to check the SFI properties for these constructs. While this does not allow flexibility when choosing a verifier and a generator it has the advantages to keep verifiers simple and fast. Since verifiers are part of the TCB, having simple code and logic in verifiers gives us stronger trust in the SFI implementations. Therefore, for modules that have not been transformed by its code generator, a verifier will most of the time reject them even if they are secure.

The verification process can be divided into 5 steps:

1. disassemble the binary of the module
2. writes and jumps use hard-coded addresses of the sandboxes or dedicated registers
3. dedicated registers holding the mask and tags are never modified
4. dedicated registers holding addresses for sandboxed jumps and writes cannot be used without going through sandboxing
5. calls to the host program pass through stubs

Verification is applied to binary modules so the first step is to disassemble them. The implementation of Wahbe *et al.* [114] was made for the architectures MIPS and Alpha. In these architectures, all the instructions are 32 bits long. This particularity makes the disassembly easier since we just need to treat sequences of 32 bits. Steps 2 to 4 ensure that no writes or jumps can access memory outside of the sandboxes. For step 4, every time the dedicated registers used for sandboxed write or jump are modified,

the verifier labels the following lines of code as an *unsafe region*. For better comprehension, we call these registers *operation registers*. It is important for these operation registers to always contain an address of a sandbox when jumping or writing in the memory. Indeed, if an instruction of the module manages to bypass the sandboxing we know that at least it won't affect memory outside of the sandbox. Therefore, these unsafe regions represent lines of the module's code where operation dedicated registers may contain addresses outside of the sandboxes. Unsafe regions start from an instruction which modifies the value of an operation register and stop when encountering a sandboxing operation or the end of the code segment. To validate a sandboxing block, the verifier makes sure that no writes or jumps can be executed in these unsafe regions.

Most techniques presented in this section were directly related to the work of Wahbe et *et al.* [114] who laid the foundations of SFI. Improvements, new issues and implementations for other architectures are presented in the following section.

2.1.2 Adapting and extending SFI to other architectures

The work of [114] presents a version of SFI for MIPS and ALPHA with certain implementation choices like forbidding the use of function pointer. However, when extending SFI to other architectures new issues appear and have to be dealt with. For example, CISC architecture do not have fixed-length instructions allowing jump in the middle of an instruction. In some architectures, like x86, return instructions use a return address stored in the stack to execute. This address also needs a form of sandboxing to prevent any jump outside of the sandboxes. Furthermore, we will show how the principles of SFI can be extended to enable certain features like function pointers or self-modifying code in untrusted modules.

2.1.2.1 Splitting the code into constant bundles

In CISC architectures, instructions vary in size, and the huge number of different possible instructions makes it possible that reading an instruction starting at the middle of another will execute something meaningful for the processor. In RISC since all instructions have the same size you can only specify an instruction number as jump target rather than an address. Therefore, it is not possible to jump in the middle of an instruction as in CISC where hiding and executing an arbitrary jump inside a seemingly

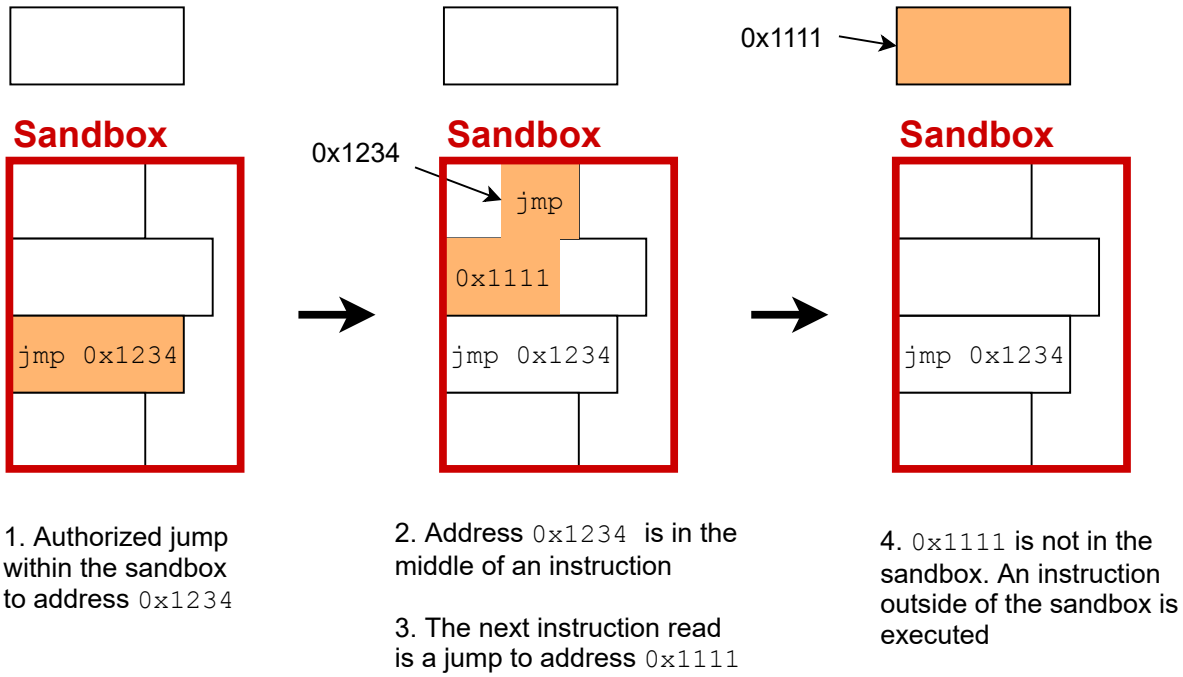


Figure 2.4 – SFI vulnerability in CISC architecture

harmless instruction is possible. Figure 2.4 illustrates how a module can jump outside of its sandbox. First it will execute a jump at address `0x1234` which is in the sandbox and is authorized. However, `0x1234` is an address targeting the middle of an instruction. The CPU when decoding from this address will read the instruction `jmp 0x1111`. This instruction has never been decoded by the verifier and would have been rejected if it had (`0x1111` is not within the sandbox). Hence the CPU will jump outside the sandbox and violate the security policies of SFI.

To avoid this, Pittsfield [73] suggests dividing the code into chunks whose size and location are a power of two. These chunks behave like atomic operations. Hence it is not possible to execute the second instruction of a chunk without executing the first one. Thanks to these properties the sandboxing mechanism can be protected so an attacker cannot avoid the masking present before every dangerous instruction. Therefore, to obtain such properties on these chunks, the following requirements need to be fulfilled:

1. Chunks have a fixed size equal to a power of two;
2. Chunks locations are aligned on their size;
3. Instructions that are targets of jumps are put at the beginning of a chunk;
4. Jump and call instructions are checked so they have their target address always

are multiples of the chunks size;

5. Call instructions are placed at the end of a chunk to have the return at the beginning of the next chunk;
6. A protected instruction and its sandboxing are gathered in the same chunk;
7. It is forbidden to have an instruction overlap on two different chunks;
8. Chunks are padded with no-op instructions.

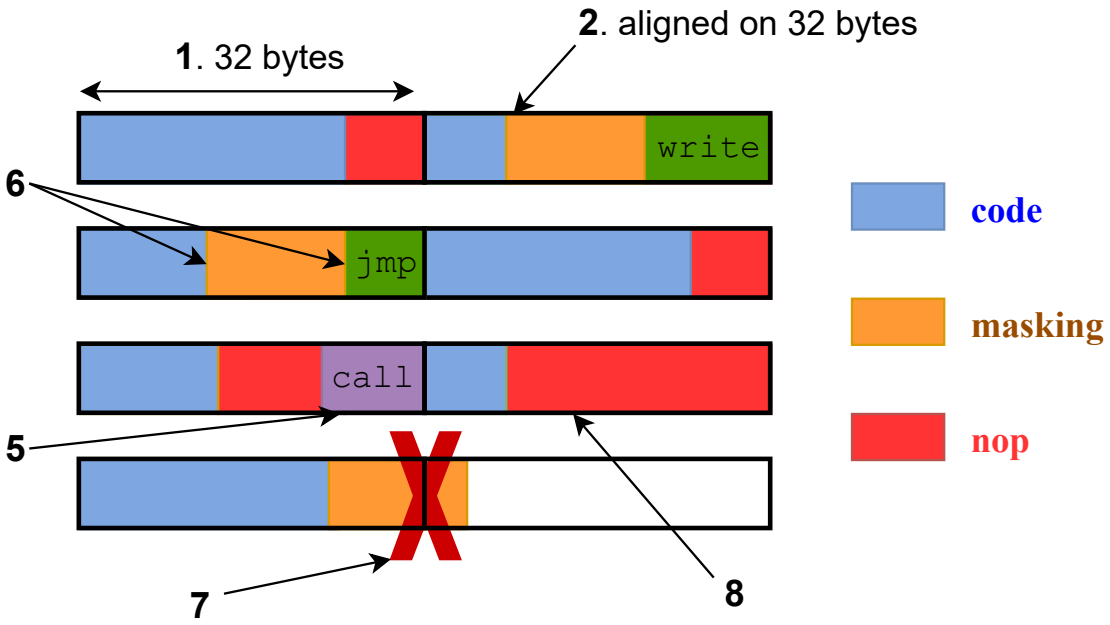


Figure 2.5 – Implementing constant bundles

Implementation of these requirements are illustrated in Figure 2.5 with 32-bytes chunks where the numbers match the requirements listed above. Drawbacks of this approach are the increased size of the code but also the overhead due to added `nop` instructions. Indeed, Pittsfield benchmarks show slowdowns reaching up to 50% in the worst case. Their analysis points that most of the slowdowns encountered was due to the `nop` instructions.

2.1.2.2 Protecting the return address

The return address is an issue appearing on architectures where the `ret` instruction does not need to explicitly specify which address it needs to return to. In x86 `ret` uses

an address stored in the stack for its target address. Since the target address is not stored in a register, it is possible to have race conditions with the classic masking mechanism. For example, if a malicious thread modify the return address in the stack after the masking operation then the CPU might return to an address outside of the allowed sandbox.

2.1.2.3 `pop+jmp`

A common technique to tackle this issue is to replace the `ret` instruction with a `pop+jmp` combination presented in Figure 2.6. The pseudo-code on the left masks the return address directly on the stack. Since stack memory is reachable from other threads it is possible that the value pointed by `esp` has been modified by another thread between the sandboxing operation and the `ret` instruction. Therefore, instead of masking the value stored at the location pointed to by `esp` we use traditional masking on register `eax` and we replace `ret` by `pop+jmp`. However, in modern processor, a shadow stack is often used to obtain better branch predictions of the return addresses which improves the program speed. Without these `ret` instructions, our programs do not benefit from the shadow stack and an average overhead of 25% was measured by [73].

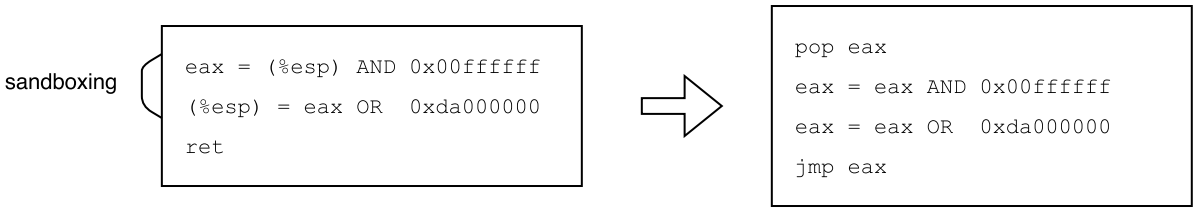


Figure 2.6 – Transforming `ret` into `pop+jmp`

2.1.2.4 Protected scoped stack

[38] separates the stack into two: the scoped and the allocated stack. The scoped stack can only be accessed by a module in a restricted manner. More explicitly all the memory accesses to the scoped stack will be an offset relative to the stack pointer. Stack frames depth and layout are known statistically so the verifier can easily check if the offset to the static pointer is within the frame and if a return address is being accessed. This precise knowledge of the scoped stack allows a memory control similar to virtual registers. Indeed, since all accesses to the stack can be identified statically

it is no different to using register names for reads and writes. Furthermore, special memory regions called guards are added on the extremities of the scoped stack to prevent any overflow. Guards are thoroughly explained later in Section 2.1.5.4. The second stack, allocated stack, includes all the variables that may be accessed through a computed address (usually pointers). These measures ensure that the verifier can detect if any return addresses can be compromised and reject the module if this is the case. A downside of this approach is that using two stacks also require two stack pointers. In particular [38] requests another dedicated register in its implementation.

2.1.2.5 Control stack

A control stack (or shadow stack) is a special stack that solely records return addresses of functions. A routine copies and restores return addresses from and to the normal stack upon function calls and returns. The shadow stack is protected against accesses from the modules, and thus guaranteed to only contain legal addresses. Even if the return address is changed on the stack by the module on the normal stack, its normal value is restored upon return, and thus the return instruction is safe. ARMor [120] and MiSFIT [107] use such technique to protect the return addresses but shadow stacks is a known technique to mitigate buffer overflows. As for the scoped stack, the technique requires an additional dedicated register to follow the evolution of the shadow stack.

2.1.3 Extending SFI

SFI ensures that writes and jumps stay within the sandbox and that calls to the host are constrained by stubs. For indirect control-flow, sandboxing mechanisms are inserted in the code to force jumps and calls to stay within the sandbox. This is enough for SFI to hold but certain implementations [107][118] chose to use stronger information flow restrictions for better security and efficiency. Furthermore, the issue of self-modifying code which was generally avoided is enabled in [6] and presented Section 2.1.3.2.

2.1.3.1 Function pointers

The main issue with function pointers is that the rewriter is unable to know if a call to a function pointer is destined to a function within the module or to a function of the trusted library. Most implementations make a conservative choice and sandbox the destination of the call which prevent any calls to the trusted library through function pointers. Here we present two solutions that can allow secure function pointers in SFI.

Virtual table Misfit [107] suggests a way to deal with function pointers to the trusted library. Since Misfit is targeted at C++ which strongly uses virtual tables to deal with polymorphism they needed a proper solution. They use a hash table containing all the functions to the module. This table is created at link time from the symbols found in the object code. To search if an address is a valid target, the address will be hashed and should correspond to an index of the table. If the index contains no value then the target is invalid, if the index contains a value which is not the target address then it checks the next index until it finds the right value or no value. This implementation ensures near-linear search time depending on the density of the table. Therefore, this mechanism allows stronger control-flow for function calls and authorize the use of function pointers to the trusted library.

Control Flow Integrity (CFI) CFI [1] derives a control-flow graph from a program using static analysis. Using software mechanisms CFI makes sure that this control-flow graph is respected during the execution of a program. This ensures that any calls or jumps of a program can be identified either statically for direct addressing or checked at runtime for indirect addressing. This property is stronger than SFI which only require control-flow to stay within the sandbox or to pass through stubs. [38] and [118] apply CFI techniques to improve SFI. To enforce strict control-flow, the authors place a label at every instruction that may be the target of a jump or a call. Furthermore, for every instruction that use indirect addressing to jump, a routine will be called to retrieve the label located at the target instruction. This label is checked against a list of allowed locations and the code continue to execute if the check is successful. With this technique, the issue of function pointer mentioned in the previous paragraph is also solved. In addition, the strong control-flow guarantees of CFI enable few optimisations of SFI transformations and are explained in Section 2.1.5.3.

2.1.3.2 Self-modifying code

Since the beginning, SFI has forbidden self-modifying code. That is because verifying statically the correctness of self-modifying code is very hard. This is due to the fact that rewriting code at runtime is not atomic and modules can jump on this code during the operation. When the code is being rewritten SFI protections such as sandboxing may not be enforced yet which create vulnerabilities in the module's code. Therefore, we have to make sure that even if the modified code is not complete it should not be able to compromise the host program. Most implementations avoid this issue by simply forbidding self-modifying code. However, some applications such as JIT compilers, need to modify the code at runtime for better efficiency. In NaCl [6], the authors suggest an extension of NaCl that allows code to modify itself. It introduces new library functions available to the code to load, modify and unload dynamically generated code. Unallocated space is filled with `hlt` instructions. This feature makes use of the fixed size bundles memory layout described in Section 2.1.2.1.

— To load a code, the implementation verifies that it respects the sandbox policy. If it does, it loads it at an unallocated space in the sandbox space. The loading phase is presented in Figure 2.7 with four bundles of memory. It needs to load the first byte of each bundle last, so that a thread that would execute during the loading of the bundle will immediately execute a `hlt`. Otherwise, the thread would be able to execute statically unknown instruction because when the writing process is in the middle of an instruction, the semantics of that instruction is unknown.

— To unload a code, the implementation writes back the `hlt` instructions, starting with the first byte of each affected bundle. In this way, any execution in this invalidated region will crash the module. Then, marking the region free can happen only when the implementation knows no thread is currently executing in the affected bundle. For that purpose, it waits for each thread to enter the trusted library, because at that time they are not inside a freed bundle, and they would execute a `hlt` instruction, at the beginning of a freed bundle afterwards. That is especially important for threads sleeping in the middle of a bundle: they should not wake up in the middle of an instruction later affected to the same location.

— To modify a code, the implementation needs to ensure the modified code is laid out exactly as the old one. First, the first byte of each bundle is rewritten to `hlt`. Each instruction is then rewritten in order to use an eight-byte atomic write (be-

cause of a hardware limitation). If an instruction is longer than that, it is first replaced by a `hlt` instruction, rewritten entirely except for the first byte, and then only the first byte is rewritten to the correct value.

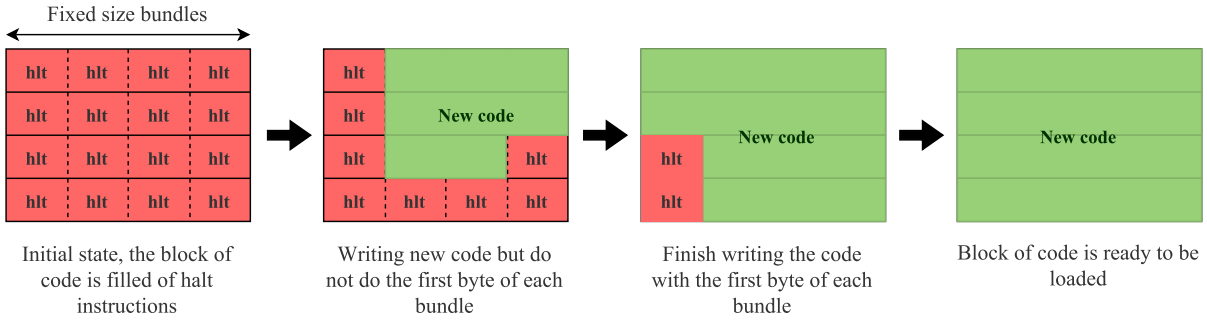


Figure 2.7 – Loading code during runtime with SFI

2.1.4 Optimisations

SFI main goal is to prevent module from compromising the host program with maximum speed. Different techniques have been developed to reduce the overhead caused by the SFI protections. These optimisations may use hardware facilities, better register management, and static analysis to produce faster secure code.

2.1.4.1 Using hardware security for isolation

Some architectures may provide specific hardware protection mechanisms. Some implementations use these protections to implement the security and speed of the transformed binary.

x86-32 has a segment mechanism that can prevent jumping, reading or writing outside designated area. Code, data and stack are all assigned to a memory segment respectively called CS, DS and SS. Therefore, these segments can serve as natural sandbox for SFI removing the need of sandboxing. For instance, NaCl [117] and Pittsfield [73] use those to constrain the module to its sandbox. Since a hardware check is used instead of a software check, this technique implies no overhead. To use it safely, implementations must forbid any modification of the segment registers from untrusted code.

2.1.4.2 The art of choosing the mask

The sandboxing operation usually uses two instructions as shown on the left side of Figure 2.8. A first AND instruction to turn off the bits matching the tag of the SFI memory region, then an OR instruction to set these bits to the sandbox tag. Pittsfield [73] suggests reserving from use, the memory starting from address 0 to the size of a sandbox. This memory region is called *zero-tag region* and triggers a fault when writing or jumping to it. The idea is that instead of clearing and setting the tag we only require clearing a part of the tag so that if the address was incorrect it will point to the zero-tag region. For example, if `[0x00000000 - 0x0fffffff]` is the zero-tag region reserved from use then we take `[0x10000000 - 0x1fffffff]` as the sandboxed data and `[0x20000000 - 0x2fffffff]` as the sandboxed code. Pointers are rewritten either to the sandbox tag, or to the zero tag which was reserved for this situation. Figure 2.8 illustrates this optimisation. Previously, on the left side, sandboxing required an AND and OR operations to work. On the right side, the new version reduced to a single AND operation. Sandboxing is required for the indirect jump and the target address must stay within the sandboxed code `[0x20000000 - 0x2fffffff]`. If the initial target was already in the sandbox then its tag and all the lower bits will be preserved by the AND operation, its value is unchanged. In the other case where the target address was not in the sandbox, its tag will keep the bits corresponding to the sandboxed code. So, either these bits are set to 1 and it will jump to the sandbox else it will jump to the zero-tag region and the module will trigger a fault, preventing any harm to the host.

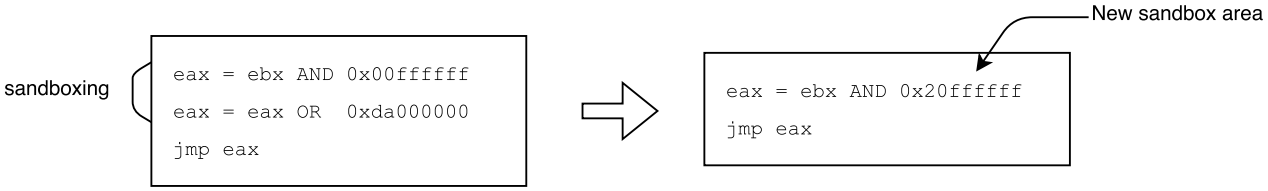


Figure 2.8 – Reducing the sandboxing to a single instruction

This optimisation has also been reused in NaCl [117] and BakerSFIeld [53].

2.1.5 Register management

2.1.5.1 Naive implementation

During the code transformation by the SFI techniques the program is already in the form of assembly code. Eventually, for the sandboxing instructions, SFI techniques need to use some registers, which have already been assigned values by the program. Therefore, before executing the sandboxing mechanisms, the program should save the register values and restore them after the sandboxing. An implementation of this sequence can be seen in Figure 2.9(a). Before starting the sandboxing instructions, the value of the register `eax` is stored on the stack with the instruction `push eax`. At the end of the sandboxing operation the register `eax` is restored with `pop eax`.

```
push eax
eax = ebx AND 0x00ffffff
eax = eax OR  0xda000000
jmp eax
pop eax
```

(a) Naive sandboxing

```
eax = ebx AND 0x00ffffff
eax = eax OR  0xda000000
jmp eax
```

(b) SFI with dedicated registers

Figure 2.9 – Register management for sandboxing

2.1.5.2 Dedicated registers

The classic implementation main issue is easily seen: the sandboxing operation needs at least five instructions including the protected instruction. Hence the cost of sandboxing becomes quite high which could penalize the usage of SFI.
An idea already used in the work of Wahbe *et al.* [114] is to use dedicated registers for the sandboxing operations. In the previous example in Figure 2.9(a), the register `eax` would be specifically used for the masking operations. Hence the instructions used to save and restore register values would not be necessary anymore and our sandboxing would become as in Figure 2.9(b), reduced to three instructions. However, keeping some registers solely for the sandboxing operations also means that the program cannot access these registers for its execution. With a reduced number of available registers, overheads are also more likely to happen due to register pressure. This downside

is especially true on certain architectures with few general-purpose registers like x86-32 which only has eight.

2.1.5.3 Register management with CFI

As explained previously in Section 2.1.3.1, CFI guarantees that the control-flow of the module follow a control-flow graph processed beforehand. Control-flow properties of CFI are stricter than those of SFI which only require that calls and jumps cannot leave the sandbox and that interactions between host and modules need to go through stubs. SFI requires additional protection like dedicated registers and fixed-size bundles to prevent jump outside of the sandbox. Dedicated registers make sure that even if a sandboxing operation is done partially it won't target an address outside the sandbox. Fixed-sized chunks prevent the module from jumping in the middle of an instruction which could lead to a malicious execution. Both of these protections can have a non-negligible impact on performance and can be avoided using CFI. Using CFI, we know that these situations cannot happen anymore and we can avoid the use of dedicated registers and fixed-size bundle. The SFI transformations then use an algorithm similar to *Register Allocation* to implement sandboxing. When some registers are unused or *dead* for the rest of the function, those are used for the sandboxing. Otherwise the code generator will simply spill registers before sandboxing. In other words, the sandboxing works as in Figure 2.9 but without the need of dedicated registers. (a) represents times where there are no dead registers and (b) when there are dead registers.

2.1.5.4 Guard zones

Some registers like `ebp` or `esp` are often used with offsets to deal with local variables or return addresses. Another example is the use of arrays. When accessing a value, an array dereferences its base pointer with some offset. Guard zones can be used to reduce the use of sandboxing in a situation where the target address is of value (sandboxed address + offset). These guard zones are areas of the memory which are unmapped and therefore invalid memory. When jumping or writing to these areas the module will crash hence preventing corruption of the host. Guard zones have fixed-size and are added around the sandbox. If for instance `esp` is checked to be inside the sandbox, it may be the case that `esp + offset` is not. As long as `offset` is inferior to the size of the guard zones the verifier can affirm that in the worst case the module will

trigger a fault and terminate. Since guard zones shall not be triggered during a secure execution they have no impact on the execution time and reduce SFI overhead.

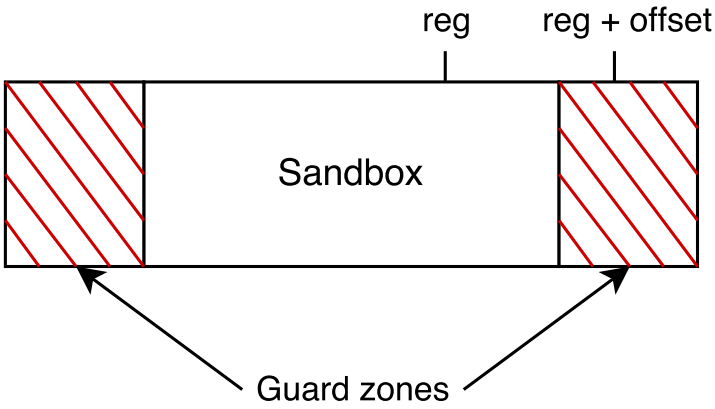


Figure 2.10 – Guard zones

2.1.6 Range analysis

Range analysis is a static analysis which evaluates the range of the possible values of the registers. For example, a freshly modified register will have its range equal to the whole virtual memory, whereas a register which have just been sandboxed will have its range equal to the sandbox memory area. Zeng *et al.* [118] use this information to produce two new optimisations presented in the next section.

2.1.6.1 Redundant checks

The optimisation of [118] takes place after the transformation of the code by SFI techniques. It aims to remove sandboxing operations which are redundant and slow the isolated module down. To detect such occurrences, a range analysis is performed for every register. An example can be seen in Figure 2.11, where the register `eax` starts with an unknown range value. Afterwards `eax` is sandboxed with a value matching an address in the sandbox. Later we see that `eax` without its value having been modified is subject to another sandboxing operation. Hence this second sandboxing is redundant and can be removed from the isolated module for better speed.

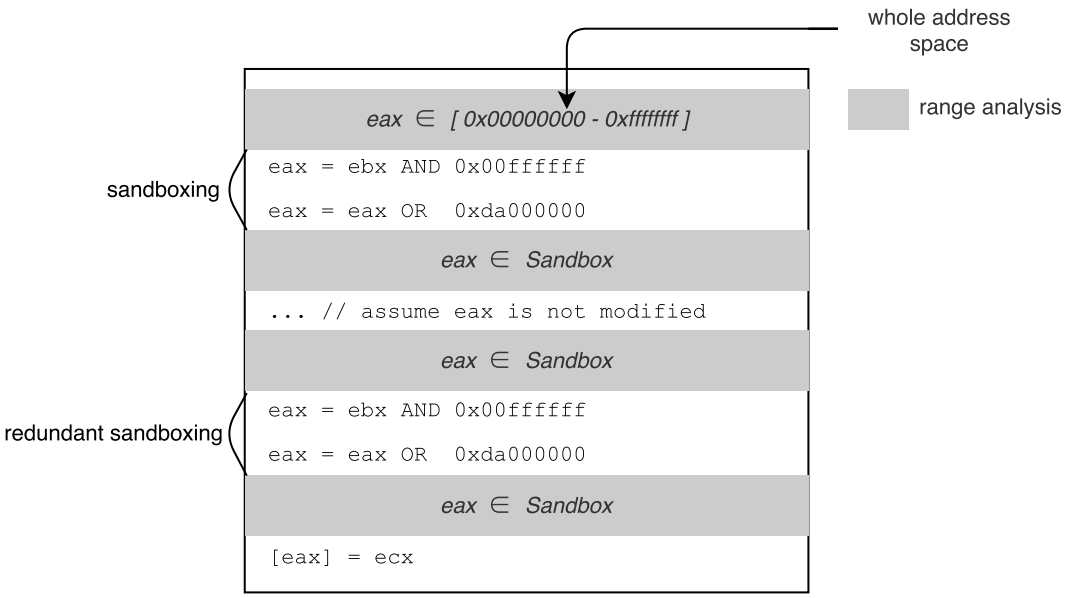


Figure 2.11 – Redundant check

2.1.6.2 Loop check hoisting

Another optimisation of [118] allows the isolated module to reduce the number of sandboxing during loops by hoisting the sandboxing before the loop. This speedup can only be applied in certain loops where the range analysis deems that in the loop, no write or jump instruction can exit the sandbox.

A simple example is presented Figure 2.12 where the fields of an array are being incremented. `array` is a pointer to an array of integers. On every occurrence of the loop, a field of the array is incremented. Therefore, the SFI techniques make sure that the pointers used to access these fields (`array+i`) cannot point to a location outside of the sandbox and do the pseudo-operation `sandbox(array+i)`.

However, thanks to range analysis on this case, one can know that if the pointer `array` is within the sandbox then the pointers `array+i` always point to the area covered by the sandbox plus its guard zones described in Section 2.1.5.4. Thus, the sandboxing in the loop is unnecessary and can be hoisted at the beginning of the loop as shown on the right side of Figure 2.12.

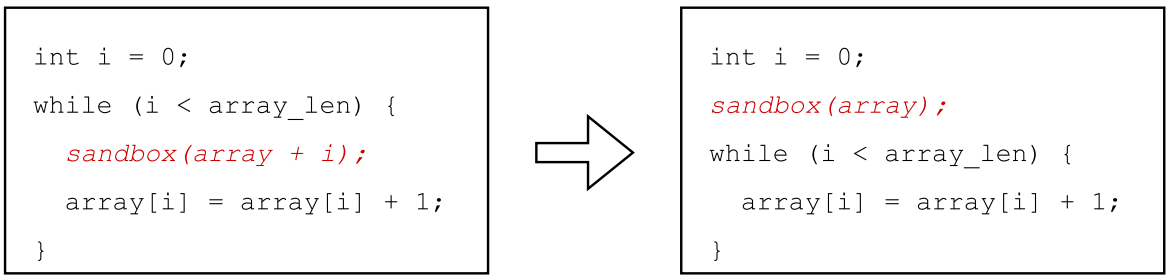


Figure 2.12 – Loop check hoisting

2.1.7 Verification techniques

Verification is usually the last step of SFI. This code is the major part of the Trusted Computing Base in SFI techniques (the interface between the host and untrusted modules are in the TCB too). This means that the amount of trust one can put in the SFI implementation is more or less equal to the trust put in their verification. Since the verification step needs to be sound, all the verifiers seen in the literature have a conservative policy. Indeed, the verifier only accepts some code if it is able to confirm that the code respects SFI properties. Hence every executable accepted by the verifier is safe. However, that also means that a safe program, which does not fulfil the criteria of the verifier will not be accepted either.

For traditional SFI the verification consists in three steps:

1. direct jumps and writes have their target address statically checked
2. indirect jumps and writes need to be sandboxed
3. calls to the host need to pass through stubs

To execute these steps, multiple solutions are available, RockSalt [77], for example, uses regular expressions directly on binary files. But most implementations of verifier start by disassembling the executables then check that the program follows the procedures of SFI. Verification of assembly code is much simpler than the disassembly phase so we could reduce the TCB to the disassembler. Unfortunately, disassembling is non computable problem for generic programs and is still an active area of research.

Therefore, this section will address the different ways found in the literature of SFI to have reliable verification like having compilation constraints for better disassembly or using formal methods for the verification.

2.1.7.1 Linear disassembly

To our knowledge every work on SFI using a disassembler used a simple linear disassembly. Linear disassembly just reads the bytes one by one until it matches an assembly instruction. Then it repeats this sequence until the end of the binary file. However, the problem of disassembling is known to be undecidable for arbitrary input program (similar to the halting problem). Instead of using arbitrary programs multiple constraints on the code generation are often added to get reliable disassembly. The constraints used are the following:

— the code section is not writable, self-modifying code is not allowed

— the code section is statically linked

— all *valid* instructions are reachable by linear disassembly

— aligned bundles of code presented Section 2.1.2.1 help disassembly, because no instruction crosses a chunk boundary and control flow only targets the beginning of a chunk

These constraints are used to make the disassembly completely reliable. Afterwards the verifier only needs to perform the three verification steps mentioned earlier directly on the assembly code.

2.1.7.2 Improving verification with static analysis

While having a simple verifier is good for the TCB it also limits the possible SFI transformations that can be checked. Usually a verifier is specific to a code generator and is only able to check a module if the SFI protections are untouched. The first issue is portability, a verifier cannot validate safe modules that have been transformed by an unknown code generator. Secondly it severely limits the optimisations that can be applied to the SFI protections since they cannot be checked afterwards. [118] uses range analysis to provide two optimisations to their implementation of SFI (Section 2.1.6). To be able to verify such optimisations the same range analysis is also used by the verifier. Work by Besson *et al.* [21] aims to create a generic verifier which can verify if a module respects the security principles of SFI. Their work defines a defensive semantic for SFI and use this semantic to check if a module is secure or not using abstract interpretation.

2.1.7.3 Certified verification

Usually the TCB of SFI implementations is mostly equal to the code of the verifier. However, in ARMor [120] and RockSalt [77] both use formal methods in order to reduce the amount of code which needs to be trusted.
In ARMor the verifier automatically extracts proofs and facts from transformed programs and uses them to verify a top-level safety proof. This means that the user can be mathematically certain the program they run is correctly sandboxed.
RockSalt uses the Coq proof assistant to build a provably correct alternative verifier for NaCl sandboxes [117]. Their method is to generate regular expressions that match any program that respects the sandbox policy in such a way that it is easy to show it is correct. Finally, they implement the verifier in C for speed. The two verifiers are proven to be correct which means that if they validate a module then it cannot break the policy of SFI when loaded and executed. Therefore, the TCB of these implementations can be summarized to the amount of trust one put in their respective proof assistant: HOL and Coq.

2.1.7.4 Removing the verifier

Portable Software Fault Isolation (PSFI) [60] is a particular version of SFI which does all the code transformation in a high-level language. Since the transformations are done on a language which is architecture independent, a single code generator can produce secure code for all the architectures supported by the compiler. Another advantage is that better speed can also be achieved since SFI protections benefit from compiler optimisations. However, in return it becomes more complicated to have the SFI security proofs on the binary code that is executed, due to various reasons coming from the compilation phases. First of all, compilers usually do not give guarantees on the code produced and one cannot be certain that the output binary will keep all the sandboxing mechanism inserted at high-level. Secondly, it might be more difficult for a verifier to check if the security property of SFI still holds for the binary. Indeed, the compiler may have modified instructions during code optimisation, so, even if the binary is secure, it is harder to be sure of it.

To solve these constraints, PSFI chooses to use a certified compiler called CompCert [67]. CompCert was written using the Coq proof assistant and was proven to keep the semantic of the compiled C programs. In this work CompCert was modified

to inject the sandboxing instructions during the compilation. Furthermore, with the Coq proof assistant, the security properties of SFI usually given by the verifier, are now guaranteed by the compiler. PSFI compiler has been proven to meet these two criteria:

1. Any input program is compiled into a program which executes safely (in the sense of SFI)
2. If the input program is SFI-safe then the compiler does not alter its behavior

This choice enables one to get rid of the verifier in the compilation chain. Indeed, since all the security guarantees are given by the compiler, a verifier is not needed anymore and follows the procedure of Figure 2.13. Starting with a source program, one only needs to compile it to obtain a SFI-safe executable. We can notice in Figure 2.13 that the TCB is now reduced to the proofs behind CompCert. This means that the trust we can put in this implementation of SFI is equivalent to the trust we have in the Coq proof assistant.

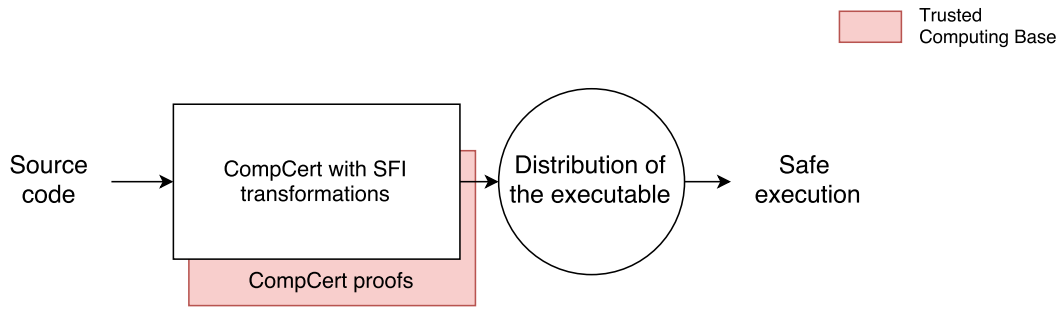


Figure 2.13 – PSFI chain

A drawback of this approach is that to be sure that the binary we execute is SFI-safe, we need a certainty that our binary has been compiled with this compiler. This condition implies that it is necessary to have either the source code of the external modules we want to run or a proof that this compiler was used to produce the binary. PSFI is explained in further details in Section 2.2.1.2.

2.1.8 Conclusion

The main purpose of SFI is to protect a host program to collaborate with untrusted modules loaded in its own memory space. This enables maximum speed for host-modules interactions with slower modules code due to SFI restrictions. Multiple tech-

niques have been developed to reduce the modules overhead caused by SFI. Hardware segmentations, guard zones, static analysis, register management have been applied to fasten modules code.

Comparing work on SFI is not evident since contexts from an implementation to another may vary greatly. Table 2.1 categorize the different works that we analysed under specific categories.

		NaCl	MiSFIT	Pittsfield	ARMor	XFI	PSFI	FBGI	BakerSFIeld	CFI_SFI
References		[117] [98] [6]	[107]	[73]	[120]	[38]	[60]	[24]	[53]	[118]
architecture	ARM	X			X		X			
	x86-32	X	X	X		X	X	X	X	X
	x86-64	X							X	
transformation	IR						X			
	assembly	X	X	X		X		X	X	X
	binary				X					
application	kernel					X		X		
	web browser	X								
	language extension	X [102] [109]								
	general purpose		X	X	X		X		X	X
Formal methods		X [77]			X		X			

Table 2.1 – Summary of techniques

2.2 COMPCERTSFI

In this section, we present our work which draws upon PSFI [60] presented previously. Similarly, our toolchain does not include a verifier but use the correctness property of COMPCERT to preserve SFI guarantees down to the binary code. Our work provides a fully functional and verified SFI toolchain called COMPCERTSFI. The SFI transformation is implemented in a high-level language called Cminor. The transformation has been proven in Coq to ensure a security property that satisfies SFI requirements. Additionally, the transformation is also proven to satisfy a safety property which when combined with COMPCERT correctness property provides strong security guarantees at the binary level. Furthermore, a trusted library and runtime for SFI have been developed to obtain a complete toolchain for SFI. Our work shows that an implementation of SFI without verifier can both provides security and performance. In Section 3.3.5, we provide experimental evidence that COMPCERTSFI is competitive and sometimes outperforms SFI in terms of efficiency of the binary code.

2.2.0.1 Software Fault Isolation through Compilation

A downside of the traditional SFI approach is that it hinders most compiler optimisations because the optimised code no longer respects the simple properties that the SFI verifier is capable of checking. For example, the SFI verifier expects that every memory access is immediately preceded by a specific syntactic code pattern that implements the sandboxing operation. A semantically equivalent but syntactically different code sequence would be rejected. Section 2.1.7.4 presents succinctly PSFI, another methodology where there is no verifier to trust. Instead isolation is obtained by compilation with a machine-checked compiler, such as COMPCERT [67]. Portability comes from the fact that PSFI can reuse existing compiler back-ends and therefore target all the architectures supported by the compiler without additional effort.

PSFI is applicable in scenarios where the source code is available or the binary code is provided by a trusted third-party that controls the build process. For example, the original motivation for Proof Carrying Code [79] was to provide safe kernel extensions [80] as binary code to replace scripts written in an interpreted language. This falls within the scope of PSFI. Another PSFI scenario is when the binary code is produced in a controlled environment and/or by a trusted party. In this case, the primary goal is not to protect against an attacker trying to insert malicious code but to prevent

honest parties from exposing a host platform to exploitable bugs. This is the case *e.g.* in the avionics industry, where software from different third-parties is integrated on the same host that needs to ensure strong isolation properties between tasks whose levels of criticality differ. In those cases, PSFI can deliver both security and a performance advantage.

2.2.0.2 Challenges in Formally Verified SFI

PSFI inserts the masking operations during compilation and does away with the *a posteriori* SFI verifier. The challenge is then to ensure that the security, enforced at an intermediate representation of the code, still holds for the running code. Indeed, compiler optimisations often break such security [104]. The insight of Kroll *et al.* is that if the intermediate code is safe (i.e., that its behaviour is well-defined) then the safety theorem of CompCert (Corollary 1) can be exploited to preserve PSFI security down to the compiled code: guaranteeing that it makes no memory accesses outside its sandbox. We explain this in more detail later in Section 2.2.1.2.

One challenge we face with this approach is that it is far from evident that the sandboxing operations and hence the transformed program have well-defined behaviour. An unsafe language such as C admits undefined behaviours (e.g. bitwise operations on pointers), which means that it is possible for the observational behaviour of a program to differ depending on the level of optimisation. This is not a compiler bug: compilers only guarantee semantics preservation *if* the code to compile has a well-defined semantics [115]. Therefore, our SFI transformation must turn any program into a program with a well-defined semantics.

The seminal paper of Kroll *et al.* emphasises that the absence of undefined behaviour is a prerequisite but they do not provide a transformation that enforces this property. More precisely, their transformation may produce a program with undefined behaviours (*e.g.* because the input program had undefined behaviours). This fact was one of the motivation for the present work, and explains the need for a new PSFI technique. One difficulty is to remove undefined behaviours due to restrictions on pointer arithmetic. For example, bitwise operators on pointers have undefined C semantics, but traditional masking operations of SFI rely heavily on these operators. Another difficulty is to deal with indirect function calls and ensure that, as prescribed by the C standard, they are resolved to valid function pointers. To tackle these problems, we propose an original sandboxing transformation which unlike previous proposals is compliant with

the C standard [50] and therefore has well-defined behaviour.

2.2.0.3 Contributions

We have developed and proved correct COMPCERTSFI, the first full-fledged, fully verified implementation of SFI inside a C compiler. The SFI transformation is performed early in the compilation chain, thereby permitting the generated code to benefit from existing optimisations that are performed by the back-end. The technical contributions behind COMPCERTSFI can be summarised as follows.

— An original design and implementation of the SFI transformation based on well-defined pointer arithmetic and which supports function pointers. This novel design of the SFI transformation is necessary for the safety proof.

— A machine-checked proof of the **security** and **safety** of the SFI transformation. Our formal development is available online [110].

— A small, lightweight runtime system for managing the sandbox, built using a standard program loader and configured by compiler-generated information.

— Experimental evidence demonstrating that the portable SFI approach is competitive and sometimes even outperforms traditional SFI, in particular state-of-the-art implementations of (P)Native Client.

The rest of the chapter is organised as follows. In Section 2.2.1, we present information about the COMPCERT compiler (Section 2.2.1.1) and the PSFI approach (Section 2.2.1.2). Section 2.2.2 provides an overview of the layout of the sandbox and the masking operations implementing our SFI. In Section 2.2.3 we explain how to overcome the problem with undefined pointer arithmetic and define masking operations with a well-defined C semantics. Section 2.2.4 describes how control-flow integrity in the presence of function pointers can be achieved by a slightly more flexible SFI policy which allows reads in well-defined areas outside the sandbox. Section 2.2.5 specifies the SFI policy in more detail, and describes the formal Coq proofs of safety and security. Section 2.2.6 presents the design of our runtime library and how it exploits compiler support. Experimental results are detailed in Section 3.3.5 and Section 2.2.9 concludes.

$$
\begin{array}{rcl}
\mathit{constant} \ni c & ::= & \mathit{i32} \mid \mathit{i64} \mid \mathit{f32} \mid \mathit{f64} \mid \&\mathit{gl} \mid \&\mathit{stk} \\
\mathit{chunk} \ni \kappa & ::= & \mathit{is}_8 \mid \mathit{iu}_8 \mid \mathit{is}_{16} \mid \mathit{iu}_{16} \mid \mathit{i}_{32} \mid \mathit{i}_{64} \mid \mathit{f}_{32} \mid \mathit{f}_{64} \\
\mathit{expr} \ni e & ::= & x \mid c \mid \triangleright\, e \mid e_1 \,\square\, e_2 \mid [e]_\kappa \\
\mathit{stmt} \ni s & ::= & \mathbf{skip} \mid x := e \mid [e_1]_\kappa := e_2 \mid \mathbf{return}\ e \mid x := e(e_1 \ldots, e_n)_\sigma \\
& \mid & \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid s_1; s_2 \mid \mathbf{loop}\ s \mid \{s\} \mid \mathbf{exit}\ n \mid \mathbf{goto}\ \mathit{lb}
\end{array}
$$

Figure 2.14 – CMINOR syntax

2.2.1 Background

This section presents additional information about the COMPCERT compiler [67] for COMPCERTSFI and the Portable Software Fault Isolation proposed by Kroll *et al.* [60].

2.2.1.1 COMPCERT

COMPCERT was discussed previously in Section 1.2 but relevant information for COMPCERTSFI are added here. In our work, SFI transformations are performed on the intermediate representation expressed in the CMINOR language.

The CMINOR language is a minimal imperative language with explicit stack allocation of certain local variables [65]. Its syntax is given in Figure 2.14. Constants range over 32-bit and 64-bit integers but also IEEE floating-point numbers. It is possible to get the address of a global variable gl or the address of the stack allocated local variables (i.e., stk denotes the address of the current stack frame). In COMPCERT parlance, a memory chunk κ specifies how many bytes need to be read (resp. written) from (resp. to) memory and whether the result should be interpreted as a signed or unsigned quantity. For instance, the memory chunk is_{16} denotes a 16-bit signed integer and f_{64} denotes a 64-bit floating-point number. In CMINOR, memory accesses, written $[e]_\kappa$, are annotated with the relevant memory chunk κ. Expressions are built from pseudo-registers, constants, unary ($\triangleright$) and binary ($\square$) operators. COMPCERT features the relevant unary and binary operators needed to encode the semantics of C. Expressions are side-effect free but may contain memory reads.

Instructions are fairly standard. Similarly to a memory read, a memory store $[e_1]_\kappa = e_2$ is annotated by a memory chunk κ. In CMINOR, a function call such as $e(e_1 \ldots, e_n)_\sigma$ represents an indirect function call through a function pointer denoted by the expression e, σ is the signature of the function and $e_1 \ldots, e_n$ are the arguments. A direct call

is a special case where the expression e is a constant (function) pointer. CMINOR is a structured language and features a conditional, a block construct $\{s\}$ and an infinite loop $\mathbf{loop}\ s$. Exiting the nth enclosing loop or block can be done using an $\mathbf{exit}\ n$ instruction. CMINOR is structured but $\mathbf{goto}$s towards a symbolic label lb are also possible. Returning from a function is done by a return instruction. CMINOR is equipped with a small-step operational semantics. The intra-procedural and inter-procedural control flows are modelled using an explicit continuation which therefore contains a call stack.

2.2.1.1.a Going-wrong behaviours in COMPCERT.

As safety is an essential property of our PSFI transformation, we give below a detailed account of the going-wrong behaviours of the COMPCERT languages with a focus on CMINOR.

2.2.1.1.1 Undefined evaluation of expressions. COMPCERT's runtime values are dynamically typed and defined below:

$$values \ni v ::= \mathbf{undef} \mid \mathbf{int}(i_{32}) \mid \mathbf{long}(i_{64}) \mid \mathbf{single}(f_{32}) \mid \mathbf{float}(f_{64}) \mid \mathbf{ptr}(b, o)$$

Values are built from numeric values (32-bit and 64-bit integers and floating point numbers), the $\mathbf{undef}$ value representing an indeterminate value, and pointer values made of a pair (b, o) where b is a memory block identifier and o is an offset which, depending on the architecture, is either a 32-bit or a 64-bit integer.

For CMINOR, like all languages of COMPCERT, the unary ($\triangleright$) and binary ($\square$) operators are not total. They may directly produce going-wrong behaviours *e.g.* in case of division by $\mathbf{int}(0)$. They may also return $\mathbf{undef}$ if i) the arguments are not in the right range *e.g.* the left-shift $\mathbf{int}(i) << \mathbf{int}(32)$; or ii) the arguments are not well-typed *e.g.* $\mathbf{int}(i) +_{int} \mathbf{float}(f)$. Pointer arithmetic is strictly conforming to the C standard [50] and any pointer operation that is implementation-defined according to the standard returns $\mathbf{undef}$.

The precise semantics of pointer operations is given in Figure 2.15. For simplicity, we provide the semantics for a 64-bit architecture. Pointer operations are often only defined provided that the pointers are valid, written V, or weakly valid, written W. This validity condition requires that the offset o of a pointer $\mathbf{ptr}(b, o)$ is strictly within the bounds of the block b. The weakly valid condition refers to a pointer whose offset is

$$
\begin{array}{lcl}
\mathbf{ptr}(b,o) \pm \mathbf{long}(l) & = & \mathbf{ptr}(b, o \pm l) \\
\mathbf{ptr}(b,o) - \mathbf{ptr}(b,o') & = & \mathbf{long}(o - o') \\
\mathbf{ptr}(b,o)\,!\!=\!\mathbf{long}(0) & = & \mathbf{tt} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) == \mathbf{long}(0) & = & \mathbf{ff} \quad \text{if } W(b,o) \\
\mathbf{ptr}(b,o) \star \mathbf{ptr}(b,o') & = & o \star o' \quad \text{if } W(b,o) \wedge W(b,o') \\
\mathbf{ptr}(b,o) == \mathbf{ptr}(b',o') & = & \mathbf{ff} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\mathbf{ptr}(b,o)\,!\!=\!\mathbf{ptr}(b',o') & = & \mathbf{tt} \quad \text{if } b \neq b' \wedge V(b,o) \wedge V(b',o') \\
\multicolumn{3}{c}{\text{where } \star \in \{<, \leq, ==, \geq, >, \,!\!=\}}
\end{array}
$$

Figure 2.15 – Pointer arithmetic in CompCert

either valid or one-past-the-end of the block b. Any pointer arithmetic operation that is not listed in Figure 2.15 returns $\mathbf{undef}$. This is in particular the case for bitwise operations which are typically used for the masking operation needed to implement SFI.

The indeterminate value $\mathbf{undef}$ is not *per se* a going-wrong behaviour. Yet, branching over a test evaluating to $\mathbf{undef}$, performing a memory access over an $\mathbf{undef}$ address and returning $\mathbf{undef}$ from the `main` function are going-wrong behaviours.

2.2.1.1.2 Memory accesses are ruled by a unified memory model [69] that is used throughout the whole compiler. The memory is made of a collection of separated blocks. For a given block, each offset o below the block size is given a permission $p \in \{\mathbf{r}, \mathbf{w}, \ldots\}$ and contains a memory value

$$mval \ni mv ::= \mathbf{undef} \mid \mathbf{byte}(b) \mid [\mathbf{ptr}(b,o)]_n$$

where b is a concrete byte value and $[\mathbf{ptr}(b,o)]_n$ represents the n$^{\text{th}}$ byte of the pointer $\mathbf{ptr}(b,o)$ for $n \in \{1 \ldots 8\}$. A memory write $storev(\kappa, m, a, v)$ is only defined if the address a is a pointer $\mathbf{ptr}(b,o)$ to an existing block b such that the memory locations $(b,o), \ldots, (b, o+ \mid \kappa \mid -1)$ have the permission $\mathbf{w}$ and the offset o satisfies the alignment constraint of κ. A memory read $loadv(\kappa, m, a)$ is only defined under similar conditions with the additional restriction that not reading all the consecutive fragments of a pointer returns $\mathbf{undef}$.

2.2.1.1.3 Control-flow transfers may go-wrong if the target of the control-flow transfer is not well-defined. Hence, a $\mathbf{goto}\ lb$ instruction goes wrong if, in the current function, there is no statement labelled by lb; and an $\mathbf{exit}\ n$ instruction goes wrong if there are less than n enclosing blocks around the statement containing the exit instruction. A conditional $\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$ goes wrong if the expression e does not evaluate to $\mathbf{int}(i)$ for some i. Also, the execution goes wrong if the last statement of a function is not a $\mathbf{return}$ instruction. Last but not least, a function call $x := e(e_1 \ldots, e_n)_\sigma$ goes wrong if the expression e does not evaluate to a pointer $\mathbf{ptr}(b, 0)$ where b is a function pointer with signature σ.

We show in Section 2.2.3 how our transformation ensures that pointer arithmetic and memory accesses are always well-defined. Section 2.2.4 shows how we make sure indirect calls are always correctly resolved. Section 2.2.5 shows that, together with other statically checkable verifications, our PSFI transformation rules out all possible going-wrong behaviours.

2.2.1.2 Portable Software Fault Isolation

Kroll, Stewart and Appel have pioneered the concept of Portable Software Fault Isolation (PSFI) [60] whereby SFI is enforced by a pass of the compiler front-end that is architecture independent. The main expected advantage is that isolation is implemented, once and for all, for any target architecture. Moreover, the generated code is optimised by the back-end passes of the compiler. Compared to traditional SFI, there is no architecture-specific binary verifier but instead the compiler enters the TCB. The key insight of Kroll *et al.* is to leverage a formally verified compiler, namely CompCert, to transfer a security proof of isolation obtained at the Cminor level through the compiler back-end, with minimal proof effort. In the following, we recall the only basic properties that a Cminor SFI transformation needs to satisfy so that isolation holds at assembly level.

In CompCert's terms, the sandbox is identified by a dedicated memory block sb. A Cminor program is secure (Property 1) under the condition that all its memory accesses are performed within the sandbox.

Property 1 (Program security)**.** *A* Cminor *program p is secure if all its memory accesses are within the sandbox block sb.*

After compilation, the assembly code is secure if its observable behaviours are

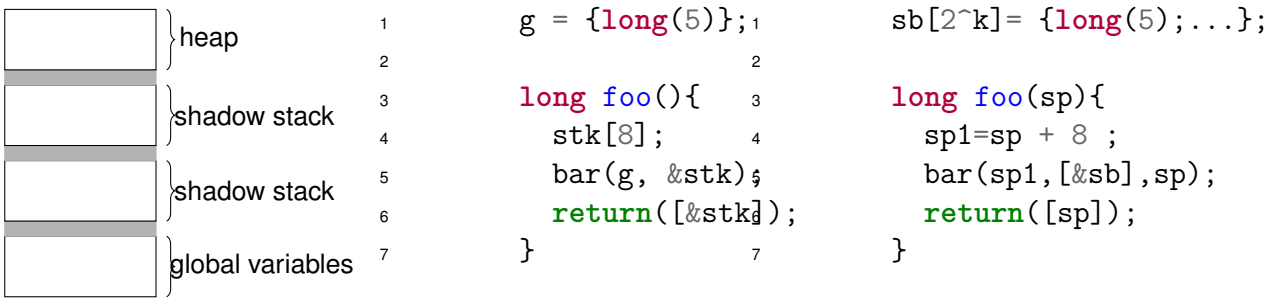
the same as the observable behaviours of the CMINOR program. In order to apply COMPCERT's semantics preservation theorem (more precisely Corollary 1), it remains to ensure that the CMINOR program has a well-defined semantics (Property 2).

Property 2 (Program safety)**.** *A* CMINOR *program* p *is safe if all its behaviours are well-defined, i.e., not wrong.*

Kroll *et al.* state Property 1 by means of an instrumented CMINOR semantics which gets stuck in case of memory accesses outside the sandbox. They prove formally that the additional semantic safeguards are never triggered for a transformed program.

They also sketch some necessary steps to prove the Property 2 of safety but do not propose a formal proof. This leaves open a number of challenging issues such as whether it is feasible to define a masking operation that has a defined CMINOR semantics and how to deal with indirect function calls through function pointers. More generally, the work leaves open whether a formal proof of Property 2 on safety is possible given the restrictions of COMPCERT's semantics (notably pointer arithmetic) and without relying on axioms asserting properties of an external masking primitive. One of the central contributions of this work is to provide a positive answer to this question and propose solutions to these issues where neither the sandboxing of memory accesses nor the sandboxing of function pointers is part of a TCB. The transformation that circumvents the limitations imposed by pointer arithmetic is original and, we surmise, is a necessary component to transfer security down to assembly. For a precise comparison with Kroll *et al.* see Section 2.2.8.

2.2.2 A Thread-aware Sandbox



(a) Layout of memory (b) Original CMINOR (c) Sandboxed CMINOR

Figure 2.16 – Sandbox transformation

The memory address space of a C program is partitioned into a runtime stack of frames, a heap and a dedicated space for global variables. The address space of a sandboxed program is re-organised to fit into a single global variable, sb, where the global variables, the heap and the stack frames are relocated. Figure 2.16a depicts the memory layout of the program after our SFI transformation. Each global variable is relocated and allocated in the sandbox at a given offset, and each global memory access of the program is translated into a memory access in the sandbox. For managing the heap, it suffices to use a sandbox-aware `malloc` implementation that allocates memory inside the sandbox.

To prevent buffer overflows, a standard approach consists in introducing a so-called *shadow stack* that is used to store the function stack frames. Our implementation supports multi-threaded applications and therefore there are as many shadow stacks as there are threads. Upon thread creation, we allocate a novel shadow stack in the sandbox. The shadow-stack pointer is passed as an additional argument to each function call. This is efficient when arguments are passed by register, with the only drawback of reserving an additional register. Frames are allocated by incrementing the shadow-stack pointer at function entry. All accesses to the original stack are then translated into accesses to the sandbox shadow stack. The following Example 1 and the code snippet in Figure 2.16 illustrate the essence of the transformation.

Example 1. *The* CMINOR *program of Figure 2.16b declares a global variable* g *initialised to the 64-bit integer* 5*. The function* foo *allocates a stack frame of 8 bytes that will be used to store a 64-bit local variable. By convention, the current stack frame is called* stk*. The function* foo *calls the function* bar *with as arguments the value of* g *and the address of the local variable* stk*; and returns the value, presumably updated by* bar*, of the local variable.*

Syntactically, the program of Figure 2.16c only performs memory accesses on the global sandbox sb *variable. The size of* sb *variable is* 2^k *for some predefined* k*. At thread creation, a shadow stack is allocated by our sandbox-aware* $malloc$ *in the sandbox after the statically allocated global variables. For our program, the unique global variable* g *is stored at offset 0 and spans over 8 bytes. Therefore, the initial value of the shadow-stack pointer* sp *is 8. After the transformation, the function* foo *reserves the space for the local variable* stk *by incrementing the pseudo-register* sp*. The function* bar *is called with the incremented shadow-stack pointer* $sp1$*, the value stored at offset 0 in the sandbox (i.e., the value of the global variable* g*) and the address of the local*

variable `stk` *which is given by the value of the stack pointer* `sp`*. At function exit, the value of the local variable* `stk` *is returned by dereferencing the shadow-stack pointer* `sp`*.*

Our SFI transformation enforces the isolation security policy stipulating that all memory accesses are performed within the sandbox sb—at the CMINOR level. However, this holds because the semantics gets stuck (i.e., the semantics *goes wrong*) whenever the program performs an access outside the bounds of the sandbox. As explained earlier, the compiler is free to translate this into an insecure program that would escape the sandbox at runtime. To get a formal security guarantee, it is necessary to transform further the CMINOR program to rule out any behaviour that *goes wrong* i.e., ensure Property 2. Given the numerous undefined behaviours of the C language, ruling out any *going-wrong* behaviour may seem a daunting task. In general, this requires ensuring both memory safety and control-flow integrity. The following two sections describe how we can exploit the SFI transformation and the knowledge that all memory accesses are inside the sandbox to ensure both memory safety and control-flow integrity.

2.2.3 Memory-safe Masking

For SFI, memory safety is obtained by making sure that every memory access is performed inside the sandbox. Starting from an analysis of the standard SFI solution, we present our own design which satisfies the additional requirements of being compliant with the semantic restrictions of COMPCERT and with a strict interpretation of the C standard.

2.2.3.1 Standard SFI Masking of Addresses

Standard SFI transformations ensure memory safety by masking memory accesses and was presented Section 2.1.1.3. The gist of it is to allocate a sandbox sb of size 2^k at a 2^k aligned memory address, say $\&sb = tag \times 2^k$. Under those constraints, enforcing that an address A is within the bounds of the sandbox can essentially be done by replacing the high-address bits by those of tag. Using bitwise operations, this can be done by the expression $(A\&(2^k-1))|tag \times 2^k$, where $\&$ is the bitwise *and* and $|$ is the bitwise *or*. More visually, this can be written $(A\& \underbrace{1\cdots1}_{k})|tag\ \underbrace{0\cdots0}_{k}$.

At binary level, this masking transformation is defined and the cost is modest: two bitwise operations. However, this masking operation has no well-defined C semantics. This is also the case for the semantics of COMPCERT and in particular for the CMINOR language. The reason is twofold: bitwise operations over pointer values return $\mathbf{undef}$ and concrete addresses (e.g. $tag \times 2^k$) are not pointers for COMPCERT where they are represented by a block and an offset (see Figure 2.15).

2.2.3.2 Specialised Masking for 32-bit Sandboxes

For 32-bit sandboxes, there exists a variant of the sandboxing primitive which has the advantages 1) that the sandbox address does not need to be aligned; 2) that the cost of masking may be reduced to a single instruction. In its simplest form, the masking primitive is defined by

$$\&sb + (A - \&sb)_{64\to32\to64}$$

where $\&sb$ is the symbolic address of the sandbox. The subtraction of $\&sb$ extracts the offset of the pointer and the double (unsigned) cast $64 \to 32 \to 64$ has the effect of truncating the offset to a 32-bit quantity that is therefore within the bounds of a 32-bit sandbox. At first sight, this masking is less efficient than the standard masking but it is efficient for typical address computations which require both displacement and scaling (e.g. $A = t + k + k' * i_{32\to64}$ where t is a 64-bit address, k and k' are constants and i is a 32-bit integer). Assuming that each cast or arithmetic operation is mapped to a single instruction[1], the masked address A can be computed using 8 instructions: 4 instructions for computing the address A and 4 more for the sandboxing primitive. Using simple properties of modular arithmetic, it is possible to distribute the $64 \to 32$ cast over addition and multiplication to obtain the following equivalent formulation of the sandboxed address:

$$\&sb + A'_{32\to64} \quad with \quad A' = t_{64\to32} + c_1 + c_2 * i$$

where c_1 and c_2 are compile-time constants: $c_1 = (k - \&sb)_{64\to32}$ and $c_2 = k'_{64\to32}$. Using this formulation, the address A' still requires 4 instructions but the cost of the sandboxing is reduced to 2 instructions making it on par with the standard sandboxing. On x86, 32-bit registers are just zero-extended 64-bit registers. Therefore, the cast $A'_{32\to64}$ is

1. Some architecture have rich addressing modes allowing for more compact encodings.

actually redundant and the overhead induced by the sandboxing is reduced to a single instruction. Our experiments (see Section 2.2.7.2) validate the practical advantage of this encoding.

Still, as for the standard sandboxing, this sanboxing primitive has no semantics in CompCert due to the limitations of pointer arithmetic. As a consequence, the solution of Kroll *et al.* [60] does not give actual code for the masking primitive, but rather axiomatise its behaviour as an external function. This prevents optimisations such as common subexpression elimination or function inlining from happening and induces the cost of a function call for each memory access.

2.2.3.3 Towards Well-defined Pointer Arithmetic

To illustrate the limitations of pointer arithmetic, we examine the semantic behaviour of the standard sandboxing primitive (the specialised sandboxing primitive has similar issues). The standard sandboxing primitive can be written $(A\&(2^k{-}1))\,|\,\&sb$ where $\&sb$ is the address of the sandbox variable. If sb is allocated at runtime at address $tag \times 2^k$ for some tag, this formulation is equivalent at binary level. Again, this heavily relies on pointer arithmetic that is undefined and on information about where the sandbox is linked at runtime.

Consider the alternative formulation $(A\&(2^k{-}1)) + \&sb$ where the bitwise $|$ is replaced by a $+$. This formulation has the advantage that incrementing a pointer, here sb, is well-defined (see Figure 2.15). As on modern hardware, both addition and bitwise operations take a single cycle, the difference in efficiency should be negligible. Moreover, at least for x86, the addition can be compiled into the addressing mode.

Still, this does not solve our issue. To understand this, suppose that A is a pointer. In this case, the bitwise $\&$, whose purpose is to extract the pointer offset, is still undefined. Therefore, the whole expression $(A\&(2^k{-}1))+\&sb$ is undefined. Because dereferencing an undefined expression is a *going-wrong* behaviour, the compiled program may have an arbitrary runtime behaviour and escape the sandbox. A prerequisite for our masking primitive is therefore to ensure that the evaluation is defined i.e., different from $\mathbf{undef}$. As all the semantic operators of CompCert are strict in $\mathbf{undef}$ (if any argument is $\mathbf{undef}$, so is the result), a necessary condition is that A is not $\mathbf{undef}$. As A can be obtained from any expression, a challenge is to ensure that every expression evaluates to a defined value. A particular difficulty is that the many undefined pointer operations (see Figure 2.15) cannot be detected by runtime checks.

2.2.3.4 Arithmetisation of the Heap

To tackle this challenge and ensure that every computation is defined, we propose an original and radical approach which ensures syntactically that pointers are neither stored in memory nor in local variables. As a result, the program is only manipulating integer values and memory addresses are only constructed by the sandboxing primitives. This approach implies, as a side-effect, that our previously undefined masking primitives are defined. Let asb be the runtime address of the symbolic address $\&sb$ of the sandbox. The masking of an address A can be written

$$A' + \&sb$$

where A' is either defined by $A' = A\&(2^k{-}1)$ or $A' = (A - asb)_{64\to32\to64}$. As A is necessarily an integer, A' is necessarily a defined integer and therefore $A' + \&sb$ returns a defined pointer $\mathbf{ptr}(sb, o)$ that is necessarily inside the sandbox.

An additional subtlety is that memory accesses are indexed by a memory chunk κ which mandates an alignment constraint (e.g. the chunk i_{64} mandates an 8-byte aligned address). As a result, the masking primitive is parameterised by the chunk κ and the masking primitive for i_{64} is $A'\&msk_{i_{64}} + \&sb$ where $msk_{i_{64}} = (2^{k-3}{-}1) \times 2^3$.

Only computing over numeric values is facilitated by the fact that the sandboxed program is only manipulating pointers relative to a single object, the sandbox. Therefore, a solution could be to only compute with pointer offsets. This is not totally satisfactory because the null pointer (i.e., 0) would be undistinguishable from the base pointer $\mathbf{ptr}(sb, 0)$. Instead, we use the integer asb that is the integer runtime address of the sandbox (i.e., we have $asb = \&sb$) and perform the following transformation t over program expressions.

$$\begin{array}{lcl} t(\&sb) & = & asb \\ t(c) & = & c \text{ for } c \in \{i32, i64, f32, f64\} \\ t(\triangleright\, e) & = & \blacktriangleright t(e) \\ t(e_1 \,\square\, e_2) & = & t(e_1) \blacksquare t(e_2) \\ t([e]_\kappa) & = & [msk_\kappa(t(e))] \end{array}$$

The operators $\blacktriangleright$ and $\blacksquare$ ensure that, if the expressions are well-typed, they never return the $\mathbf{undef}$ value. Typical examples include division, modulus, and bitwise shifts. We transform expressions so that they evaluate to an arbitrary value when their original

semantics is undefined. For example, we transform the left-shift operations on 32-bit integers so that the resulting expression always has a shift amount less than 32:

$$\texttt{a<<b} \quad \rightsquigarrow \quad \texttt{a << (b \& 31)}.$$

Similarly, we transform divisions and modulus in the following way, to rule out the undefined cases of division by zero and signed division of `MIN_SIGNED` by `-1`:

$$\texttt{a/b} \rightsquigarrow \texttt{(a+(a==MIN\_SIGNED \& b==-1))/(b+(b==0))}.$$

We can prove that the resulting division expression is always defined. Most of the other expressions are always defined and do not need further transformations.

2.2.4 Enforcement of Control-Flow Integrity

Correct sandboxing of code requires some degree of control-flow integrity. Existing SFI implementations enforce a weak form of control-flow integrity which only ensures that jumps are aligned and within a sandbox of code. This is achieved by inserting a masking operation before indirect jumps, that will mask the target address to ensure that the jump is within the sandbox. Additional padding with no-ops is inserted to ensure that all the instructions are indeed aligned [117, 99]. We enforce a stronger, more traditional, form of control-flow integrity where any control-flow transfer has a well-defined CMINOR semantics.

2.2.4.1 Relaxation of the CMINOR SFI Property

Intraprocedural control-flow integrity is ensured by simple syntactic checks. For instance, they ensure that a $\mathbf{goto}\ lb$ has a corresponding label lb and that an $\mathbf{exit}\ n$ has at least n enclosing blocks. The semantics of CMINOR prescribes that function calls and returns necessarily match. For this to still hold at the assembly level where the return address is explicitly stored in the stack frame, it is sufficient to prove that the CMINOR program has no *going-wrong* behaviour. To ensure control-flow integrity, the only remaining issue is due to indirect calls through function pointers. Our control-flow integrity counter-measure implements software trampolines and ensures that an indirect call with signature σ can only be resolved by a function pointer towards a function

with signature σ.

For this purpose, the existing CMINOR SFI security policy i.e., Property 1, which rules out any memory access outside the sandbox is too restrictive. As we shall see, the implementation of trampolines necessitates controlled memory reads, outside the sandbox, within compiler-generated variables. To accommodate for this extension, we propose a slightly relaxed SFI security property which, in addition to memory accesses inside the sandbox, authorises other memory reads in read-only regions.

Property 3. *A* CMINOR *program is secure if all its memory accesses are within either the sandbox block* sb *or some read-only memory.*

This relaxed property still ensures the integrity of the runtime because all memory writes are confined to the sandbox. Note that Property 3 and Property 1 are equivalent if the trusted runtime library has no read-only memory. This can be achieved at modest cost by modifying slightly the source code and remove the C type qualifier `const` which instructs the compiler that the memory is read-only.

2.2.4.2 Control-Flow Integrity of Indirect Calls

In Section 2.2.3, we have eluded the presence of function pointers. They actually perfectly fit our strategy of encoding pointers by integers. In this case, each function pointer is encoded as an index and the trampoline code translates the index into a valid function pointer.

Consider a function f of signature σ and suppose that the function pointer $\&f$ is compiled into the index i. The reverse mapping from indexes to function pointers is obtained from a compiler-generated array variable A_σ such that $A_\sigma[i] = \&f$. The array variable A_σ is made of all the function pointers with signature σ. The array variable is also padded with a default function pointer such that its length is a power of two. At the call site, the instruction $e(e_1 \ldots, e_n)_\sigma$ is transformed into $[te \& msk_\sigma + \& A_\sigma](te_1, \ldots, te_n)_\sigma$ where te, $te_1 \ldots$, te_n are transformed expressions such that all memory accesses are masked and msk_σ is the binary mask ensuring that the index te is within the bounds of the variable A_σ. In our actual implementation, we optimise direct calls and in this case, bypass the trampoline. Therefore, when the expression e is a constant pointer $\&f$ to an existing function with signature σ, we generate directly $(\&f)(te_1 \ldots, te_n)$. As a result, only C code using indirect calls goes through the trampoline code.

Though our implementation only exploits the relaxation of Property 3 for the sake of trampolines, a more aggressive implementation could sometimes avoid relocating read-only memory inside the sandbox. This could have a positive impact on optimisations which exploit the immutability of read-only memory.

2.2.5 Safety and Security Proofs

We next give an overview of our fully verified Coq proof of security and safety.

2.2.5.1 Security Proof

Property 3 is an informal formulation of our security property that is formally stated as a CMINOR instrumented semantics. This semantics mimics the CMINOR semantics with the exception that memory accesses are restricted: a memory read is either performed within the sandbox or in a read-only memory region; a memory write is necessarily performed within the sandbox.

The goal of the security proof is to show that all the memory accesses abide by the restrictions of the instrumented semantics. This is stated by Theorem 2 which establishes that for a transformed program tp, no behaviour of the standard CMINOR semantics gets stuck for the instrumented CMINOR semantics.

Theorem 2 (Security)**.** *For any transformed program tp, every behaviour of tp in the standard semantics of* CMINOR *is also a behaviour of tp in the instrumented semantics.*

The proof is based on the standard technique of forward simulation that is used in COMPCERT to ensure the preservation of semantics by compiler passes. Here, the forward simulation has the distinctive feature of relating the same (transformed) program equipped with a standard and an instrumented semantics. Since the only difference between the two semantics is that memory accesses must be secure, the crux of the proof lies in the correctness of the masking primitive, as stated in the following lemma.

Lemma 1. *For any masked expression e, if e evaluates to some pointer $\mathbf{ptr}(b, o)$, then b is the block of the sandbox i.e., sb.*

The proof relies on the definition of the masking primitive: a masked expression e is of the form $e' + \&sb$. Since $\&sb$ evaluates to the pointer $\mathbf{ptr}(sb, 0)$, then if the whole expression evaluates to a pointer $\mathbf{ptr}(b, o)$, necessarily $b = sb$.

2.2.5.2 Safety Proof

In order to benefit from COMPCERT's semantic preservation theorem and transport our security proof to the compiled assembly program, we must also prove that the sandboxed program is safe, i.e., it never gets *stuck*. We address all the going-wrong behaviours that we enumerated in Section 2.2.1.1.a. The well-formedness properties of a program (calling only defined functions, accessing only defined variables, jumping only to defined labels, exiting from no more blocks than currently enclosed in) are checked statically and make the transformation fail if they are violated. Next, the memory accesses require the addresses to be valid and adequately aligned: our masking operation ensures that this is always the case. Then, the evaluation of expressions must always be defined: this has mostly been dealt with the arithmetisation of the memory (Section 2.2.3.4). Finally, function calls should always be performed with the appropriate number of well-typed arguments. This is easy to check statically for direct function calls, but requires trampolines (as described in Section 2.2.4.2) for indirect function calls. The following sandbox invariant encapsulates all these conditions.

Definition 1 (Sandbox Invariant)**.** *A state S of program P satisfies the sandbox invariant if the following conditions are satisfied:*

1. *indirect control-flow transfers are well-defined in P (e.g. `goto` instructions in the functions of P only jump to defined labels);*
2. *every function of P ends with an explicit return;*
3. *every function of P is well-typed;*
4. *every function of P starts by explicitly initialising its local variables;*
5. *the global array A_σ for signature σ contains function pointers to functions of signature σ;*
6. *the environment for local variables and the memory in S only contain properly initialised, numerical values.*

Properties 1, 2, 3 are ensured by a set of syntactic checks over the bodies of all the functions of the program. Property 4 is enforced by our function transformation which inserts assignments that explicitly initialise all declared local variables. Property 5 is ensured by construction of the arrays for function pointers. All these properties can be established solely on the program body and do not change during the execution of

the program. By contrast, Property 6 cannot be checked statically and depends on the state of the program at each point.

2.2.5.2.a Safe Evaluation of Expressions.

A necessary condition for the safe evaluation of expressions is that the program is well typed. COMPCERT does not generate these type guarantees so we have integrated a verified (simple) type-inference algorithm for CMINOR programs. Type-checking alone is not sufficient to rule out undefined behaviours of C operators, but together with the transformations explained in Section 2.2.3.4, we prove the following lemma about the evaluation of transformed expressions.

Lemma 2 (Safe evaluation of expressions)**.** *In a memory state and a well-typed environment for local variables containing only defined numerical values, the transformation of any well-typed expression e evaluates to a defined numerical value.*

Lemma 2 follows directly from the properties of our expression transformation.

2.2.5.2.b Safety of Calls through Trampolines.

As mentioned in Section 2.2.4, we implement software trampolines to secure function calls through function pointers. To ensure the safety of indirect function calls, we maintain a map $smap$ from function signatures to the corresponding array identifier and the length of this array. The proof of safety relies on the fact that for every function f of signature σ present in a program, we have $smap(\sigma) = (A_\sigma, l_\sigma)$ such that all offsets lower than l_σ in A_σ contain a pointer to a function of signature σ. The safety proof of indirect calls itself is not hard, but we need to set up this signature map and establish invariants relating it to the global environment of the program.

2.2.5.2.c Safety Theorem.

Considering the invariants defined in Definition 1, we prove Lemma 3 which is our main technical result.

Lemma 3 (Safety)**.** *For any* CMINOR *program state S that satisfies the invariants, either S is a final state or there exists a sequence of steps from S to some S' such that S' also satisfies the invariants.*

A subtlety of the proof is that at function entry, the local variables carry the value $\mathbf{undef}$ and therefore the sandbox invariant only holds after they have been initialised by a sequence of assignments (see Property 4 of Definition 1).

Using Lemma 3, we can show Property 2, in the form of Theorem 3.

Theorem 3 (Safety of the transformation)**.** *All behaviours of the transformed program are well-defined, i.e., not wrong.*

Proof. A going-wrong behaviour occurs precisely when a state is reached, from which no further step can be taken, though it is not a final state. Lemma 3, together with a proof that the initial state of the transformed prorgam satisfies the invariants, tells us that no such reachable state exists, concluding the proof. □

As a result, we benefit from COMPCERT's semantic preservation theorem and can transport the security proof down to the assembly program.

Theorem 4 (Security of the compiled program)**.** *Let p be a transformed* CMINOR *program. If p compiles into the assembly program tp, then tp is secure.*

The proof uses Corollary 1 and Theorem 2 to conclude that the behaviours of tp are the same as those of p, and hence secure.

2.2.6 SFI Runtime and Library

Our modified COMPCERT compiler, COMPCERTSFI, takes as input a C program unit in the form of a list of C files. Each C file is first compiled down to the CMINOR language using the existing passes of the COMPCERT compiler. Then, all the CMINOR programs are syntactically linked [54] together to form the program unit to be isolated inside the sandbox. COMPCERTSFI comes with a lightweight runtime and a generic support for interfacing with a trusted library (e.g. a libC). An originality of our approach is that the runtime is using a standard program loader. Moreover, the runtime gets some of its configuration through compiler-generated variables.

2.2.6.1 Loading the SFI Application

The sandboxed code is linked with our runtime library by a linker script which specifies where to load at runtime the sb variable, viewed as the data segment. The compiler

also emits a sandbox configuration map which contains the symbolic address of the sandbox, its numeric value at runtime, the total size of the sandbox and the range of addresses reserved for global variables.

Our runtime code is executed before starting the sandboxed `main` function. It first checks that the sandbox is properly linked according to the sandbox configuration map, sets the shadow-stack pointer and initialises the sandbox heap using our sandbox-aware implementation of `malloc` based on `ptmalloc3` [2].

By construction, our runtime stack is free of buffer overruns. Yet, if the recursion is too deep, the stack may overflow. Therefore, the runtime inserts an unmapped page guard at the bottom of the stack and intercepts the segmentation fault. This protection suffices provided that the size of each function stack frame does not exceed a page; which can be checked at compile-time. Eventually, after copying its arguments inside the sandbox, the runtime calls the `main` function of the sandboxed application.

2.2.6.2 Monitoring Calls to the Runtime Library

The runtime library is trusted and therefore part of the TCB. To ensure isolation, each call towards the runtime library is monitored to check the validity of the arguments. For this purpose, a call to a library function, say `foo`, is renamed in the object file into a call to a function `sb_foo` which sanitises its arguments before really calling the function `foo`. The verifications are library specific but usually straightforward to implement. For `stdio`, the `FILE` structures are allocated by the runtime outside of the sandbox. Hence, the returned `FILE*` cannot be dereferenced to corrupt the `FILE` structure. To prevent the sandboxed program to forge `FILE*` pointers, the runtime maintains at all time the set of valid `FILE*`. For variadic functions *e.g.*, `printf`, we statically compile the format into a sequence of safe primitive calls. (We reject programs using formats computed at runtime). For functions in `string`, we check beforehand that the range of memory accesses is within the range of the sandbox. We also allow callbacks and therefore a runtime function may take a function pointer as argument. To ensure that the function is valid, the runtime is using the trampoline programming pattern presented in Section 2.2.4.2.

2. `http://www.malloc.de/malloc/ptmalloc3-current.tar.gz`

2.2.6.3 Communication via Global Variables

Programs may not only communicate *via* function calls but also directly *via* global variables. For the libC, this includes e.g. `stdout` or `errno`. To ensure isolation, COMPCERTSFI relocates those variables inside the sandbox but also generates a global variable map which is an array variable of the form

$$\{\&n_1, o_1, \ldots, \&n_i, o_i, \ldots, \&n_m, o_m\}$$

where $\&n_i$ is the symbolic address of a global variable and o_i is its offset in the sandbox. Using this information, the runtime has the ability to synchronise the values of the variables inside and outside the sandbox. For example, at program startup, the value of `stdout` (a `stream` pointer) is copied inside the sandbox at the relevant offset. This allows the sandboxed program to call `stdio` functions but protects the integrity of the stream. For `errno`, it is the responsibility of each runtime library call to synchronise the value of `errno` in the sandbox.

2.2.7 Experiments

We have evaluated our PSFI approach over the COMPCERT benchmark suite and a port of QUAKE. All the experiments have been carried over a quad-core Intel 6600U laptop at 2.6GHz with 16GB of RAM running Linux Fedora 27. For QUAKE, we explain how to adapt the code to our runtime library and verify the absence of noticeable slowdown. For the other benchmarks, we make a more detailed performance evaluation and compare COMPCERTSFI with COMPCERT, GCC, CLANG but also the state-of-the-art (P)NaCl implementation of SFI. In our experiments, all the benchmarks are ordered by increasing running time. Moreover, for computing a runtime overhead, the running time is obtained by taking the harmonic mean of 3 consecutive runs.

2.2.7.1 Porting Quake

QUAKE engines come in various flavours and we use the tyr-quake [3] implementation linking with XLIB. The port requires the addition of several functions to our runtime library from XLIB and the LIBC. Most of them are not problematic and require no or little

3. `https://disenchant.net/git/tyrquake.git`

modification. For instance, the `getopt` function which is used to parse command-line options is using the global variables `optarg`, `optind`, `opterr`, and `optopt`. As explained in Section 2.2.6.3, the runtime library copies the values of these variables at reserved places inside the sandbox.

Other functions, *e.g.* `gethostbyname`, allocate memory on their own and return a pointer to this piece of data which is therefore not accessible to the sandboxed code. For the specific case of `gethostbyname`, the library provides the function `gethostbyname_r` which, instead of allocating memory, takes as argument a data-structure that is filled by the function. In our case, we pass as argument a sandbox allocated piece of memory. This does not solve our problem entirely as inner pointers may still point outside the sandbox. To cope with this issue, we perform a deep copy of the relevant piece of data inside the sandbox.

A last issue is that the video memory is shared between the application and the X server using the system call `shmat`. Fortunately, the libC provides the relevant flags to bind shared memory at a specific address. Hence, we were able to allocate it inside the sandbox thus allowing a seamless communication with the X server. After these modifications, the sandboxed QUAKE runs without noticeable slowdown which is encouraging and an indication of the good overall performance of our sandboxing technique. In the following, we complement this with a more precise runtime evaluation for the COMPCERT benchmarks.

2.2.7.2 PSFI Overhead: Impact of Sandboxing Primitives

Next, we compare the efficiency of a standard masking primitive (Section 2.2.3.1) with a specialised version for 32-bit sandboxes (Section 2.2.3.2).

Figure 2.17 shows the overhead of the standard sandboxing primitive with respect to the specialised sandboxing primitive. There are 6 benchmarks for which the overhead incurred by the standard sandboxing is above 10% reaching 40% for 2 benchmarks. These cases illustrate the significant performance advantage that is sometime obtained by the specialised sandboxing. For some benchmarks, the standard sandboxing outperforms our optimised sandboxing. Yet when it does it is by a very small margin (below 3%). Overall, for the vast majority of our benchmarks, the specialised sandboxing primitive is very competitive.

In Section 2.2.3.1, we gave theoretical arguments for the advantage of the specialised sandboxing. Another argument comes from the fact that the specialised sand-

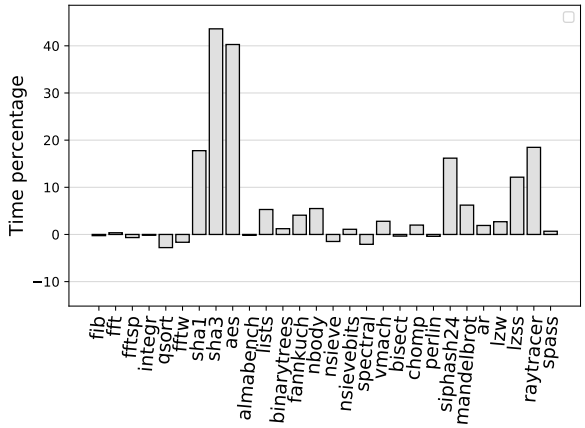


Figure 2.17 – Overhead of standard w.r.t specialised sandboxing

boxing is easier to optimise. First, note that the standard and the specialised sandboxing primitives are both using a bitwise mask but for different purposes. For the standard primitive, it is used to enforce that the pointer is within the sandbox bounds but also to enforce alignment constraints. For the specialised primitive, it is only used to enforce alignment constraints. Using the existing COMPCERT dataflow framework, we have implemented an alignment analysis that is quite effective at removing redundant alignment masks. To enable more optimisations, we explicit alignment constraints in the CMINOR code program (e.g. by specifying that function arguments of a pointer type are necessarily aligned). Thus, our experimental results are explained by both the theoretical advantages given in Section 2.2.3.2 and the effectiveness of our alignment analysis.

2.2.7.3 PSFI Overhead: Impact of Compiler Back-end

As a second experiment, we evaluate the overhead of our PSFI transformation for various compilers: COMPCERT, GCC and CLANG. COMPCERT is a *moderately optimising compiler* and the benchmarks run significantly faster using GCC and CLANG. In Figure 2.18, the baseline is given by the minimum of the execution times of the three compilers without PSFI instrumentation. The black bar is the overhead of a compiler (e.g. COMPCERT), with respect to the baseline and the grey bar is the overhead of the same compiler but with the PSFI transformation (e.g. COMPCERTSFI). In order to use GCC and CLANG, we implement a trusted decompiler from our secured CMINOR programs to CLIGHT, a subset of C in COMPCERT. These CLIGHT programs are then

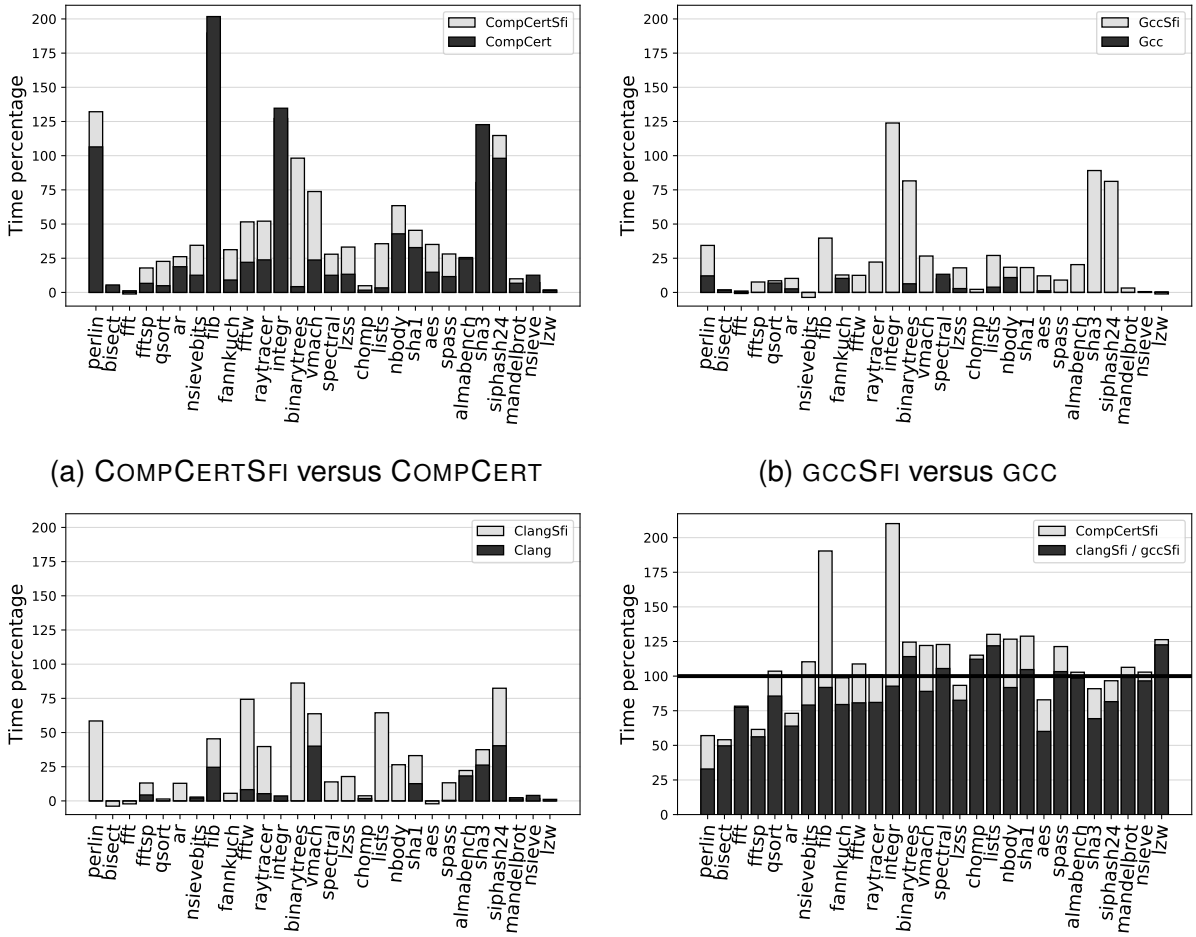
compiled with GCC or CLANG.

For a fair comparison, we should compare programs for which we actually have a reasonable security guarantee. We have a formal proof of security and safety (see Section 2.2.5) for the sandboxed CMINOR program, and we are confident that our syntax-directed decompiler preserves this property. For COMPCERT, this would suffice to preserve the security of the compiled CLIGHT code, but this is not the case for GCC and CLANG because of semantic discrepancies between the compilers. To limit this risk, we have set the compiler flags to instruct GCC and CLANG to adhere to the specificity of COMPCERT semantics: signed integer arithmetic is defined and so are wraps around (flag `-fwrapv`), strict aliasing is irrelevant (flag `-fno-strict-aliasing`), and floating-point arithmetic is strictly IEEE 754 compliant (flags `-frounding-math` and `-fsignaling-nans`). We also instruct the compilers to ignore any knowledge about the C library (`-fno-builtin`).

Our experimental results are shown in Figure 2.18. In Figure 2.18a, we have the overhead of COMPCERT and COMPCERTSFI. The overhead of COMPCERT over GCC and CLANG is expected and corroborates existing results [4]. For 10% of the benchmarks, the overhead COMPCERTSFI over COMPCERT is negligible and sometimes the PSFI transformation even improves performance. Those are programs for which the PSFI transformation introduces few masking operations, if any. For 41% of the benchmarks, the overhead is below 10% and can be considered, for most applications, a reasonable efficiency/security trade-off. For all the other benchmarks except `binarytrees` and `vmach`, the overhead is below 25%. The two remaining benchmarks have a significant overhead reaching 82% for `binarytrees`. This corresponds to programs which are memory intensive and where sandboxing cannot be optimised.

In Figure 2.18b and Figure 2.18c, we perform the same experiments but with GCC and CLANG. The results have some similarities but also have visible differences. For about 60% of the benchmarks the overhead is below 20%. Moreover, for both compilers, the average overhead is similar: 22% for GCCSFI and 24% for CLANGSFI. Yet, on average GCCSFI makes a better job at optimising our benchmarks and best CLANGSFI for about 75% of the benchmarks. For the rest of the benchmarks, we observe a significant overhead, up to 20%, indicating that the PSFI transformation hinders certain aggressive optimisations. The results also seem to indicate that optimisations are fragile as the overhead is not always consistent across compilers. The case of the

4. `http://compcert.inria.fr/compcert-C.html#perfs`



(a) COMPCERTSFI versus COMPCERT

(b) GCCSFI versus GCC

(c) CLANGSFI versus CLANG

(d) PSFI versus (P)NaCl

Figure 2.18 – Overhead of PSFI:COMPCERT, CLANG, GCC, (P)NaCl

`integr` benchmark is particularly striking because it runs with negligible overhead for CLANGSFI but exhibits the worst case overhead for GCCSFI. The `integr` program is using a function pointer inside a loop and we suspect that GCCSFI, unlike CLANGSFI, fails to optimise the program due to the inserted trampoline code. Though less striking, the benchmarks `fftw` and `raytracer` follow the opposite trend; these are programs where the overhead of CLANGSFI is much higher than GCCSFI.

2.2.7.4 PSFI versus (P)NaCl

We also compare our compiler-based SFI approach with (P)NaCl [99], which to our knowledge is one of the most mature implementations of SFI. Figure 2.18d shows the

overhead of COMPCERTSFI, GCCSFI, CLANGSFI with respect to (P)NaCl. The baseline is given by the best among NaCl and PNaCl. The best of CLANGSFI and GCCSFI is given in dark gray and COMPCERTSFI is given in light grey.

We first analyse the results of COMPCERTSFI. Our benchmarks are ordered by increasing runtime. The first 5 benchmarks have a runtime below one second. They are not representative of the performance of both approaches but only illustrate the fact that (P)NaCl has a startup penalty due to the verification of the binary and the setup of the sandbox. The overhead peaks above 75% for two programs (i.e., `fib` and `integr`). As the PSFI transformation keeps `fib` unmodified and only inserts a trampoline call in `integr`, these programs only highlight the limited optimisations performed by COMPCERT. Of the remaining benchmarks, 40% of them run faster or have similar speed with COMPCERTSFI. For those benchmarks, the average overhead of COMPCERTSFI w.r.t (P)NaCl is around 9%. Except for a few programs whose overhead skyrockets due to COMPCERT not being specialised for speed, we can say that COMPCERTSFI performance is comparable to (P)NaCl, having programs with better speed in both sides and a large number having similar results.

We also matched GCCSFI/CLANGSFI against (P)NaCl to compare the impact on performance of more aggressive optimisations. Here 60% of the programs are faster with GCCSFI/CLANGSFI. Among the remaining programs, `lzw` and `chomp` are programs for which the (P)NaCl code runs faster than the optimised GCC CLANG code without the PSFI transformation. As (P)NaCl is based on CLANG, more investigation is needed to understand this paradox that may be explained by code running outside the sandbox *i.e.* the trusted runtime library. Among the remaining benchmarks, `binarytrees` and `lists` still show a noticeable overhead. Those are recursive micro-benchmarks for which our PSFI is costly (see Figure 2.18). For `lists`, 99% of the time is spent in a tight loop where only a single address is masked. For `binarytrees`, 70% of the time is spent in the runtime code of `malloc` and `free` and therefore this highlights the fact that our implementation is less efficient than the (P)NaCl counterpart. Overall these results indicate that our implementation of SFI is competitive with (P)NaCl, given similar compilers. Furthermore, speed can be improved with more sandbox-dedicated optimisations; these would be harder for (P)NaCl to check.

2.2.8 Related Work

Since Wahbe et al. [114] proposed their initial technique for SFI, there has been a number of proposals for efficiently confining untrusted software to a memory sandbox (see [73, 98, 117, 76, 120, 106, 101]). One of the most prominent is Google's Native Client (NaCl) [117], which provides an infrastructure for executing untrusted native code in a web browser. NaCl was specifically targeted at executing computation-intensive applications without incurring a performance penalty. Certain features (in particular self-modifying code) were ruled out. These restrictions were addressed in a subsequent work [6].

RockSalt [77] is an SFI verifier for x86 code which has been developed and formally verified with the proof assistant Coq. The major contribution of RockSalt is to provide a formal model of the x86 architecture, from which it is possible to extract a decoder for a subset of the very rich set of x86 instructions, and build a verifier for the NaCl sandbox policy. Their experiments show that the formally verified checker performs marginally better than the NaCl verifier. In comparison, our approach avoids the complexities of the x86 instruction set by relying on the COMPCERT compiler back-end to produce binaries whose adherence to the sandbox policy is guaranteed by a combination of a sandbox verification at a higher level (CMINOR) and the COMPCERT's correctness theorem.

ARMor [120] is using the binary rewriter Diablo [89] to implement SFI for ARM processors. Using an untrusted program analysis, a proof of SFI safety is automatically constructed using the HOL theorem prover. ARMor was tested with some programs of the MiBench benchmark [44], namely BitCount and StringSearch. These programs required 2.5 and 8 hours respectively to prove the memory safety and control-flow integrity of the executables, which means that the approach is not practically viable as it is.

Kroll et al. [60] proposed PSFI as an alternative methodology to the standard, verification-based SFI. In PSFI, the sandbox is built by inserting the necessary masking instructions during compilation. This means that the correctness of the transformation can be argued at an intermediate stage in the compilation where the program representation retains a high-level structure. Our work extends the seminal proposal in a number of ways that we detail below. Unlike Kroll et al., we exclude from the TCB the masking primitive and the trampoline mechanism for calling external functions. In our implementation, these crucial components are written entirely in CMINOR and proved

correct without introducing trusted, unproved, code. Kroll et al. sketch a proof of safety but do not identify the issue of pointer arithmetic. To sidestep the semantics limitation of pointer arithmetic, we introduce a compile-time encoding of pointer as integers. This transformation is instrumental for our Coq verified proof of safety, which itself is mandatory to transfer security down to assembly.

Since the seminal work of Norrish [84], several works propose formal semantics of the C language [59, 37, 46]. All these share the limitations of CompCert with respect to pointer arithmetic. Recent works specifically aim at providing a more defined semantics for pointers. The proposal of Besson *et al.* [16] is able to cope with most existing low-level pointer manipulations and has been ported to CompCert [17, 18]. Yet, it has nonetheless limitations and the design of our PSFI transformation would not benefit from the increased expressiveness. The semantics of Kang *et al.* [54] is more permissive because, after a cast, a pointer is indistinguishable from an integer value. To our knowledge, their semantics has not been ported to the CompCert compiler. Our SFI transformation has the advantage of being compatible with the existing semantics of CompCert with the caveat that pointers need to be explicitly compiled into integers.

2.2.9 Conclusion

We have presented CompCertSfi, a formally verified implementation of Software Fault Isolation based on the CompCert compiler. Our approach provides security guarantees at runtime when the source code may be malicious or has security vulnerabilities but the build process is trusted. This is typically the case when a final product is built using code originating from multiple third parties. Our work shows that it is possible to perform security-enhancing compilation that is both formally verified and competitive with existing approaches in terms of efficiency. CompCertSfi does not rely on *a posteriori* binary verification for guaranteeing security, and hence has a reduced TCB compared to traditional SFI solutions. The reduction in TCB is obtained through a formal, machine-checked proof of the fact that the security guaranteed by our SFI transformation in the compiler front-end, still holds at the assembly level. Key to achieving this property has been to fine-tune the transformation (and in particular its pointer manipulations) to ensure that the secured program has a well-defined semantics.

The impact of SFI has been evaluated on a series of benchmarks, showing that the

transformed code can in a few cases be more efficient, and that the average runtime overhead incurred is about 9%. We have evaluated the impact of back-end optimisation on the transformed code on three different compilers. The gains vary, with CLANG being more efficient than COMPCERT and GCC, and COMPCERT being slightly more efficient than GCC. The experiments show that COMPCERTSFI combined with an aggressive back-end optimiser can sometimes achieve performances superior to Native Client implementations. In addition, there is still room for further optimisation of the generated code. We have observed that existing optimisations are sometimes hindered by our SFI transformation, so we gain by having more optimisation before the SFI transformation. We also intend to investigate optimisations for removing redundant sandboxing operations and in particular hoisting sandboxing outside loops.

CHAPTER 3

INFORMATION-FLOW PRESERVING TRANSFORMATION

This chapter will present another way of addressing security during compilation. Previously with COMPCERTSFI, regardless of the source program, the compiled program will always conform to SFI policy. Here we will talk about preservation of security during compilation. The goal is that the compiled program needs to be at least as secure as the source program. In other words, security guarantees of the compiled program should be equivalent to the security guarantees of the source. In this setting, the view of security is vastly different from SFI where the source code was untrusted. With preservation, we believe that the developers are responsible for enforcing the security of their programs. The compiler's job is solely to translate the intent of the developers down to the executables.
Unfortunately, it is well known that compilers struggle to fulfill this task correctly [115], and the situation worsen when accounting for side-channels. Side-channels target behaviours of the systems that are not usually described in programming language standards. Therefore, compilers may misunderstand side-channel countermeasures and compromise them which results into serious security vulnerabilities in the compiled programs.

In this chapter, we present a solution to tackle this issue. Our work mainly focuses on side channel attacks which can probe the memory of the system during an execution such as power analysis or cold boot attacks. First, we will present the context and the motivations behind our work. Then we will explore different researches which inspire us for our work and point out how they fare for our problematic. Afterwards we will present our solution to preserve security against a memory attacker during compilation using our notion of *Information-Flow Preservation* (IFP). We also show our IFP property is used to secure two compiler passes: dead store elimination and register allocation. Finally, we will discuss how to extend our property to other compilation passes and

how can improve the property for future work.

3.1 Motivations

In this section, we present the different motivations behind our work. We first present the attacker model we want to prevail against, and then show few examples where compilation fails to preserve security of programs. Lastly, we discuss few security properties that mitigate information leakage in programs and that we would like to preserve during compilation.

3.1.1 Memory probing attacker

Our overall goal is to prevent information leaks from being introduced by the compilation process. Compiled code should be no more vulnerable to passive side-channel attacks than the source code. Such side channels correspond to an attacker who is granted physical memory access at specific observation points, and who is parametrised by the amount of information he is allowed to read. At the semantic level, side-channels can be modelled using a leakage function exposing a partial view of the program state to the attacker [11].

A typical example of a timing channel is the leaking of information through observations of the memory cache behaviour. Formally, this channel can be modelled by leaking the program counter and the memory accesses [5]. Another example is the power channel that can be modelled by the so-called Hamming Weight model [56] where what is leaked is the Hamming Weight of the program state *i.e.*, the number of information bits that are set to 1.

In our work, we want to protect against a category of attackers which are able to probe the memory state of a program during its execution. This attacker model is suitable to use for concrete attacks such as power analysis attack, electromagnetic waves analysis attacks or cold boot attacks. The three attacks cited are able to probe the content of the memory whenever during a program execution. Furthermore, for the first two attacks, the cost of mounting the attack is directly related to the amount of information one wants to get from the memories. Therefore, we want our property to also preserve the amount of information leaked in a program.

3.1.2 Information Flow Preservation by Examples

We present a list of simple program transformations which break security of programs using the syntax of an imperative language where a • indicates program points where the program's memory is leaked to the attacker.

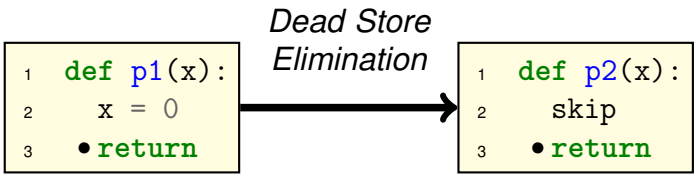


Figure 3.1 – Direct leakage

Data remanence Figure 3.1 is an example where an optimisation like *Dead Store Elimination* (DSE) creates a direct leakage in the program `p1`. DSE is the typical example when talking about compiler inability to preserve security of source programs. A description of DSE can be found in the thesis introduction. In program `p1`, the value of the variable `x` is erased by the instruction `x = 0` before returning. Thus, its initial value will not remain in the memory at the end of the program. DSE can optimise away the erasure instruction hence breaking safe erasure of `p1` since the attacker can read the initial value of `x` when `p2` returns.

Lifetime extension Variables lifetime may be extended due to optimisations like code motion which moves certain line or block of code to improve the speed of the code. An example is to extract an instruction out of a loop to avoid redundancy or to improve locality of variables that are often used together.

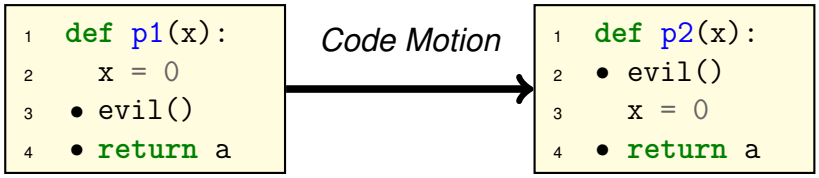


Figure 3.2 – Lifetime extension

In Figure 3.2 we assume that an attacker is able to read the memory state just before calling the function `evil`. Hence in `p1` an erasure instruction is placed right before (line 3) to prevent the value of `x` from being leaked. Unfortunately, code motion

may change the execution order of the instructions and place the erasure instruction after the vulnerable function `evil` which defeats its purpose and breaks safe erasure. Extending the lifetime of sensitive values increases the potential vulnerabilities of a program which is something we want to capture with our preservation property.

Worsening of leakage We also want to prevent a leak from increasing during a transformation. For example, in Figure 3.3 the program `p1` wants to protect a secret value which is `x + y`. It is unable to protect it from leaking but mitigates the leak by splitting the full value into two shares `x` and `y` and avoid keeping `x + y` in memory. In this situation, a leak is present since an attacker is able to get the secret value `x + y` but he is still required to uncover the value of both `x` and `y` to compute it which increases the difficulty of the attack. This is similar to masking, presented in Section 3.1.4.2, where secrets are split into shares and operated on independently before reconstructing the final value.

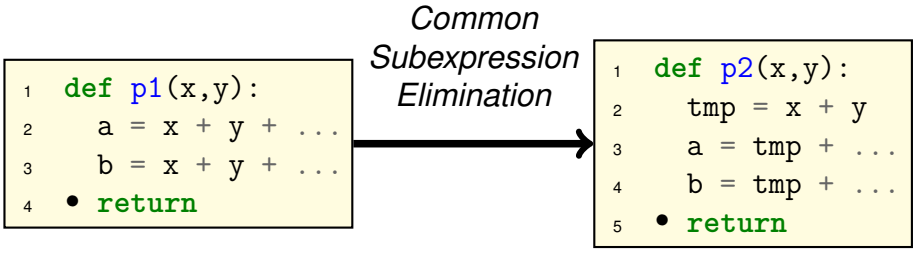


Figure 3.3 – Worsening of leakage

However, a compilation optimisation like *Common Subexpression Elimination* can decide to store temporarily the value `x + y` to avoid redundancy during computations. Therefore, an attacker is able to get the secret solely by looking at the value of the variable `tmp` in `p2` while he needed to both look at `x` and `y` in `p1`. This worsening of leakage should be captured by our preservation property since we do not want a program to get less secure than it was before during compilation.

Duplication Lastly, we believe that having multiple instances of a secret value also multiplies the possible attacks that can be carried out on a program. A transformation should not make this possible but unfortunately this may happen during transformations like *Register Allocation* (RA).

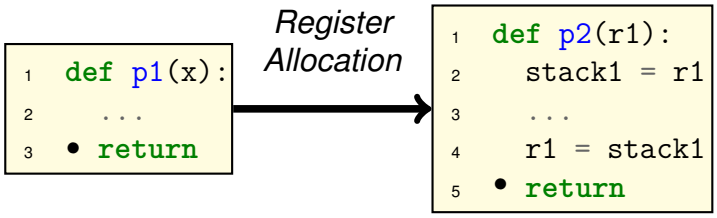


Figure 3.4 – Duplication

```
1   f: •1 a = x ^ y;  x = 0; •2 return a;
2
3   g: •1 a = x ^ y; •2 return a;
```

Figure 3.5 – Safe Erasure of One-Time Pad

RA is a pass that takes into account the limited number of registers of a processor and uses the stack to alleviate the data contained in the registers. RA is tackled in our work and is presented thoroughly in Section 3.3.4. In program `p1` of Figure 3.4 the value we want to protect is `x` and is leaked at the end of the program. This leak is aggravated in `p2` during RA. In `p2` the secret value is stored in the register `r1` at the beginning of the program. This value is then stored in the stack line 2 to free `r1` for other computations. At the end of line 4 the secret value is restored in `r1`, however it is still present in the stack which makes two instances of the secrets in the memory and worsen the initial leakage.

3.1.3 The "Full Information Flow" Paradox

We work with a parametrised attacker model where the attacker makes partial observations. Partial observations are essential, because they capture partial information flows that would go unnoticed if we only had an omniscient attacker observing the whole memory. In the context of dynamic information flow policies, Askarov and Chong [7] have already observed that a program could be secure against a powerful attacker but insecure for a weaker attacker. To see this, consider the program f and the transformed program g in Figure 3.5. Using a cryptographic analogy, Program f is encoding the plain-text y using the one-time pad key x and erases the key x. Program g performs the same encryption without erasing the key. If the attacker observes $x\bullet_2$ in Program g, he obviously learns the key x. Perhaps surprisingly, the key x can also be

learnt for Program f by an attacker observing both the plain-text $y\bullet_2$ and the cipher-text $a\bullet_2$. The attacker obtains x by solving the equation $a\bullet_2 = x\hat{}y\bullet_2$ whose unique solution is $x = a\bullet_2\hat{}y\bullet_2$. Thus, no additional information flow is introduced and the transformation is (erroneously) deemed secure. To rule out this insecure transformation, we stipulate that both attackers need to observe the same amount of information *i.e.*, the same number n of bits. If both attackers observe a single bit, the observation $a\bullet_2$ in g cannot be simulated in f and therefore we conclude that the transformation is, as expected, not information-flow preserving.

3.1.4 Some security properties against information leakage

We just went through some examples in the previous section to get a feel of our property. Here we discuss three software properties which are *Safe Erasure* [27], *Masking* [42] and *Non-Interference* [41]. These three properties allow a program to limit the amount of information leaked during an execution and are generic to most kind of attacks. Safe erasure guarantees that secret values should not be observable anymore by an attacker past a point of execution. This reduces the lifespan of sensitive values in the memory which decreases the attack surface on the temporal dimension. Masking split a sensitive value into multiple independent shares. Computations are then done on these shares separately which means that an attacker need to retrieve all the different shares to be able to get the secret value. The cost of an attack becomes steeper depending on the number of shares used. Non-Interference is a really strong property that when enforced prevents an attacker from discerning between multiple executions of a program using different secret values. Contrary to safe erasure and masking, non-interference is not a mitigation property. Indeed, safe erasure and masking makes attacks on a program more difficult to achieve but secret values will be leaked anyway if we assume an attacker with unrestrained observations. Non-interference states that two executions of a non-interfering program should be indistinguishable to an attacker and therefore contains no leaks.

3.1.4.1 Safe Erasure

Safe erasure is a simple security measure that consists in wiping from the memory the secrets that are not used anymore and is illustrated in Figure 3.1. The idea is that

since attacks are able to retrieve values from the memory, the time where secrets are stored in the memory should be as short as possible.

The problem of secure erasure of secrets has been studied from an information-flow perspective. Chong and Myers [27] propose semantics foundations for defining erasure policies. A main insight behind that work is that erasure can be seen as the dual of declassification [94]. Later on, Chong and Myers [26], but also Hunt and Sands [48], propose type-systems for verifying erasure policies of the source code on WHILE languages. Askarov *et al.* enforce the erasure policies in the presence of so-called write-once locations which cannot be overwritten. The key insight is to store encrypted data in write-once locations and simulate erasure by the erasure of the cryptographic key. However, those different works focus on logical erasure and not physical erasure: make sure that the data is effectively erased from the media. Safe erasure principles are evident but in practice, compiler optimisations are known to remove such erasure instructions and is even listed as a CWE [30] and recognised by the CERT at CMU [1].

Dead Store Elimination (DSE) is the main culprit for removing erasure instructions and break security [116] during compilation. Reports about DSE breaking erasure can easily be found in multiple projects involving compilation or security. GCC relayed this vulnerability in 2002 [85] and the problem was still ongoing in 2015 with OpenSSL [91]. Numerous solutions have been proposed but to our knowledge there is no consensus of a preferred solution in the community.

[104] and [116] acknowledge this issue and survey the different techniques used to effectively erase data from the memory in C programs. Windows systems possessed a `SecureZeroMemory()` function that is never optimized away by compilers but is specific to this operating system. Standard C11 recommends using `memset_s` which is not widely adopted (gcc does not want to support it). Another countermeasure is to use the `volatile` qualifier which indicates that a variable may be modified by another source than the program. This method is a trick for to preserve erasure since `volatile` semantic has no relation to erasure and therefore does not completely address the issue [47]. All of these solutions possess their flaws and no ideal candidate has been found so far. To mitigate the problem, Simon *et al.* [104] also proposes a variant of the Clang compiler where the stack and the registers are wiped after each sensitive function execution. While this approach gives extra security, it is not flexible and comes with non-negligible overhead. Similarly, Yang *et al.* [116] also leverage the Clang compiler

1. CERT MSC06-C

and modifies the dead store elimination pass by combining the most reliable techniques surveyed to preserve the erasure instructions. Experiments seem promising; however no formal guarantees have been expressed and wide adoption of the compiler has yet to happen.

3.1.4.2 Masking

The idea of masking is to split secret data into multiple independent and random shares. Cryptographic implementations will apply their computations on these different shares independently and we can recover the final value by recombining the shares. Masking has two main advantages, first, an attacker needs to get all the shares to reconstruct the secret, which makes the attacks more costly. Second, all these shares which are chosen randomly, add additional noise to the computation which cannot be predicted since they are changed for every execution. This idea has been actively improved during the past years and have been implemented to both software [95] and hardware [49] and been proven secure [88].

Masking is especially used against power analysis attacks (Section 1.1.2.2) considered as a powerful side-channel attack. However, masking can also be used against other attack like cold boot attack.

We give the basic properties of a *Threshold Implementation* which is a popular model for masking [35][82].

Threshold Implementation We take the initial program p that takes as input the secret x and output the value y, $(p, x) \Downarrow y$. We note $\bar{p}$ the threshold implementation of p with masking that takes as input the share vector $\bar{x}$ and output the vector $\bar{y}$. $\bar{x}$ = $(x_1, \ldots, x_n)$ and $\bar{y}$ = $(y_1, \ldots, y_n)$ are composed of n elements. $\bar{p}$ respects the following properties:

— **Uniform Masking**, all the possible input share vector $\bar{x}$ have an equal chance to occur

— **Correctness**, the reconstruction of $\bar{y}$ should return the value y of the original program p

— **Non-completeness**, all component functions of $\bar{p}$ use at most $n - 1$ input shares of $\bar{x}$

— **Uniformity**, given a uniform input vector $\bar{x}$ then the output vector $\bar{y}$ returned by $\bar{p}$ should be uniform too

The key property for security in a threshold implementation is the non-completeness and is the property we aim to preserve. Uniform masking is about the inputs of the program and correctness is not the goal of our work but is preserved under a semantic preserving transformation similar to the CompCert theorem. Non-completeness guarantees the security of masking and declares that no intermediate values should be related to all the n input shares. Therefore, an attacker should not be able to deduce the whole input vector $\bar{x}$ from observing a limited number of intermediate values. We will not deal with uniformity in our work since we do not have a probabilistic model for our program executions.

3.1.4.3 Non-Interference

Non-interference [41] is mostly known as a multi-level security policy. The goal is to ensure that information-flow from different level of security do not interfere with each other. The flow policy which is usually used forbids any information flow from a high security level to a lower one. However, in our model we do not have any notion of security level. Indeed, for our probing attacker model all the values stored in the memory are equally accessible to the attackers: one does not need higher authorization to access certain parts of the memory. Therefore, the classical notion of non-interference cannot be preserved during compilation, since, in our setting we do not differentiate between high and low security memory locations. Hence, we use another notion of non-interference called *Observational Non-Interference* from Barthe *et al.* [11] that we adapt to our model.

Definition 2 (Observational Non-Interference)**.** *Program p is non-interfering with regard to the policy (ϕ, ψ) if:*

$$\forall (c, c'). \begin{pmatrix} (p, c) \Downarrow t \\ \wedge \\ (p, c') \Downarrow t' \end{pmatrix} \wedge \phi(c, c') \quad \Rightarrow \quad \psi(t, t')$$

$(p, c) \Downarrow t$ signifies that program p executed with initial memory c yields the trace t. Exact definition of a trace will be provided later in Section 3.3.1.1.

Choice of the policy (ϕ, ψ) depends on the security we want to enforce. A usual choice for ϕ is that memories should agree on addresses with low security levels. Definition of ψ depends on the leakage model but you often want that the traces observed by the attackers to be indistinguishable, so a good choice for ψ is equality.

3.2 Related works

Overall our goal is to preserve security countermeasures against a memory probing attackers during compiler transformations. Secure refinement shares similar issues with our topic, therefore we give an overview of the different works we found. Since our attacker model is close to side-channel attackers (cold boot, power analysis) we first explore several works concerning side-channels and look at existing countermeasures and see how they are dealt with during compilation. Secondly, we elaborate on preservation of security properties during transformations. Part of it was presented in the previous section on safe erasure, masking and non-interference.

3.2.1 Secure refinement

Safety and liveness properties that can be defined on a single trace are preserved under refinement [52]. However, this is not the case for information-flow properties (defined on a set of traces) and this issue has been pointed out since 1989 by Jacob [52]. Information-Flow properties is a class of properties that include many security properties like constant-time or non-interference in general. While our work focuses on program transformations, refinement of systems specifications shares this similar issue on preservation of security properties. System specifications are defined by a set of traces which are usually composed of events between entities of the system. The refinement from an abstract specification T_a to a concrete specification T_c reduces the underspecification of T_a which entails the following relation: $T_c \subseteq T_a$. A transformation, on the other hand, is defined as a computable function mapping a specification to another specification [97]. Information-Flow properties are defined by two components: a flow-policy and a definition of information-flow. In its simplest form, a flow policy is composed of two domains H and L and the relation $H \not\rightsquigarrow L$ which forbids flows from H to L. For deterministic systems, non-interference is accepted as the standard definition of information-flow. Mantel [72], defines a framework to check and construct refinement

operators that preserve information-flow properties and such, for different definitions of information-flow. These operators specify under which conditions low and high events can be disabled or should stay enabled during refinement. Moreover, he also proves that these refinement operators are optimal in the sense that any further disabling of events breaks the information-flow property chosen. Seehusen *et al.* [97] take the opposite approach and propose a schema in which secure specifications can be defined and are, by construction, preserved by a category of refinement called property refinement. The main idea behind their schema is to make a clear differentiation between underspecification and unpredictability. Underspecification corresponds to the traces that should be removed during refinement whereas unpredictability corresponds to the sets of traces which are necessary to have security. Indeed information-flow properties are defined on sets of traces and if elements of these sets are removed during refinement, the properties may not hold anymore. Hence, they propose to define specifications with a set of obligations which contain the minimal traces which are necessary to enforce an information-flow property. In this setting, refinement removes underspecification but does not lessen unpredictability. Another result of their work is an additional sufficient condition to adapt their schema to transformations instead of refinements. While the field of secure refinement of systems differs from our topic, problematics and security definitions are similar to ours and can inspire our work on secure transformations.

3.2.2 Power Analysis

Power analysis attacks belongs to a category of attackers which resembles the most to our attacker model. While simple power analysis only leaks values which are used by instructions with specific power usage such as branching, this is not the case for differential power analysis (explained in Section 1.1.2.2) (DPA). DPA is a statistical attack using multiple executions of a program to correlate the value of certain bits to its power consumption. Therefore, attackers using DPA are able to get almost any intermediate value from an execution given enough observations. This is similar to our memory probing attacker which has limited observational power.

Numerous proposals to defend against power analysis have been submitted with each their advantages and flaws. *Elimination techniques* pursue the goals of reducing the leakage observable by the attacker by minimizing the signal to noise ratio. Kocher *et al.* [56] in their paper already suggest several proposals such as balancing the ham-

ming distance [87][51], which ensures that an equal number of $0 \rightarrow 1$ and $1 \rightarrow 0$ bit transitions occur at each clock cycle. Other ideas involve choosing instructions with low impact on power consumption or directly shielding the device.

The main issue of these elimination techniques is that they do not fare well against an attacker with unlimited observations using DPA.
Another popular possibility is to use *masking* [25] presented previously Section 3.1.4.2. Masking can be both be implemented in software and hardware. Surprisingly, we did not find any work concerning the preservation of software masking during compilation. Moreover, we know from experimentations that generic compilers like GCC or Clang can compromise this countermeasure. Therefore, a motivation of our work is to tackle this issue.

3.2.3 Preservation of Constant-Time implementations

Timing attack is one of the most infamous side-channel attacks with a large body of work [55, 9, 14]. Even though constant-time has a different attacker model (the attacker cannot probe the whole memory), works about preservation of constant-time countermeasures can be found in the literature. Since differential power analysis and timing attacks share similarities (passive attacker and requires multiple executions to succeed), we inspire from their proof techniques to preserve security during transformations.

Jasmin [3, 4] Jasmin is a compilation framework which goal is to easily develop high-speed and high-security cryptographic code. Jasmin compilation chain uses the Jasmin language, which is enough low level to generate predictable assembly code but still be architecture independent. To attain high security, Jasmin programs can be translated into Dafny [63] programs where memory safety and constant time security can be verified automatically using the Z3 SMT solver [78]. Furthermore, the compilation framework is designed to preserve such security properties down to the assembly code. First of all, the Jasmin compiler has been proven with Coq to be correct, more explicitly to preserve the semantics of the Jasmin programs. Moreover, Jasmin is also proven to preserve constant-time security: jumps and writes in the memory do not depend on secrets. They prove for each pass of compilation that the transformation does not introduce any additional leak using a standard notion of simulation. More precisely,

they show that for each execution of the source program the corresponding execution of the transformed program will have equal or less leakage. Note that due to the Jasmin language being low-level, some aggressive optimisations that cannot be proven using a simulation are not implemented in Jasmin.

Jasmin has recently been improved [4] to support vectorized instructions used by realistic cryptographic implementations. Furthermore, embeddings of the Jasmin language have been added to the EasyCrypt [12] proof assistant for provable security. Hence, EasyCrypt can be used to automatically verify properties, such as functional correctness or constant-time, of Jasmin programs. Coupled with Jasmin preservation properties, Almeida *et al.* [4] proposes a complete toolchain to design and implement verified high-speed cryptographic implementations.

Preserving side-channel countermeasures [11] Barthe *et. al.* present a framework to prove that a transformation preserves some security property using the example of constant-time. However, their work is generic and can be applied to prove the preservation of other side-channel countermeasures. They use an operational semantic to model the behaviour of programs where each step between states is labelled by a leakage function. The leakage function is then defined depending on the attacker model we want to prevail against. In the case of the following constant-time policy "no secrets are used for jumps and write" we have the following leakage function:

— boolean is leaked during control flow instructions

— memory address is leaked during memory writes and reads

In their work, they define their security preserving compiler around a notion named *Observationally Non-Interfering Program* that we use for our work. The novelty of their approach is the proof technique they propose to prove the preservation of observationally non-interference. Instead of using the standard simulation where one proves that the transformed program leak equally or less than the source one; they use a 2-simulations to prove that for two source executions that are non-interfering, their transformed executions will be non-interfering as well. The main advantage is that they do not require any constraints between the leakage of the source and the transformed. A transformed execution is allowed to leak more than its source as long it is still non-interfering with the other transformed executions.

This proof technique is applicable to all information-flow properties which are decidable by comparing at least two executions of a program.

3.2.4 Information-Flow preservation

Deng and Namjoshi [33, 34] introduce an information flow concept of *leaky triple* and ensure that compiler transformations preserve the information flow of programs. Leaky triples (a, b, c) are a triple of inputs where a and b are high security and c is low. (a, b, c) is a leaky triple for a program p if the executions with inputs (a, c) and (b, c) give different outputs. They define a transformation from program p to program p' to be secure if any leaky triple of p' is also a leaky triple for p. Their definition supplies the notion of relative security that we seek: the transformed program should be at least as secure as the source. The main limitation of their definition is that they do not include any notion of amount of information leaked. For example, the transformation from p to p' is secure in their terms even if p leaks one bit of a password on input c whereas p' leaks the whole password with c.

Their first work [33] present a secure algorithm for DSE. Given a list of dead assignments they remove the ones which may introduce a new information leak using a taint analysis. They also show that their DSE should not encounter the limitation we pinpointed earlier. An additional result of this work is a proof technique to prove a category of transformations secure. If a transformed program p' is a strict refinement R of the source p then the transformation is secure. More precisely, if two states (t, t') of p' have equal low variables $t =_L t'$ then their refined states (s, s') of p, also agree on their low variables $s =_L s'$. This result can certainly be adapted in our work to prove that certain non-leaky transformations (like *Constant Propagation*) are secure.

Their following work [34] focuses on *Single Static Assignment* (SSA) transformations. The SSA intermediate form assign every new value to a fresh variable, therefore all the variables of a SSA program are always assigned a single value during an execution. The main issue is that when transforming a program into SSA form, any instruction meant to limit a leakage (like an erasure) will be assigned to a fresh variable. Therefore, Deng and Namjoshi keep track of related SSA variables during the SSA transformation and regroup them back during the unSSA transform.

3.3 Information-Flow Preserving Transformations

In this work, we propose a formal definition of preservation of information leakage which, if verified by program transformations, ensures that the target code is not less

secure than the source code, with respect to a passive but strong attacker able to read an arbitrary amount of memory. We stress that our purpose is not to enforce a security property; neither at the source level, nor at the target level. What we seek to identify is a general property that provides a formal account of how optimisations increase the information leakage and therefore render the compiled program insecure.

Our attacker model (ability to read arbitrary memory) is quite strong and one may wonder whether compiler optimisations preserve the information leakage. In our work, we review some optimisations and show that some of them need to be adapted. Our contributions can be summarised as:

— We propose a notion of preservation of information leaks and assess its relevance against a list of simple transformations.

— We show how to strengthen our initial proposal to capture information leaks due to partial observations.

— We present sufficient conditions for the absence of information leaks which provide convenient reasoning principles.

— We review two optimisations, dead store elimination and register allocation; and show how they can be adapted and proven to preserve information leakage.

— An implementation and experiments with an information-flow preserving register allocation pass within the CompCert compiler.

The rest of the chapter is organised as follows. In Section 3.3.1, we give a formal definition of IFP relying on the notion of Attacker Knowledge [7]. Section 3.3.2 presents sufficient conditions for an IFP transformation which are easier to reason about in practice. Sections 3.3.3 and 3.3.4 show how the previous reasoning principles apply to two standard program optimisations: dead store elimination and register allocation. We discuss some extensions of our formal model in Section 3.4.

3.3.1 Information-Flow Preservation

In this section, we first define formally what is an IFP transformation and then we will go through some examples to show that this definition answers the motivations stated in Section 3.1.

3.3.1.1 Execution model

Without loss of generality, we assume that the program state is only made of a bit-addressable memory:

$$\begin{aligned} Mem &= Addr \to Bit \\ Bit &= \{0,1\} \end{aligned}$$

A structured program state, s, can always be mapped to a memory $m \in Mem$ at the cost of some encoding. For instance, a language using program variables would represent them by reserving certain memory addresses.

The semantics of a program p is given by a one-step deterministic transition relation $\cdot \to_e \cdot \subseteq Mem \times Mem$ where, $e \in E = \{\epsilon, o\}$ is either, ϵ, denoting a silent transition, or o indicating that the end state of the transition is leaked to the attacker. From an initial memory m^0, a run of a program p is given by a sequence of memories $m^0 \cdot m^1_{e_1} \cdots m^n_{e_n}$ such that for every $i < n$ we have $m^i \to_{e_i} m^{i+1}$. Given such a run, the view of the attacker is given by the sub-trace of leaked memories *i.e.* those memories resulting from a transition tagged o. Formally, we define the transition relation of the attacker

$$\cdot \rightsquigarrow \cdot \subseteq Mem \times Mem$$

as a sequence of silent transitions followed by a transition leaking its final memory:

$$\frac{m \to^*_\epsilon m' \quad m' \to_o m''}{m \rightsquigarrow m''}$$

As a result, from an initial memory m_0, the trace of memories $t = m^1 \cdots m^n$ that is leaked to the attacker is such that for every $i < n$ we have $m^i \rightsquigarrow m^{i+1}$. In the following, we write $(p, c) \Downarrow t$ for a trace of the attacker obtained by running the program p from an initial memory c.

3.3.1.2 Partial Attacker Knowledge

In our model, in order to protect against an arbitrary side-channel, we quantify over a hierarchy of attackers A_n who have access to the programs code and where n is the number of bits of information that the attacker is allowed to extract from the program memory during the program execution. Thus, we do not fix one leakage function *a priori* but rather consider protecting against a hierarchy of leakage functions.

This attacker model is sufficient to formally capture the leakage of information. However, it does not take into account the practical difficulty of a physical attack *i.e.* how hard it is to extract bits of information when the memory is indirectly observed through a noisy side-channel.

We next define precisely the information leaked through a (partial) trace of attacker observations, using the notion of Attacker Knowledge. Following [8], the Attacker Knowledge from a trace t of program p is defined by

$$\mathcal{K}^t(p) = \{c \in Mem \mid (p, c) \Downarrow t\},$$

i.e., the attacker knowledge corresponds to the initial memories that may lead to (and hence are indistinguishable by) the observation of the trace t. We generalise the notion of Attacker Knowledge to partial observations of n bits of the trace t. A partial observation o is a trace of partial memories, *i.e.*, a sequence of partial maps from addresses to bits:

$$(Addr \hookrightarrow Bit)^*$$

Partial memories $m, m' \in Addr \hookrightarrow Bit$ are ordered so that $m \sqsubseteq m'$ if m and m' agree on all the addresses where m is defined. Formally, this is the standard (point-wise) ordering of partial functions:

$$m \sqsubseteq m' \stackrel{\triangle}{=} \forall x \in dom(m).\ m(x) = m'(x).$$

Two partial traces are ordered if they have the same length and the memories (at the same position) are ordered using $\sqsubseteq$. Formally:

$$\frac{}{\epsilon \sqsubseteq \epsilon} \qquad \frac{m \sqsubseteq m' \quad o \sqsubseteq o'}{m \cdot o \sqsubseteq m' \cdot o'}.$$

The number of bits n of a partial trace o is written $|o|$ and is obtained by summing the number of bits defined by the partial memories m in o.

$$\begin{aligned} |\epsilon| &= 0 \\ |m \cdot o| &= |dom(m)| + |o|. \end{aligned}$$

We can then define the notion of partial observation formally as follows.

Definition 3 (Partial Observation)**.** *The set of partial observations of n bits of a trace t*

is defined by

$$Obs(t, n) = \{o : (Addr \hookrightarrow Bit)^* \mid o \sqsubseteq t \wedge |o| = n\}$$

Definition 4 generalises the standard notion of Attacker Knowledge to partial observations.

Definition 4. *For a program p and a trace t, the Partial Attacker Knowledge for a partial observation $o \in Obs(t, n)$ is given by*

$$\mathcal{K}_n^t(p, o) = \bigcup_{o \sqsubseteq u} \mathcal{K}^u(p).$$

In Definition 4, we quantify over all the complete traces $u \in Mem^*$ that are compatible with the partial observation o *i.e.*, $\{o \mid o \sqsubseteq u\}$. The Partial Attacker Knowledge is therefore defined as the union of the Attacker Knowledge for all the complete traces that are indistinguishable from the point of view of o.

3.3.1.3 Information-Flow Preserving Transformation

We shall use Partial Attacker Knowledge to define precisely what we mean by a secure transformation of a source program into a target program. Intuitively, a source program p is at least as vulnerable as a target program p' if any partial observation o' of a (target) trace of p' can always be matched by a partial observation o of a (source) trace of p such that the source observation o leaks more information than the target observation o'.

The notion of *matching observation* is formalised by a function mapping partial observations from $Obs(t', n)$ to $Obs(t, n)$. To forbid transformations that increase the lifetime of an information leak, we also enforce that the observations at the target and source level are performed in lock-step *i.e.*, at each instant, both attackers observe the same amount of information.

Definition 5. *The set of lock-step observation mappings Ω is defined by*

$$\Omega(t', t, n) = \{\omega : Obs(t', n) \to Obs(t, n) \mid \forall o', sup(o') = sup(\omega(o'))\}$$

where the support of an observation is defined by

$$sup(\epsilon) = \epsilon \qquad sup(m \cdot o) = |dom(m)| \cdot sup(o)$$

Using lock-step mappings, we are ready to define the security refinement of a source program p by a target program p'.

Definition 6. *A target program p' is at least as secure as p for an attacker observing n bits (written $p \preccurlyeq_n p'$) iff*

$$\forall c, t, t'.(p, c) \Downarrow t \wedge (p', c) \Downarrow t' \quad \Rightarrow \quad \exists \omega \in \Omega(t, t', n).\forall o'.\mathcal{K}_n^t(p, \omega(o')) \subseteq \mathcal{K}_n^{t'}(p', o')$$

p' is as secure as p for attacker of level n if for any executions from the same input c which gives traces t' and t we have:

— every partial observation o' on t' of n bits is matched by a corresponding observation $\omega(o')$ on t of the same level

— the knowledge obtained from o', $\mathcal{K}_n^{t'}(p', o')$ should be a superset of the knowledge obtained from $\omega(o)$, $\mathcal{K}_n^t(p, \omega(o))$.

We remind that if an attacker knowledge is small it means that the set of possible initial memories is small and therefore information gained by the attacker is precise. Therefore, the relation $\mathcal{K}_n^t(p, \omega(o')) \subseteq \mathcal{K}_n^{t'}(p', o')$ means the knowledge obtained from o' (transformed) is less precise than the knowledge obtained from $\omega(o')$ (source). This is according to our initial intuition that an attacker should not learn more information from the transformed program than from the source program.

In order to rule out the duplication of information, we can further restrict the function ω to be injective, by adding the constraint

$$\Omega_1(t', t, n) = Obs(t', n) \rightarrowtail Obs(t, n)$$

where $\rightarrowtail$ denotes the set of bijective functions.

In case of duplication of information, several target observations with the same Partial Attacker Knowledge would need to be mapped to the same source level observation. The fact that ω is injective rules out this possibility. In the following, we take the constraint on matching functions to be either Ω or $\Omega \cup \Omega_1$ and discuss when the constraint Ω_1 is a too strong requirement.

A transformation T is IFP if $T(p)$ is at least as secure as p for attackers with any level of observational power.

Definition 7. *A transformation* $T : Prog \to Prog$ *is IFP iff*

$$\forall p, \bigwedge_{n \in \mathbb{N}} (p \preccurlyeq_n T(p)).$$

3.3.1.4 Does it answer our expectations?

We show in this section that our definition of IFP solve the examples we've shown previously in Section 3.1.2. We wanted our property to be able to reject transformations where lifetime of variables is expanded, leakages are aggravated or duplication of values appear. To reject a transformation, we just need to find an attacker able to observe n bits which can retrieve more precise knowledge from the transformed program than from the source.

Lifetime extension We study our example of Figure 3.2. We take an attacker capable of observing the whole memory before the call of the function `evil`. In `p1` this attacker is unable to get the initial value of the variable `x` since it was erased, therefore his attacker knowledge $\mathcal{K}_1$ is equal to Mem. However, in `p2` the erasure instruction was moved right after the call to `evil()`. Hence the same attacker here is able to get exact value of `x` and its attacker knowledge $\mathcal{K}_2$ is the set of memories where the address of `x` contains the value it has observed. We have the relation $\mathcal{K}_1 \not\subseteq \mathcal{K}_2$ which means that our property effectively rejects this transformation.

Worsening of leakages Here we show how partial attackers can be used to capture a leakage introduced by a transformation. Instead of taking an attacker which can observe the whole memory, we take one which can observe n bits which is the size of a variable. In Figure 3.3, the attacker wants to get the value of the variable `x + y`. In `p1`, he is able to see bits of `x`, `y`, `a` and `b`. However, with its n bits of observation he is unable to recover the whole value of `x + y` and can get at most $n/2$ bits of the secret (by observing $n/2$ bits of `x` and `y`). Whereas in `p2` this attacker is able to do so by observing the variable `tmp` which contains the value `x + y`. Therefore, the knowledge of `x + y` is more precise in `p2` than in `p1` and the transformation is rejected by our property. Note that with an attacker with at least $2 * n$ bits of observation the transformation is secure since he is able to get the value of `x + y` by observing both `x` and `y` separately. Since our property requires the condition to hold for all the attackers, this transformation is

indeed rejected.

Duplication Finally we also show that our property enhanced with duplication also rejects the transformation of Figure 3.4. We also take the case of an attacker able to observe n bits (size of a variable). We remind that the for the duplication property, we require from the lockstep mapping from transformed to source observations to be injective. In `p2` there exists two observations that can reveal the whole secret (in `r1`) to our attacker. Indeed, both an observation on `r1` and an observation on `stack1` can give the attacker the value of the secret. Therefore, it is necessary to find at least two observations in `p1` that reveal the secret (in `x`). This is not possible to do in `p1` since only the observation on `x` enables our attacker to get the whole secret. Therefore, this transformation is also not IFP since it does not satisfy our duplication condition for an attacker with n bits of observation.

3.3.2 Sufficient Condition for IFP and its Proof Principle

The definition of an IFP transformation does not suggest an immediate composition proof principle, and proving directly that a program transformation abides to Definition 7 would be a daunting task. To prove that a transformation is IFP, we present a simulation-based proof technique, ensuring that a source memory can be simulated by a target memory.

3.3.2.1 Partition and Simulation Relation

In order to facilitate concrete proofs, we partition the memories leaked to the attacker. In program terms, a typical partitioning would assign to the same partition index i all the memory leaked by a given syntactic observation point $\bullet_i$. To each partition index i, we attach a simulation function $\alpha_i : Addr \rightarrow Addr + Bit$. Given a source memory m and a target memory m' mapped to the same partition index i, the mapping α_i explains how m can be effectively simulated by m'. This is done by showing how any bit in m' is either a constant (thus without any information) or a bit that is stored in m possibly at another address as indicated by α_i.

Example 2. *Consider the Programs f and g of Figure 3.6. For this simple case, each observation point $\bullet_i$ (both in the source and target program) would be mapped to the*

partition index $i \in \{1, 2\}$*. For the initial observation point* $\bullet_1$*, both memories are the same and therefore* α_1 *is the identity function* $\lambda a.a$*. For* $\bullet_2$*, the memories are the same except for the address* x *which carry the constant* 1 *in Program* g*. Therefore, we would have* $\alpha_2 = \lambda a.$*if* $a = x$ *then* 1 *else* a*.*

```
1        f: •1 x = 0; •2 return;
2
3        g: •1 x = 1; •2 return;
```

Figure 3.6 – Safe Erasure of One-Time Pad

As we show in Theorem 5, these are sufficient conditions to ensure that m' contains at most the same amount of information as m.

Traces are characterised by a function $id : Memory \to [1, \ldots, n]$ which links a memory of a trace to its partition index. Therefore, in the following, we write $m.id$ for the index of a memory m.

Definition 8. *Let* $\alpha_i : Addr \to Addr+Bit$ *be a simulation function indexed by* $i \in [1, \ldots, n]$*. A source memory* m *is simulated by a target memory* m' *(written* $m \sim_\alpha m'$*) iff we have* $m.id = m'.id = i$ *and*

$$\forall a'.m'(a') = \begin{cases} \alpha_i(a') \text{ if } \alpha_i(a') \in Bit \\ m(\alpha_i(a')) \text{ if } \alpha_i(a') \in Addr \end{cases}$$

Lifted to traces we get,

$$\frac{}{\epsilon \sim_\alpha \epsilon} \qquad \frac{m \sim_\alpha m' \quad t \sim_\alpha t'}{m \cdot t \sim_\alpha m' \cdot t'}$$

In order to avoid duplication of information and enforce the constraint Ω_1, the function α_i needs to be further constrained to forbid mapping distinct target addresses to the same source address *i.e.*, $\exists a \in Addr.a = \alpha_i(a_1) = \alpha_i(a_2) \Rightarrow a_1 = a_2$.

Using the previous definitions, our necessary conditions for IFP can be stated using Theorem 5.

Theorem 5. *Consider two programs* p *and* p'*, and an indexed simulation function* α_i : $Addr \to Addr + Bit$*. If*

$$\forall c, t, t'. \begin{pmatrix} (p, c) \Downarrow t \\ \wedge \\ (p', c) \Downarrow t' \end{pmatrix} \Rightarrow t \sim_\alpha t'$$

then $\forall n, p \preccurlyeq_n p'$*,* i.e.*, the transformation is IFP.*

Proof. Suppose $t \sim_\alpha t'$. We need to prove, for any n,

$$\exists \omega. \forall o'. \quad \mathcal{K}_n^t(p, \omega(o')) \subseteq \mathcal{K}_n^{t'}(p', o').$$

Let $t' = m'_0 \cdot \cdots \cdot m'_n$ and $t = m_0 \cdot \cdots \cdot m_n$. Remember that ω has type $Obs(t', n) \to Obs(t, n)$ and that observations are made in lock-step fashion. As a result, it is sufficient to provide a mapping $\omega_i : Obst(m'_i, k) \to Obs(m_i, k)$ which given a partial observation o' of the target memory m'_i reconstructs an observation $\omega_i(o')$ of the source memory m_i. As $m_i \sim_{\alpha_i} m'_i$, this can be done as follows. Suppose that $o'(a') \neq \bot$ for some address a'. If $\alpha_i(a') \in Addr$, we can obtain the same observation from memory m_i and $\omega_i(o')(a') = m_i(\alpha(a'))$. Otherwise, If $\alpha_i(a') \in Bit$, the observation does not provide any information and therefore it suffices to pick an arbitrary fresh address a, and have $\omega_i(o')(a) = m_i(a)$.

From this construction, for any partial observation $o' \in Obs(t', n)$ we can derive a partial observation of the source execution $\omega(o')$ which contains at least as much information as in o'. It follows that the target attacker knowledge obtained from an observation o' is always bigger than the source attacker knowledge of the observation $\omega(o')$ and therefore the property holds. □

Theorem 5 does not provide a complete characterisation of IFP transformations. It captures transformations where information is perhaps moved around but is otherwise unmodified. To illustrate the limitation, consider the following contrived example

$$x{:=}y \quad \to \quad x'{:=} \sim y$$

where the bit values of y are flipped bitwise. The variables x and x' contain the exact same information *i.e.*, the value of y. Yet, the transformation of values cannot be modelled by a simulation function α. We show that standard optimisations including register allocation and dead store elimination are in the scope of Theorem 5.

3.3.2.2 Simulation-Based Principle

The pre-condition of Theorem 5 can be proved using a lock-step backward simulation principle over the attacker semantics.

Definition 9 (Backward Simulation)**.** *A simulation function* $\alpha_i : Addr \to Addr + Bit$ *is a backward simulation if:*

1. *From the initial memory* c*, the first source memory* m *and target memory* m' *leaked to the attacker are in relation (*$m \sim_\alpha m'$*).*
2. *Given memories* $m_1 \sim_\alpha m'_1$*, for every attacker step of the target program* $m'_1 \rightsquigarrow m'_2$ *there exists a memory* m_2 *in the source program such that* $m_1 \rightsquigarrow m_2$ *and* $m_2 \sim_\alpha m'_2$*.*

Theorem 6. *Suppose two programs* p *and* p' *and a simulation function* α*. If there is a backward simulation (according to Definition 9) for* α*, then the pre-condition of Theorem 5 holds* i.e.*,*

$$\forall c, t, t'. \begin{pmatrix} (p, c) \Downarrow t \\ \wedge \\ (p', c) \Downarrow t' \end{pmatrix} \Rightarrow t \sim_\alpha t'.$$

Proof. The proof is by induction over the length of the trace t' and follows using Condition 1 of Definition 9 for the base case and Condition 2 of Definition 9 for the inductive case. □

Our proof technique is used in the next two sections to secure two compilations steps: *Dead Store Elimination* (DSE) and *Register Allocation* (RA). First, we present a modified DSE algorithm and prove that the transformation is IFP. Then, using a translation validation approach we built a RA validator for CompCert that checks if the program transformation is IFP. In case of failure the validator is also able to patch the transformed program to render the transformation IFP. Benchmarks and analysis for RA are also available in Section 3.3.5.

3.3.3 Securing Dead Store Elimination

DSE is the archetypical program transformation that is not IFP. Informally, a dead store is a memory write which is provably unnecessary to compute the program result. Hence, from an optimisation point of view, it is a perfectly legal transformation to remove a dead store instruction. Security-wise, as shown by one of our motivating example (see Figure 3.1), the transformation does not preserve information-flows. Indeed, an attacker may gain more knowledge by observing the value that is not overwritten due to the removal of a dead store.

A drastic solution would be to disable this optimisation. We propose a modified Dead Store Elimination optimisation based on a revised and strengthened notion of dead store.

3.3.3.1 Liveness based DSE

Dead stores are typically identified using a *liveness analysis* [81, p. 2.1.4]. A liveness analysis is a classic backward program analysis. For each program point, it computes an over-approximation of the *live* variables *i.e.* variables that are necessary to compute the program result. Dually, *dead* variables are those variables that are not live and a dead store is a memory write ($x = e$) where the variable x is dead. Therefore, a classic DSE performs a liveness analysis and removes dead stores.

Using our abstract model of programs based on observation points we define a trace property of a liveness analysis:

Definition 10 (Sound Liveness)**.** *Given a program p, a liveness analysis is a list of pair of set of addresses $((V_0, W_0), \ldots, (V_j, W_j)) \subseteq (Set(Addr) \times Set(Addr))^*$. With each pair associated to an observation point of p such that:*

$$\forall i, c, c', t, t'. \left(\begin{array}{c} (p, c) \Downarrow t \\ \wedge \\ (p, c') \Downarrow t' \\ \wedge \\ t(i).id = j \end{array} \right) \quad \Rightarrow \quad c \simeq_{V_j} c' \Rightarrow t(i) \simeq_{W_j} t'(i)$$

where

$$e \simeq_V e' \Leftrightarrow \forall x \in V. e(x) = e'(x)$$

for $(e, e') \in Memory \times Memory$.

Definition 10 says that if two input memories c and c' agree on the values of the input live addresses V_j then the memory states at partition index j agree on their live addresses W_j. For the purpose of optimisation, the sets W_j contain a minimum set of addresses *i.e.*, only those needed by the return statements of the different functions; the sets V_j being computed by a backward fixpoint iteration (see [81, p. 2.1.4] for details).

As DSE removes the dead statements, it keeps unchanged the values of live addresses. As a result, in our formal model, a DSE transformation can be characterised by Definition 11.

Definition 11 (Dead Store Elimination)**.** *Given a program p and a liveness analysis $((V_0, W_0), \ldots, (V_j, W_j))$ of p. The DSE transformation from p to p' is correct if p' is indistinguishable from p for all the live variables W. Formally, we have:*

$$\forall c. \begin{pmatrix} (p,c) \Downarrow t \\ \wedge \\ (p,c') \Downarrow t' \end{pmatrix} \quad \Rightarrow \quad \forall i. \begin{pmatrix} t(i).id = j \\ \Leftrightarrow \\ t'(i).id = j \end{pmatrix} \wedge t(i) \simeq_{W_j} t'(i)$$

This specification is partial. In particular, it says nothing about the dead variables but this is enough to conclude that these dead variables may leak information and violate our IFP property.

3.3.3.2 Shadow Store Elimination

In order to get an IFP DSE, we propose to extend the sets of addresses W_j to the whole set of addresses $Addr$. This transforms the current definition of DSE to:

$$\forall c. \begin{pmatrix} (p,c) \Downarrow t \\ \wedge \\ (p,c') \Downarrow t' \end{pmatrix} \Rightarrow t = t'$$

We can easily convince ourselves that such program transformation is IFP since for any attackers the programs p and p' are indistinguishable.

Interestingly, this result can be obtained in a non-intrusive way by only slightly modifying the initial condition of the liveness analysis. Indeed, it is sufficient to impose that, for each observation point of the program, every address is live. As liveness analysis is backward, this can always be done.

Theorem 7. *Given a program p and a liveness analysis $((V_0, Addr), \ldots, (V_j, Addr))$ of p, DSE becomes IFP and produces the program p'.*

Proof. Suppose that $(p,c) \Downarrow t$ and $(p',c) \Downarrow t'$. By Theorem 5, it suffices to prove that there exists a simulation function α such that $t \sim_\alpha t'$. By Definition 11 of a DSE

transformation, we have $t \simeq_{Addr} t'$ *i.e.*, $t = t'$. Taking α as the identity function we get $t \sim_\alpha t' \iff t = t'$. As a result, Theorem 7 holds. □

The effect of this modification is illustrated in Figure 3.7, only dead stores that are shadowed by a following store at the same address can be safely removed.

```
1 def hash1(text):
2   h = text * ...
3   text = 5
4   text = 0
5   return h    •
```

```
1 def hash2(text):
2   h = text * ...
3   Skip
4   Skip
5   return h    •
```

```
1 def hash3(text):
2   h = text * ...
3   Skip
4   text = 0
5   return h    •
```

Figure 3.7 – Original program (left) - Classic DSE (middle) - IFP DSE (right)

`hash1` is the original program after DSE. It computes a hash from the variable `text` and before returning there are two erasures on the variable `text` to wipe its value before returning. In a classic DSE only the value of the variable `h` matters. Therefore, the liveness analysis would start from $W = \{h\}$ and would deem that writes on `text` are unnecessary after the computation of `h` line 2. The result is shown in `hash2` where both erasures lines 3 and 4 are replaced by `Skip` instructions. For our IFP DSE, we impose the condition $W = Addr$. In this case, the liveness analysis will also take into account the final value of `text` before the observation point. The transformed program `hash3` has only its penultimate erasure removed which prevents any leakage of `text` initial value.

3.3.4 Translation Validation for Register Allocation

Translation validation [86] is a verification technique which consists in validating *a posteriori* (and automatically) that a program p' is obtained by a valid transformation of a program p. We adapt this principle and design a specialised algorithm to validate *a posteriori* whether programs obtained through the Register Allocation (RA) pass of the verified CompCert C compiler [66] satisfy our IFP property. This is done by explicitly constructing a backward simulation using the sufficient condition of Section 3.3.2. An interesting feature of our algorithm is that certain failures can be interpreted as potential information leak and that these leaks can be closed automatically by inserting erasing instructions.

3.3.4.1 Register Allocation in a Nutshell

Before RA, programs make use of an unbounded number of pseudo-registers. The role of RA is to explicit the constraint that the physical machine has only finitely many registers and therefore allocate each pseudo-register to a machine register. RA is also responsible for implementing calling-conventions *i.e.* passing arguments in the right register and restoring *callee-saved* registers. What makes RA a complex optimisation task is that this resource allocation task may be impossible due to a shortage of registers. In that case, a register may be *spilled* in the function stack frame *i.e.* its content copied, for later reuse. After the last use of a spilled register, a conventional RA algorithm has no reason to explicitly erase the stack location. This breaks our IFP property because i) the value is duplicated (it is stored in both a register and a stack location); ii) this introduces an information leak if the stack location is not erased after the last use of the register. This is illustrated by Example 3.

Example 3. *Consider the simple function* `cipher` *of Figure 3.8 and the function* `cipher2` *obtained by a typical RA pass for an hypothetical target architecture with only two registers* `r1` *and* `r2` *. The code has been arranged to highlight the relation*

```
1  def cipher(text):
2    key  = get_key()
3    salt = get_salt()
4
5
6    tmp  = text^salt
7
8    key  = key^tmp
9  • return key
```

```
1  def cipher2(stack_text):
2    r2   = get_key()
3    r1   = get_salt()
4    stack_key = r2        # spill
5    r2   = stack_text     # load
6    r2   = r2^r1
7    r1   = stack_key      # load
8    r1   = r1^r2
9  • return
```

Figure 3.8 – Original program (left) After register allocation (right)

between a source instruction and the corresponding sequence of target instructions. For instance, the source instruction of Line 6 is compiled into the sequence of target instructions of Lines 4-6. The Function `cipher2` *contains additional spilling/loading instructions inserted by RA Lines 4,5 and 7. In Function* `cipher` *the secret value in* `key` *is overwritten Line 8, hence an attacker at • cannot observe its value. However, in Function* `cipher2`*, the value of* `key` *in stored in register* `r2` *and copied in variable* `stack_key` *(see Line 4). As the variable* `stack_key` *is never erased, an attacker at • can observe*

its value. Hence this transformation is not IFP and will be rejected by our validation algorithm.

3.3.4.2 Register Allocation Languages of CompCert

In CompCert, the RA pass compiles a source in the Register Transfer Language (RTL) into a target in the Location Transfer Language (LTL). RTL and LTL are both fairly classic control-flow graph program representations where nodes are labelled with three-address code instructions.

Instructions representative of RTL are given below.

RTL instructions:

$$
\begin{aligned}
I \ni i \quad ::= \quad & \texttt{nop}(s) \\
| \quad & \texttt{op}(op, \overline{args}, dest, s) \\
| \quad & \texttt{load}(addr, \overline{args}, dest, s) \\
| \quad & \texttt{store}(addr, \overline{args}, src, s) \\
| \quad & \texttt{cond}(cond, \overline{args}, s_{true}, s_{false}) \\
| \quad & \texttt{call}(sig, fid, \overline{args}, dest, s) \\
| \quad & \texttt{return}(arg)
\end{aligned}
$$

Each instruction $i \in I$, attached to a node $n \in \mathbb{N}$, specifies its immediate successor s which is the node of the instruction to execute next. The `nop` instruction does nothing. The `op` instruction computes the value of the variable $dest$ using the values stored in the vector of pseudo-registers $\overline{args}$. The `load` instruction moves the value at the address computed by $\overline{args}$ with the addressing mode $addr$ to the destination $dest$. Similarly, `store` move the value from src to the address computed by $\overline{args}$ and $addr$. The `call` instruction calls the function fid with signature sig with parameters $\overline{args}$; the result of the call is stored in $dest$. The instruction `cond` models a conditional and has two successors. Depending on the value of arg, the instruction to execute next is either s_{true} of s_{false}. Finally, `return` exits the current function and return the value of arg.

The LTL language is similar to RTL with the differences that i) so-called locations corresponding to stack slots or machine registers are used in place of the unlimited pseudo-registers called temporaries and; ii) `call` and `return` instructions are bounded by the calling conventions of the target architecture. As a result, in the program syntax, $\overline{args}$ and $dest$ are locations made of either machine registers or stack slots used for spilling registers. Compared to temporaries, stack slots are pointers into the current stack frame and special care must be taken to make sure that spilled registers do not

overlap. The calling conventions stipulate where function arguments and return must be stored and are represented by a list of locations. For instance, for x86_32, arguments are passed on the stack and the result is stored in eax while for x86_64, arguments are passed in different registers depending of the type of the argument. LTL instructions are listed here:

LTL instructions:

$$
\begin{aligned}
I \ni i \quad ::= \quad & \texttt{nop}(s) \\
| \quad & \texttt{op}(op, \overline{args}, dest, s) \\
| \quad & \texttt{load}(addr, \overline{args}, dest, s) \\
| \quad & \texttt{store}(addr, \overline{args}, src, s) \\
| \quad & \texttt{cond}(cond, \overline{args}, s_{true}, s_{false}) \\
| \quad & \texttt{call}(sig, fid, s) \\
| \quad & \texttt{return}
\end{aligned}
$$

The only differences lie in the `call` and `return` instructions where the parameters and return value are not specified in the instructions but implicitly dictated by the calling conventions.

3.3.4.3 CompCert Translation Validation of RA

CompCert is using an untrusted RA algorithm whose output is verified using a specialised translation validation approach [92], ensuring that each target LTL function is a sound compilation of the source RTL function. Observational correctness of the whole program is then achieved by composing the validation of each pair of functions. As our own IFP validator is reusing some key components, we give a brief overview of their algorithm.

The translation validator of CompCert exploits that the RTL and LTL functions have a very similar structure and only differ in that the LTL function introduces *move* instructions to materialise spills and reloads of stack slots; and data movements between registers.

The untrusted RA algorithm takes as argument an RTL function and returns an LTL function. The LTL function is structured in such a way that it is straightforward to rebuild a mapping from a single RTL instruction to the list of LTL instructions it is compiled into. Because RA is only using a few local transformations, the list of LTL instructions is always made of *move* instructions *i.e.* assignments between locations, followed by the actual instruction which is obtained from the original RTL instruction by replacing

registers by locations. The only possibilities are presented in the first column of Figure 3.9. Each possible association of instructions are categorized into block-shape and form whole block-shape functions. Similar to RTL and LTL, block-shape functions are structured as a CFG and each block-shape is parametrised with the nodes of its successors. Moreover, each possible block-shape contains a list of move instructions from the LTL code, labelled mv in Figure 3.9, which are executed before the core instruction.

COMPCERT translation validation algorithm is composed of two parts. The first one is a structural check which verifies that the LTL function respects a certain structure with respect to the RTL function. This check is carried out while constructing the block-shape function. For example, to construct `BSop` the validator needs to find `op` instructions in both RTL and LTL and check that the operations match. Another example is `BScond` where the condition and successors must be the same in both `cond` instructions. If an unexpected pattern is found between the source and transformed functions then the structural check fails and so does the validation. To complete the translation validation algorithm, COMPCERT performs an additional backward data flow analysis to verify that the two functions effectively compute and use the same values. Similar to a liveness analysis, the backward analysis first assume that the return values are equal between RTL and LTL functions. From there the analysis go backward each instructions of the functions and compute the necessary conditions on values to verify the assumptions of the next instructions. For example, if a RTL program returns the variable x, then we have the singleton assumption $\{x = eax\}$ on the `return` instructions (eax comes from the calling conventions). Therefore, the predecessor instructions of these `return` must compute the necessary conditions to fulfill the assumption $\{x = eax\}$. This process continues to backtrack until reaching the entry point of the functions. Then a fixpoint computation is made to propagate the conditions to the whole functions. If no incoherence has been found while computing the conditions then the data flow analysis is successful. If the structural checks also succeeds, then the transformation is validated.

3.3.4.4 Modular IFP Validation Algorithm

Our security policy is determined by the location of observation points • in both the RTL and LTL programs. In order to get a translation validation algorithm integrated in the RA compiler pass, it is necessary to be able to process one function at a time. Our solution is to set observation points at function boundaries, more precisely at function

calls and returns. From a security point of view, this ensures that the LTL locations do not leak information at function return *i.e.* the stack frame of LTL only contains information that is also present in the pseudo-registers of RTL.

In the following, we detail our IFP validation algorithm. In Section 3.3.2, we have shown that proving IFP preservation can be done by exhibiting some mapping $\alpha_i : Addr \rightarrow Addr + Bit$ used to establish a simulation between the source and the target memories at every observation point $\bullet_i$. To get to this point, our IFP validator needs to construct richer objects in order to cater for the RTL and LTL CompCert memory model [68] and the fact that, for intermediate program points, the existence of such an α_i mapping is too strong a requirement. Hence, the IFP validator constructs a set of associations between locations and temporaries $\beta_i \in \mathcal{P}(Loc \times Temp)$; computes the set of modified location $\gamma_i \in \mathcal{P}(Loc)$ and performs a constant analysis $cst_i : Loc \rightarrow Value + \{\top\}$ such that, given a program point i,

— $(l, t) \in \beta_i$ iff the value of the LTL location l is the same value as the RTL temporary t,

— $l \in \gamma_i$ iff the location l may be modified by the current function,

— $cst_i(l) = v$ iff for any execution the location l always contains the value v.

The constant analysis cst_i and the set of modified locations γ_i are computed by iteratively solving standard forward data-flow equations using the existing data-flow framework of CompCert.

3.3.4.4.a Inference of Location Mapping β_i

Compared to the existing RA validator of CompCert, a difference is that we construct β_i using a forward data-flow analysis over the block-shapes of Figure 3.9. The reason is that, for compiler correctness, it is just necessary to ensure that the return LTL register, say eax, is mapped to the return RTL temporary, say t. For IFP, we need a complete mapping for all the RTL temporaries and LTL locations.

The transfer functions for the possible block shapes generated by RA can be found in Figure 3.9. The transfer functions are using the following notations. We write $(l \rightarrow t)$ for a pair $(l, t) \in \beta_i$ with the interpretation that the LTL location l is mapped to the RTL temporary t. We extend this notation to vectors of locations and temporaries and write $(l_1, \ldots, l_n) \rightarrow (t_1, \ldots, t_n)$ for the set $\{(l_1 \rightarrow t_1), \ldots, (l_n \rightarrow t_n)\}$. Because of copy instructions, in both LTL and RTL, the mapping is not unique and there may be several

pairs with l as first component and t as the second component. Given a temporary t, we write $(\_ \to t)$ for the set of all pairs such that the second element is t. Symmetrically, for a given location, we write $(l \to \_)$ for the set of all pairs such that the first element is l.

All block shapes have a sequence of move mv instructions on the LTL side. A move $l_1 = l_2$ assigns l_2 to l_1. Given an initial mapping B, the effect of move $l_1 = l_2$ is to remove the existing mappings for l_1 ($(l_1 \mapsto \_)$) and map l_1 to existing mappings of l_2 in B. As a result, we get

$$transfer_{l_1=l_2} = B \setminus (l_1 \to \_) \cup (\{l_1 \to x\} \mid \{l_2 \to x\} \in B).$$

For valid code, there should always be an existing mapping for l_2. If not, the analysis continues but l_2 and l_1 would be flagged as potential information leaks at the next observation point.

A `BSnop` only consists in a list of LTL move instruction and its effect is modelled by iterating the transfer function for a single move instruction. For `BSop`, we check after computing the consequence of the moves that the LTL arguments $\overline{args'}$ are mapped to the RTL arguments $\overline{args}$. If this is the case, we have the guarantee that the arguments $\overline{args}$ and $\overline{args'}$ have the same values and therefore the destinations tmp and loc also have the same value. After evaluating the effect of the moves, the transfer function invalidates the existing mappings for tmp and loc and adds the mapping $(loc \to tmp)$. For `BSLoad`, the transfer function is similar but exploits the additional invariant that the memory of RTL and LTL agree for every address a that is neither an LTL location $a \notin Loc$ nor a temporary $a \notin Temp$. For `BSstore`, we check that the computed addresses $\overline{args}$ and $\overline{args'}$ compute the same value in both RTL and LTL. We also check that the stored values are the same *i.e.*, the current mapping includes $(loc \to tmp)$. Except for the potential move instructions, a memory store has no effect on the current mapping. Yet, our verifications ensure that the LTL and RTL memory still agree for addresses that are neither temporaries nor locations. For `BScall`, we check that the arguments of the call are the same. In RTL, the arguments and return value are explicitly passed; in LTL, the functions $Params(sig)$ and $Ret(sig)$ implement the architecture dependent calling conventions of functions arguments and return. In RTL, all the temporaries are restored after a function call. In LTL, the calling conventions state that both stack locations and a subset of the registers *i.e.*, so called *callee-saved* registers, are restored after a call.

Block Shape	*Conditions*	*Transfer Function*
$\texttt{BSnop}(s)$: $\texttt{nop};$ | $mv;$ $\texttt{nop};$		$transfer_{mv}(mv, B)$
$\texttt{BSop}(s)$: $tmp := \texttt{op}(\overline{args});$ / $mv;$ $loc := \texttt{op}(\overline{args}');$	$(\overline{args}' \rightarrow \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$ $\setminus (\_ \rightarrow tmp) \setminus (loc \rightarrow \_)$ $\cup \{loc \rightarrow tmp\}$
$\texttt{BSload}(s)$: $tmp := [addr(\overline{args})];$ / $mv;$ $loc := [addr(\overline{args}')];$	$(\overline{args}' \rightarrow \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$ $\setminus (\_ \rightarrow tmp) \setminus (loc \rightarrow \_)$ $\cup \{loc \rightarrow tmp\}$
$\texttt{BSstore}(s)$: $[addr(\overline{args})] := tmp;$ / $mv;$ $[addr(\overline{args}')] := loc;$	$(loc \rightarrow tmp) \in transfer_{mv}(B) \ \wedge$ $(\overline{args}' \rightarrow \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$
$\texttt{BScall}(s)$: $tmp := f(sig, args)$ / $mv_1;$ $Ret(sig) := f();$ $mv_2;$	$(\overline{Params(sig)} \rightarrow \overline{args})$ $\subseteq \ transfer_{mv_1}(B)$	$transfer_{mv}(mv_2,$ $\{Ret(sig) \rightarrow tmp\} \cup$ $callee\text{-}save(transfer_{mv}(mv_1, B)))$
$\texttt{BScond}(s1, s2)$: $\texttt{cond}(cond, \overline{args})$; / $mv;$ $\texttt{cond}(cond, \overline{args}');$	$(\overline{args}' \rightarrow \overline{args}) \subseteq transfer_{mv}(B)$	$transfer_{mv}(mv, B)$
$\texttt{BSreturn}()$: $\texttt{ret}(arg);$ | $mv;$ $\texttt{ret};$	$(arg' \rightarrow arg) \in transfer_{mv}(B)$	$transfer_{mv}(mv, B)$

Figure 3.9 – Transfer functions for β_i

This is modelled in the transfer function by the function $callee\text{-}save$ which invalidates mappings to registers which are not callee-saved. For `BScond`, we simply check that the conditions evaluate to the same value thus ensuring that both program have the same control-flow. For `BSreturn`, we also check that the return values are the same and only update the mapping to model move instructions.

In order to get β_i, we iterate the data-flow equations until a fixpoint using data-flow framework of CompCert.

Example 4. *We apply our mapping algorithm to our example Figure 3.8. To simplify our example, we make the hypothesis that function calls do not have side-effects. Mapping of line n corresponds to its state when instructions of line n have been executed. Orange highlighting specifies the updates of the mapping.*

```
1 {text ← stack_text}
2 {text ← stack_text, key ← r2}
3 {text ← stack_text, salt ← r1, key  ← r2}
4 {text ← stack_text, salt ← r1, key  ← r2, key ← stack_key}
5 {text ← stack_text, salt ← r1, text ← r2, key ← stack_key}
6 {text ← stack_text, salt ← r1, tmp  ← r2, key ← stack_key}
7 {text ← stack_text, key ← r1, tmp  ← r2,  key ← stack_key}
8 {text ← stack_text, key ← r1, tmp  ← r2,  ??? ← stack_key}
```

We explain how we build our mapping for each line with our forward analysis:

1. *we assume that parameters from both functions have equal values*
2. `r2` *and* `key` *both store the result value of* `get_key()`
3. `r1` *and* `salt` *both store the result value of* `get_salt()`
4. `stack_key` *copies the value of* `r2` *which was mapped to* `key`
5. `r2` *copies the value of* `stack_text` *which was mapped to* `text`
6. `r2` *and* `tmp` *are the results of the same operation with equal operands*
7. `r1` *copies the value of* `stack_key` *which was mapped to* `key`
8. `r1` *and* `key` *are the results of the same operation with equal operands.* `stack_key` *was mapped to the initial value of* `key` *which has been erased during the last instruction. We are unable to find a mapping for* `stack_key`

We hit an observation point before executing the `return` *at line 9. We were unable to do a complete mapping for all variables of the LTL program at this observation point so the transformation is rejected. This is the result we expected since we previously pointed out that there was a leak through the variable* `stack_key` *which is detected by our analysis.*

3.3.4.4.b Verification of Sufficient Conditions

If the computation of β_i succeeds, the next step consists in verifying, for every observation point, the IFP verification conditions. For a given observation point $\bullet_i$, we verify that for every LTL location $l \in \gamma_i$ that may be modified by the current function, there exists either a mapping to an RTL register t *i.e.*, $(l \to t) \in \beta_i$ or the location l is constant $cst_i(l) = v$ for some $v \neq \top$.

Theorem 8. *Let p be an RTL program and p' be an LTL program. Suppose that we have successfully constructed β_i, γ_i and cst_i for every program point of p. If for every observation point $\bullet_i$, the following condition hold:*

$$\forall l \in \gamma_i.(\exists t.(l \to t) \in \beta_i) \vee (\exists v.cst_i(l) = v \wedge v \neq \top)$$

then the RA transformation of p to p' is IFP.

Moreover, if every pair (l_1, l_2) of distinct locations is mapped to distinct temporaries, the transformation prevents data duplication.

In the following section, we give some key insights on how the previous verification conditions are sufficient to ensure the existence of a simulation function α_i and a backward simulation between p and p'.

3.3.4.4.c Correctness of the Validation Algorithm

In this section, we prove the soundness of our validation algorithm for IFP. We suppose an execution of p and p' where memories m_1 and m'_1 are respectively leaked from observation point $\bullet_1$ and there exists α_1 such that $m_1 \sim_{\alpha_1} m'_1$. Similarly, we note $\bullet_2$ the observation point leaking m_2. It remains to prove that we can construct a backward simulation according to Definition 9. Hence we need to prove two goals: the existence of m_2 and α_2 such that $m_1 \rightsquigarrow m_2$ and $m_2 \sim_{\alpha_2} m'_2$. By definition of $\rightsquigarrow$, there is a derivation

of length n' such that $m'_1 \rightarrow^{n'} m'_2$. We have to show that there is a derivation of length n such that $m_1 \rightarrow^{n} m_2$. Such a proof is already needed by the original CompCert semantics preservation proof of the RA pass [92] and we therefore inherit it for free. The construction of α_2 partitions addresses in the following three categories:

i) addresses which are not LTL locations;

ii) LTL locations that may be modified by the current function;

iii) and LTL locations that have not been modified.

First, consider addresses that are not locations in the LTL memory model. The memory content at these addresses can only be modified via `load` and `store` instructions. While building β_2 we check that these memory accesses always agree on their computes arguments. Therefore, in p and p', the memory content remains the same and α_2 is the identity function for these addresses. Second, consider LTL locations that may have been modified since the start of the current function *i.e.* $l \in \gamma_2$. We know from our precondition of Theorem 8 that we are able to map l to either a temporary or a constant. Thus, α_2 is equal to β_2 or cst_2 for every location of $l \in \gamma_2$.

Third, consider for LTL locations that are necessarily unmodified *i.e.* $l \notin \gamma_2$. We know that when reaching $\bullet_1$ the memories m_1 and m'_1 were leaked and that $m_1 \sim_{\alpha_1} m'_1$. Moreover, locations and temporaries are local to their functions and observation points are placed before function calls and returns. From these facts we deduce that $\alpha_1(l)$ still holds at $\bullet_2$ and gives us $\forall (l \notin \gamma_2),\ \alpha_2(l) = \alpha_1(l)$.

Lastly, to ensure the absence of duplication of information, we need to prove that α_2 is injective. For addresses which are not locations, α_2 is the identity function which is injective. For addresses not in γ_2 this is given by the fact that by induction α_1 is also injective. For the other addresses, we have the hypothesis that every location maps to distinct temporaries which proves that α_2 is injective.

3.3.4.5 Patching algorithm

An interesting property of our IFP validator is that we can recover from certain validation failures; track down the origin of the information leak and apply a patch to a function f' to automatically close the information leak. When this is the case, it is possible to run existing optimisations unmodified, and during a post-treatment, detect and remove potential information leaks. Suppose that, for a given observation point $\bullet_i$, the verification conditions needed by Theorem 8 do not hold. Typically, there is a

location l that is neither a constant ($cst_i = \top$) nor has a mapping to some RTL temporary ($\beta_i(l) = \varnothing$). In this situation either the validator is not precise enough or l is responsible for an information leak. To close the potential leak detected by the IFP validator, our patching algorithm inserts an erasure instruction and sets the location to the constant 0 just before the observation point $\bullet_i$. After the addition of those erasure instructions, we have the guarantee that the transformed program is IFP. Moreover, those erasure instructions cannot compromise the correctness of the transformation. The rationale is that an erasure instruction for location l is necessarily a *dead* store. To see this, consider by contradiction that l is live. In that case, to establish semantics preservation, the original CompCert validator needs to establish a mapping for location l and an RTL temporary t. As our forward validator always computes more mappings than the original backward validator of CompCert, it would establish the same mapping. This contradicts our assumption.

As a result, we have the guarantee that inserting erasure instructions, according to the rules above, is a sound algorithm to make CompCert RA a semantic preserving IFP transformation.

Example 5. *We show the result of the patching on the example of Figure 3.8.*

```
1  def cipher2(stack_text):
2    r2    = get_key()
3    r1    = get_salt()
4    stack_key = r2
5    r2    = stack_text
6    r2    = r2^r1
7    r1    = stack_key
8    r1    = r1^r2
9  • return
```

```
1  def cipher3(stack_text):
2    r2    = get_key()
3    r1    = get_salt()
4    stack_key = r2
5    r2    = stack_text
6    r2    = r2^r1
7    r1    = stack_key
8    r1    = r1^r2
9    stack_key = 0    #patching
10 • return
```

Figure 3.10 – Before patching (left) - After patching (right)

Our validation algorithm detected the potential leak from `stack_key` *therefore, here we erase this location right before the observation point. Redoing the validation with* `cipher1` *and* `cipher3` *will map the constant* `0` *to the location* `stack_key` *at the observation point. This will give us a complete mapping of the locations of* `cipher3` *and the validation will succeed.*

3.3.5 Experiments

The translation validation algorithm of Section 3.3.4 has been integrated as part of the register allocation pass of CompCert [66]. We run our IFP validator; close any potential information leak using the patching algorithm of Section 3.3.4.5. Afterwards, we run the existing validator of CompCert, thus, ensuring the semantics correctness of our security transformation.

In our model, the attacker can only observe the memory at so-called observation points. As explained in Section 3.3.4, observation points are set at function boundaries: the attacker may observe memory just before call and return instructions. With this policy, our IFP validator enforces that register allocation does not introduce information leaks due, for instance, to stack allocated variables (or spilled registers) not being properly erased at function return; an acknowledged security issue [31, 116, 105].

3.3.6 Results and analysis

The experiments have been conducted on a quad-core Intel i7-6600U at 2.60GHz with 16GB of RAM running Fedora 27. We have tested our IFP validator on 24 programs which are all part of the CompCert test suite. For every program, our validator detected potential information leaks introduced by CompCert RA. All the information leaks have been successfully closed by our patching algorithm and all the resulting programs have passed the CompCert validator.

The impact of the security transformation on their efficiency is summarised in Figure 3.11. The first bar represents the runtime overhead of our secured RA compared to the original RA of CompCert. The benchmarks are sorted by increasing overhead, have a running time ranging from 1 to 10 seconds and the results are obtained by averaging 50 runs. The second (grey) bar represents the overhead in the number of executed instructions.

The overhead of our functions ranges from -5% to 15%. First for the speedups, we believe that they are due to lucky side-effect, probably due to an improved behaviour of the cache of instructions. Overall, we can see the expected trend that overheads are related to the number of additional instructions executed, except for cases like `nbody` or `nsieve`. Similarly, we suspect side-effects of the architecture, which this time may create additional overhead with very few patching instruction executed for `nsieve`. Fur-

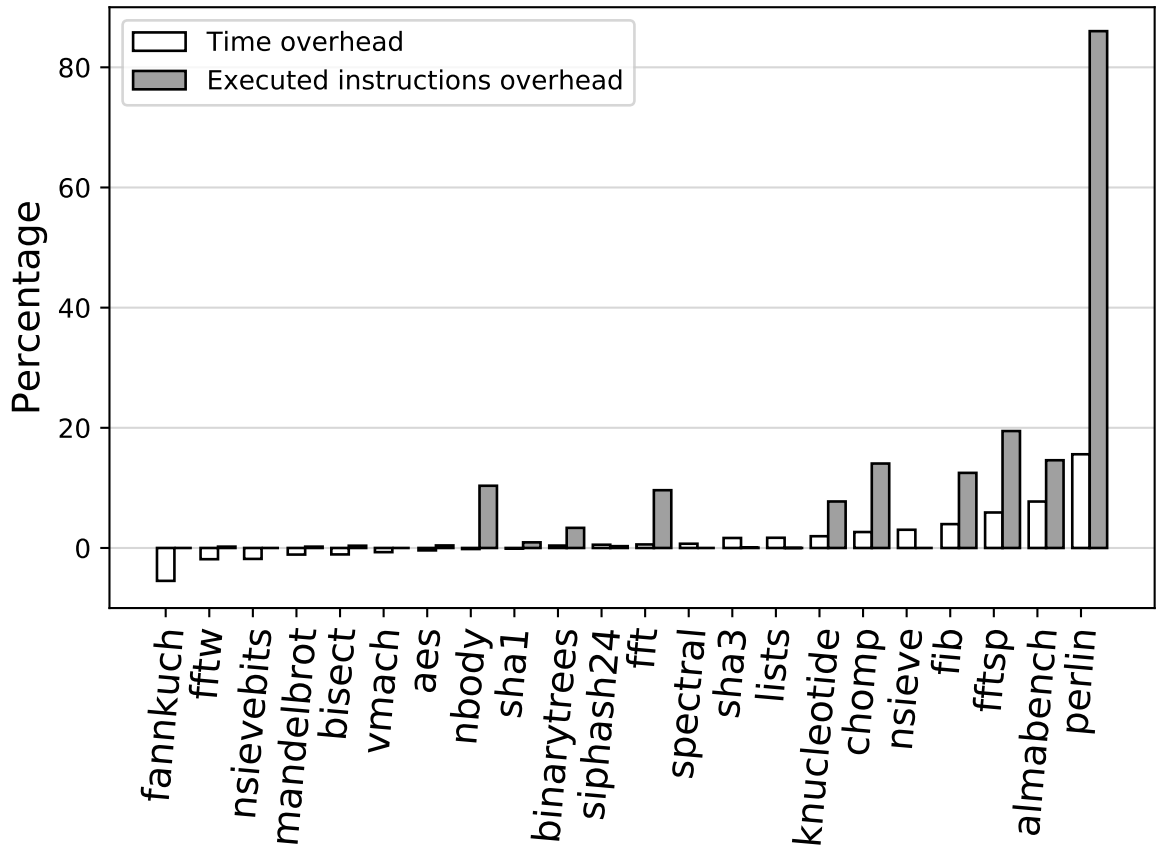


Figure 3.11 – Patched programs compared to original compared programs

ther investigations would be required to exactly pinpoint the origins of these anomalies and is left as future work. Anyway, these results are encouraging and show that an improved security can often be obtained without sacrificing too much efficiency. Finally, we see that for `perlin`, the number of additional instructions executed reach 85% which almost doubles the execution number of the original program. Fortunately, the impact on the overhead is not as high (15%), but the large number of patching instructions executed can be explained by the conservative analysis of our validation. Indeed, our IFP validator needs to be quite conservative about the behaviour of function calls relying exclusively on guarantees given by calling conventions. This is illustrated by Example 6 where a potentially spurious instruction needs to be inserted to protect against a potential information leak.

Example 6. *Consider the code snippet of Figure 3.12.*

```
r := g(x);
x := 2;
return x;
```

```
eax := g(edi);
eax := 2;
return;
```

Figure 3.12 – RTL program (left), LTL program (right)

According to standard x86 calling conventions, it is typically compiled as the LTL program of Figure 3.12. After the call of `g` *, we only have the guarantee that registers are either overwritten or carry the same value as before. Therefore, our validator makes the conservative assumption that* `edi` *(that is not callee-saved) was not modified by* `g` *, may leak the initial value of the variable* `x` *and therefore needs to be explicitly erased, thus adding a potential spurious erasure instruction* `edi := 0` *.*

3.4 Discussion and Perspectives

In this section, we first show how useful is our IFP property to preserve safe erasure, masking and non-interference mentioned in Section 3.1.4. Then we will discuss how to extend our current work to other compilation pass and finally we show the different directions that can improve our current work.

3.4.1 Which properties can I preserve?

We discuss informally if IFP preserve the different properties cited in our motivations. We also present some conditions which are necessary to preserve such properties and the different limitations.

3.4.1.1 Safe Erasure

Safe erasure is used to reduce the lifetime of secrets in the memory by using erasure instructions. Our IFP property ensures that information that were removed from the memory in the source program should not be present in the memory of the transformed program. Therefore, we can easily ensure that IFP preserves safe erasure. The only limitation resides in the fact that IFP only guarantees that the secrets will be erased when reaching an observation point. If the policy used is too lax and possessed few observation points, the lifetime of secrets can be extended if the erasure instruction is moved within observation points like in Figure 3.13.

```
x := f();
x := 0;
y := g();
return; •
```

```
x := f();
y := g();
x := 0;
return; •
```

Figure 3.13 – Safe erasure broken by code motion

The program on the left can be transformed to the one on the right by the optimisation named *Code Motion*. Initially the value `x` is supposed to be erased before calling the function `g()` to avoid possible leaks. However, due to code motion the erasure instruction has been delayed and is now called after `g()` which defeats its purpose. Unfortunately, this transformation is IFP since that `x` is effectively erased when reaching the observation point. The main issue here is the misplacement of the observation points and could be solved if an observation point was located right after the erasure instruction. The relation between IFP and code motion is discussed further in Section 3.4.2.2.

3.4.1.2 Masking

Masking split a secret into multiple shares random shares and computations are done on each share independently before reconstructing the final value. This increases

the difficult of guessing a secret through the observation of different values and add noise to the computations since the shares are changed for every execution. Among the properties of the threshold implementation we presented in Section 3.1.4.2, non-completeness is the properties that guarantee security against power analysis attacks. Non-completeness states that none of the intermediate values used during the computation is dependent of all the initial shares. Therefore, if n shares was used at the beginning, intermediate values are related to at most $n-1$ of these shares. Hence an attacker is not able reconstruct the secret from intermediate values if he does not get at least n intermediate values. This is similar to attacker knowledge where the attacker tries to deduce the inputs from a partial observation. Since our IFP property makes sure that any observation on the transformed program can be simulated by an observation of the source program, then the transformed program cannot have an intermediate value related to all the n shares. We are confident that the security of masking is preserved by our IFP property but additional work would be required to bridge the gap to the probabilistic model used by most work on masking.

3.4.1.3 Non-Interference

We are convinced that IFP transformations preserve the following definition of non-interference inspired from Barthe *et al.* [11]:

Definition 12 (IFP Non-Interference)**.**

$$\forall(c,c').\left(\begin{array}{c}(p,c)\Downarrow t\\ \wedge\\ (p,c')\Downarrow t'\end{array}\right)\wedge c=_L c' \quad\Rightarrow\quad \mathcal{K}^t(p)=\mathcal{K}^{t'}(p)$$

with $c =_L c'$ meaning that c and c' are equal on the addresses of the set L

This is ongoing work and we explain our intuition on this matter but proofs would be required to confirm our statement. For simplicity, we assume that the addresses are split into two independent sets L and H corresponding to low security and high security addresses respectively.

First, we remark that we only use the notion of full attacker knowledge in this definition rather than quantifying over all attackers. This is due to the fact that we are working on non-interference which implies that the information learned from executions of c and

c' should be the same, and this for any attacker. Therefore, instead of quantifying on partial attackers we can limit our definition to the full attacker which is able to gather all the information contained in a trace.

Proof outline We know that p is non-interfering for inputs c and c' which agree on low security addresses. Hence, the attacker knowledge derived from observing their executions are equal and we name it K. Since the executions are indistinguishable for the attacker and that c and c' only differ on high security addresses, then we know that K does not contain any information on the initial values used at high security addresses of the input memories.

We call K_t and K'_t, the knowledge obtained from observing the executions of the transformed program p_t with inputs c and c' respectively. Since the transformation from p to p_t is IFP, we know that K_t and K'_t can only contain less or equal information than K on the input memory used for an execution. If the information is equal, then we have $K_t = K'_t = K$ which means that p_t is also non-interfering for c and c'. We show that even if K_t and K'_t are different from K we still have $K_t = K'_t$. Since K has no information on high-security addresses and that K_t contains less information than K, then only information from low-security addresses has been removed in K_t. Let's call a a low-security address which initial value was removed during the execution of p_t but preserved with p which gave the knowledge K_t from K. Therefore, the initial value at address a was replaced in p_t by a new value which is either:

— a program constant, then that means K'_t also lost the information from a by the same constant meaning that $K_t = K'_t$

— other input values:

  — the new value of a only involves low-security inputs. Then K'_t is affected in the same manner as K_t since the inputs c and c' agree on the low-security addresses. Hence we still have $K_t = K'_t$

  — the new value of a involves high-security addresses. This is not possible because K contains no information about high-security inputs. Since the transformation is IFP, this should not be the case with K_t and K'_t either

Therefore, from their source knowledge K, the information removed to obtain K_t and K'_t can only be identical which gives $K_t = K'_t$ meaning that p_t is also non-interfering for inputs c and c'.

3.4.2 Securing others compiler passes

In this section, we do a quick overview of how our IFP property can be applied to other compilation passes. In their paper, D'Silva *et al.* [31] mention that compiler passes such as *Code Motion* an *Inlining*, because they modify the lifetime of variables, may introduce information leaks. In the following, we explain how these transformations fit with our IFP property.

3.4.2.1 Inlining

In our experiments, observation points are attached to function calls. As inlining consists in replacing a function call by a function body, if has the effect of removing an observation point. As observations need to be synchronised, inlining breaks this property and therefore is simply not an IFP transformation. As this may seem overly restrictive, a first solution to accommodate some inlining would be to weaken the security policy and attach observation points only to security critical functions. As a result, those critical functions would not be inlined but other functions could be freely inlined without modifying observation points. In this restricted scenario, inlining would be an IFP transformation. Another tempting approach would be to detach the observation point from the function call. Yet, this raises the issue of code motion that we discuss below.

3.4.2.2 Code motion

The risk of code motion is that code, and therefore information, could move around observation points and thus modify the security policy. Consider for instance, the code of Figures 3.14 and 3.13 where the code is subject to code motion. In Figure 3.14

```
•1 x := f();          •2 y := g();
•2 y := g();          •1 x := f();
```

Figure 3.14 – Code motion

the function calls order has been reversed. Since our IFP property works use a lock-step relation between traces, this transformation is inherently not IFP. It follows that a IFP-aware compiler needs to limit code motion within the bounds of observation points. However, code motion can still mess up the security policy such as in Figure 3.13. Here

the erasure instruction has been delayed and which leaves the secret value x vulnerable to attacks during the execution of the function g. The result is that the transformation is formally IFP but defeats the intention of the security policy.

3.4.3 Observation points

Limitations of our observation points are brought to light by these two compiler passes: inlining and code motion. Observations points are not robust to this kind of transformations where instructions can be moved around. Therefore, we identify the pros and cons of using observation points to be able to improve our future work.

Capturing lifetime Capturing lifetime is a core point for preservation properties since having extended lifetime for sensitive variables increase the opportunities to carry out an attack. Observation points can conveniently capture lifetime of variables in presence of erasure instructions. Indeed, if a variable is alive until an erasure instruction, our IFP property can guarantee that this remains true after transformation by placing an observation point after the erasure. Granularity of lifetimes depends on how often observation points are hit during the execution. In the case where a single observation point is hit during an execution, we can only say if a variable is alive at this point. A solution would be to place observation points at every instruction, which in turn would severely limit the number of allowed transformations. For example, DSE which removes some instructions would automatically be not IFP since it would remove some observation points in the program. Ideally, we would like a measure of variable lifetime which is more fine-grained than observation points and flexible enough to allow common compiler transformations.

Order of instructions Another usage of observation points is that, when combined with our lockstep relation between traces, they preserve the instructions order during a transformation. A use-case is when an erasure instruction is placed right before a vulnerable instruction, then the order must be preserved during transformations. On the other hand, if the ordering is too strict we can fall back to the example of Figure 3.14, where code motion is restricted since it cannot change the order of observation points. It is important to keep instructions order in situations where some specific part of the code is vulnerable and likely to be targeted by attackers. In these cases, the secu-

rity protections definitely needs to be placed before the vulnerable code. However, in our context, attackers can freely probe the memory of a program during its execution. Therefore, all the code executed is equally vulnerable which means that preserving instructions order is not be a necessity in our case. We believe that as long as the lifetime length of the secrets is properly preserved during transformation it should not matter when the critical computations are made during the execution.

Relevant attacker model Lastly, as we said in the motivations, we focus on attackers that can probe the memories of a program execution. Observation points are too strict since they are statically placed in program points. Additional improvements can be made by giving attackers more freedom in their observations.

Using these analysis, we can definitely improve our current work. One possibility is to use attackers which can observe the memory at any time but during k instructions. Similarly, we should be able to simulate any such attacker of the transformed program in the source program. Having this parametrized k for the length of an observation allow us a flexible measure for lifetime of variables. Furthermore, the attacker would be able to probe the memory at any time which is more realistic when working with side-channel attacks. This idea requires to be further developed and is left as future work.

CONCLUSION

Our society has been growingly dependent on computer systems and this tendency will not slow down in the incoming years. Nowadays, critical services such as health or economy are based on computer systems and cannot function properly without them. Similarly, interests over cybersecurity have been increasing alongside the possible consequences brought by successful attacks on these systems.

This thesis tackles the issue of security of systems and especially focuses on compilation to achieve its goal. We explore the two possible behaviours of a secure compiler which are enforcement and preservation. Compilers which enforce security properties make sure that regardless of the source program, transformed programs will be in compliance with the compiler security policy. In the other hand, security preserving compilers goal is to guarantee that the transformed program is at least as secure as the provided source program. Choosing between enforcement and preservation depends on the amount of trust given to the source program provider. In a situation where the source program is untrusted (may contain malicious code or vulnerabilities), enforcement guarantees that the compiled program will conform to the security policy of the compiler. However, in the situation where the source program provider is trusted, enforcement of security by the compilers may be too rigid. Indeed, a compiler may fail to meet the developer expectations if the security policy enforced during compilation differs from the developer needs. For example, take a developer which only wants to prevent secrets from leaking out of its program and a compiler forbidding any flow outside of the program. Here the restrictions imposed by the compiler may be too strong for the developer and might transform the program in a non-desirable way. A program such as a password checker would not be able to function properly with such restrictions imposed by the compiler. Another possible consequence can be unnecessary overhead introduced by the secure compiler.

Preservation of security properties is more suited to these situations where the source program is trusted. The core principle of preservation is to make sure that the compiled program should be as secure as the source program. The notion of security can change depending on the case but the motivation stays the same. Most of the

time rather than strict preservation, which requires equal security for source and transformed programs, secure compilers settle for a looser notion of preservation which requires transformed program to be at least as secure as the source. Strict preservation has the disadvantage to leave very few leeway for compilers to do their transformations. Therefore, looser preservation is sometimes a better compromise between speed and security. One may argue that a compiler enforcing a really strong security policy may be considered as a compiler preserving security since the compiled program will almost always be more secure than the source. While this is true, the main goal of the two compilers are different since a preserving compiler aim to translate as best as possible the developer security intentions. Whereas an enforcing compiler will implement a fixed security policy on transformed program regardless of the source.

This thesis explored both kind of secure compilers. First, CompCertSfi was implemented which is a compiler enforcing the security policy of software fault isolation on compiled programs. Then we focused on preserving memory probing countermeasures during compilation by defining a notion of information flow preserving transformation.

Software Fault Isolation

In Chapter 2 we presented SFI, an isolation technique which focuses on the speed of the interactions between the different entities. In SFI, a host program will setup the different modules in its own virtual memory. To prevent those modules from accessing the host data, they will be confined in specific memory areas called sandboxes. The usual implementations of SFI are composed of a SFI generator (included in the compiler) and a verifier. The generator enforces SFI transformations on the programs and the verifier makes sure that the executables are compliant with the SFI policy before loading them. SFI is an example of a compiler enforcing security on programs. Even if the source program is already secure, SFI transformations will still be done on programs to comply with SFI security expectations. We developed CompCertSfi an implementation of SFI from the certified C compiler CompCert. The originality of our approach is that we take advantage of CompCert semantics preservation theorem to get rid of the verifier. Inspired by the work Kroll *et al.* [60], SFI transformations are done on a high-level language called Cminor. Our SFI transformations have been proven to be SFI-secure and also to produce programs devoid of undefined behaviours. The latter point is crucial since CompCert safety preservation corollary only applies

on programs with no going-wrong behaviour. Security of programs is then preserved during compilation using this preservation property. The first advantage of such approach is portability, since the transformations are done at high-level, CompCertSFI automatically supports the CompCert architectures. A second advantage is that SFI transformations can benefit from compiler optimisations with this approach. Analysis of programs are more complex to perform at assembly level. Therefore, to reduce the impact on speed, traditional SFI verifiers only check syntactic properties of the programs. This limits the possible optimisations that can be carried out on traditional SFI transformations since the verifiers have limited capability. However, in our approach, the verifiers is removed from the SFI toolchain which allows us to do more optimisations as long as they are proven correct. In our benchmarks, we show that CompCertSFI have comparable execution speed to Native Client [117, 98, 6].

Information-Flow Preserving Transformation

Chapter 3 is centered around preservation of security countermeasures against a memory probing attacker. Our attacker model is a passive attacker which is able to peek into a process memory during its execution. We define the security of a program using the notion of *Attacker Knowledge* from Askarov *et al.* [7]. Attacker knowledge corresponds to the set of possible initial memories that match the observations made by the attacker during an execution. We also use a notion of partial attackers which are able to observe a parametrized number of bits of an execution trace. These partial attackers allow us to have a measurement of the security of a program. For example, a secret value may leak with a partial attacker observing the whole trace but not with an attacker with only n bits of observation. We formalize the notion of IFP transformation using the preservation principle of secure compilers where the security of a program is defined by attacker knowledge and partial attackers. More precisely:

For any partial attacker, an attacker knowledge derived from a transformed execution can always be matched by a more precise attacker knowledge derived from a source execution

We also provide a proof technique to prove that a transformation is IFP. The proof technique consists in mapping every memory location of a transformed program to either a memory location of the source program or a constant. If one is able to construct

such mapping then the transformation is IFP. We use our IFP property to propose two secure compiler passes: dead store elimination (DSE) and register allocation (RA). An IFP algorithm of DSE can be proven IFP by slightly modifying the liveness analysis. For register allocation, we modify CompCert RA validator to also check that the transformation is IFP. The validator succeeds when it manages to construct the mapping used by our proof technique. In case of failure, a patching algorithm can be used to further modify the program to get an IFP transformation. We benchmarked our patching algorithm to measure the cost of our security. The overhead peaks at 15% but most programs are under 5% which is rather encouraging. These results can be improved in the future using additional global analysis to avoid doing conservative choices. Finally, we show that our IFP property is suitable to preserve security properties such as safe erasure, masking and non-interference. Future work should be aimed at improving our property in order to have more flexible attacker observations.

Discussion and perspectives

CompCertSFI

We fulfilled our first objective with CompCertSFI to show that it is possible to do SFI without a verifier in the toolchain, as envisioned by Kroll *et al.* [60]. In this section, we discuss our advancement in regard to our second goal, which is, to have a competitive SFI implementation for real-world applications. To define the roadmap to reach our objective, we evaluate our current work with other SFI implementations under different criteria: security conditions, security policy, trusted computing base (TCB), portability and speed. Native Client [117, 98, 6] is our main competitor since this is the only work which have been deployed to the industry.

Security conditions Having a verifier is a headache for optimisations but can also be a blessing. Indeed, with a verifier we can just check untrusted binaries before loading them. With CompCertSFI we require to have the source code of the untrusted program to ensure a secure execution. Therefore, our work have stronger requirements when it comes to security conditions. This is inherent to our design which cannot be changed and we need to make up for this weakness with other factors.

Security policy Since we are comparing different SFI implementations, the different policies are roughly the same. The biggest difference may lie in how function pointers are handled. It is difficult to check the control flow of assembly programs due to the usage of indirect addressing. Native Client chose to split their code into chunks of fixed size and use masking alignment on indirect addressing. This ensures that any jump is directed to the beginning of a chunk. Our implementation has better control since we dynamically check that function pointers calls always lead to an existing function and that their signatures match.

Portability This is the selling point of the PSFI approach since we can support all CompCert architectures with a single implementation of the SFI transformations. On the other hand, traditional SFI needs to implement a generator and a verifier for each architecture they want to target.

TCB and speed We regroup these two categories since SFI overhead is related to the complexity of the verifier. In SFI, the TCB is composed of the trusted library and the verifier. CompCertSfi also possesses a trusted library, and the rest of the TCB is equivalent to the trust we have in CompCert which is proven in Coq. Therefore, comparison of TCB is decided between CompCert TCB and the trust we put into the verifier. Verifiers can use syntactic checks which are simple and can be easily checked or more advanced static analysis. Indeed, optimisations of SFI transformations are limited by the capacity of their verifier to check that a binary is SFI-secure. A simple verifier is not able to check optimisations of SFI constructs but is more easily trusted and can also be certified [77]. In our benchmarks, we show that CompCertSfi performance are comparable to Native Client.

Among all the criteria compared, speed has significant importance and is also the one which can be improved the most. There are two solutions to speed up CompCertSfi. First, we argue that by using PSFI approach, our SFI constructs benefit from compiler optimisations which should improve the performance. Therefore, a direction of improvement is to design optimisations specifically crafted to speed up SFI constructs and limit the overhead brought by SFI transformations.
Secondly, we showed that another bottleneck resided in the usage of the compiler CompCert which is slower than Clang or Gcc. Hence, we can also focus on improving the overall performance of CompCert to reach our objective. An idea would be to

integrate CompCertSSA [10] which uses *Single Static Assignment* optimisations to speed up programs.

Information-Flow Preserving transformation

Preservation of security properties during a compilation is a relatively recent field of research which stems from semantics preservation of compilers. To our knowledge, we are the first to preserve security of programs against a memory probing attacker. This attacker model is suited for side-channel attacks such as power or electromagnetic analysis, where an attacker can get any intermediate value using a statistical attack with enough observations. Our definitions can be used on the issue of preserving countermeasures against differential power analysis, such as software masking, which has not been solved yet.

This part of the thesis already has practical implementations but still has room for growth. Multiple prospects can be explored in the future.

Preservation of Masking As mentioned previously, we are convinced that our IFP property is suited to preserve masking implementations. Threshold implementations (Section 3.1.4.2) are defined over probabilistic programs so our first would be to modify our work to fit in this execution model. Afterwards, it would be interesting to implement our preservation property in a compiler such as Jasmin [3] to produce DPA-resistant cryptographic implementations.

Framework for Passive Attackers The most powerful passive attacker that we can think of is able to see the whole memory of a program during the entire execution. An attractive prospect would be to have a framework to generate any kind of passive attacker by parametrizing the different aspects of this most powerful attacker. In our work, we have already parametrized when and how much an attacker is able to observe. Currently, an attacker is only able to observe when the execution reaches an observation point and is also limited by how many bits it is able to observe. A suggestion would be to also limit our attackers on where in the memory they are be able to observe. We would obtain a generic framework to easily formalize any passive attacker we want to prevail against. For example, to preserve constant-time programs we would have an attacker which is able to see an unlimited number of bits but only at the addresses used when

branching or accessing the memory. We could also preserve certain security properties that label memory areas with different security levels where the attackers can only access areas with low credentials.

BIBLIOGRAPHY

[1] Martín Abadi et al., « Control-flow Integrity Principles, Implementations, and Applications », *in*: *ACM Trans. Inf. Syst. Secur.* 13.*1* (2009), 4:1–4:40, ISSN: 1094-9224, DOI: 10.1145/1609956.1609960, URL: http://doi.acm.org/10.1145/1609956.1609960.

[2] *ACCESS_ONCE() and compiler bugs.* https://lwn.net/Articles/624126/, 2014.

[3] José Bacelar Almeida et al., « Jasmin: High-Assurance and High-Speed Cryptography », *in*: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 1807–1823, DOI: 10.1145/3133956.3134078, URL: https://doi.org/10.1145/3133956.3134078.

[4] José Bacelar Almeida et al., « The Last Mile: High-Assurance and High-Speed Cryptographic Implementations », *in*: *CoRR* abs/1904.04606 (2019), arXiv: 1904.04606, URL: http://arxiv.org/abs/1904.04606.

[5] José Bacelar Almeida et al., « Verifying Constant-Time Implementations », *in*: *USENIX Security 16*, USENIX, 2016, pp. 53–70.

[6] Jason Ansel et al., « Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code », *in*: *SIGPLAN Not.* 46.*6* (2011), pp. 355–366, ISSN: 0362-1340, DOI: 10.1145/1993316.1993540, URL: http://doi.acm.org/10.1145/1993316.1993540.

[7] Aslan Askarov and Stephen Chong, « Learning is Change in Knowledge: Knowledge-Based Security for Dynamic Policies », *in*: *CSF 2012*, IEEE Computer Society, 2012, pp. 308–322.

[8] Aslan Askarov and Andrei Sabelfeld, « Gradual Release: Unifying Declassification, Encryption and Key Release Policies », *in*: *SP'07*, IEEE, 2007, pp. 207–221, ISBN: 0-7695-2848-1, DOI: 10.1109/SP.2007.22, URL: https://doi.org/10.1109/SP.2007.22.

[9] Michael Backes and Boris Köpf, « Formally Bounding the Side-Channel Leakage in Unknown-Message Attacks », *in*: Oct. 2008, pp. 517–532, DOI: 10.1007/978-3-540-88313-5_33.

[10] Gilles Barthe, Delphine Demange, and David Pichardie, « Formal Verification of an SSA-Based Middle-End for CompCert », *in*: *ACM Trans. Program. Lang. Syst.* 36.*1* (2014), 4:1–4:35, DOI: 10.1145/2579080, URL: https://doi.org/10.1145/2579080.

[11] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte, « Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time" », *in*: *CSF 2018*, IEEE, 2018, pp. 328–343.

[12] Gilles Barthe et al., « EasyCrypt: A Tutorial », *in*: *Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures*, ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli, Cham: Springer International Publishing, 2014, pp. 146–166, ISBN: 978-3-319-10082-1, DOI: 10.1007/978-3-319-10082-1_6, URL: https://doi.org/10.1007/978-3-319-10082-1_6.

[13] David A. Basin et al., « Formal Analysis of 5G Authentication », *in*: *CoRR* abs/1806.10360 (2018), arXiv: 1806.10360, URL: http://arxiv.org/abs/1806.10360.

[14] Daniel J. Bernstein, « Cache-timing attacks on AES », *in*: 2005.

[15] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Jan. 2004, ISBN: 3540208542, DOI: 10.1007/978-3-662-07964-5.

[16] Frédéric Besson, Sandrine Blazy, and Pierre Wilke, « A Precise and Abstract Memory Model for C Using Symbolic Values », *in*: *APLAS*, vol. 8858, LNCS, Springer, 2014, pp. 449–468.

[17] Frédéric Besson, Sandrine Blazy, and Pierre Wilke, « A Verified CompCert Front-End for a Memory Model Supporting Pointer Arithmetic and Uninitialised Data », *in*: *Journal of Automated Reasoning* (2018), Accepted for publication, ISSN: 1573-0670.

[18] Frédéric Besson, Sandrine Blazy, and Pierre Wilke, « CompCertS: A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics », *in*: *ITP*, vol. 10499, LNCS, Springer, 2017, pp. 81–97.

[21] Frédéric Besson, Thomas Jensen, and Julien Lepiller, « Modular Software Fault Isolation as Abstract Interpretation », *in*: *Static Analysis*, ed. by Andreas Podelski, Cham: Springer International Publishing, 2018, pp. 166–186, ISBN: 978-3-319-99725-4.

[23] James Bowman, « Format String Attacks 101 », *in*: (Jan. 2000).

[24] Miguel Castro et al., « Fast Byte-granularity Software Fault Isolation », *in*: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, Big Sky, Montana, USA: ACM, 2009, pp. 45–58, ISBN: 978-1-60558-752-3, DOI: 10.1145/1629575.1629581, URL: http://doi.acm.org/10.1145/1629575.1629581.

[25] Suresh Chari et al., « Towards sound approaches to counteract power-analysis attacks », *in*: Springer-Verlag, 1999, pp. 398–412.

[26] Stephen Chong and Andrew C. Myers, « End-to-End Enforcement of Erasure and Declassification », *in*: *CSF'08*, IEEE, 2008, pp. 98–111, ISBN: 978-0-7695-3182-3, DOI: 10.1109/CSF.2008.12, URL: https://doi.org/10.1109/CSF.2008.12.

[27] Stephen Chong and Andrew C. Myers, « Language-Based Information Erasure », *in*: *CSFW'05*, IEEE, 2005, pp. 241–254, ISBN: 0-7695-2340-4, DOI: 10.1109/CSFW.2005.19, URL: https://doi.org/10.1109/CSFW.2005.19.

[28] Véronique Cortier, Alicia Filipiak, and Joseph Lallemand, « BeleniosVS: Secrecy and Verifiability against a Corrupted Voting Device », *in*: *32nd IEEE Computer Security Foundations Symposium (CSF'19)*, To appear, Hoboken, June 2019.

[29] Crispin Cowan et al., « StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks », *in*: *In Proceedings of the 7th USENIX Security Symposium*, 1998, pp. 63–78.

[30] *CWE-14: Compiler removal of code to clear buffers*, http://cwe.mitre.org/data/definitions/14.html, 2013.

[31] Vijay D'Silva, Mathias Payer, and Dawn Song, « The Correctness-Security Gap in Compiler Optimization », *in*: *SPW '15*, IEEE, 2015, pp. 73–87, ISBN: 978-1-4799-9933-0, DOI: 10.1109/SPW.2015.33, URL: http://dx.doi.org/10.1109/SPW.2015.33.

[32] Joe Damato, *Beware of Compiler Optimizations*, 2006, URL: https://wiki.sei.cmu.edu/confluence/display/c/MSC06-C.+Beware+of+compiler+optimizations.

[33] Chaoqiang Deng and Kedar S. Namjoshi, « Securing a Compiler Transformation », *in*: *SAS*, vol. 9837, LNCS, Springer, 2016, pp. 170–188.

[34] Chaoqiang Deng and Kedar S. Namjoshi, « Securing the SSA Transform », *in*: *SAS*, vol. 10422, LNCS, Springer, 2017, pp. 88–105.

[35] Siemen Dhooghe, Svetla Nikova, and Vincent Rijmen, *Threshold Implementations in the Robust Probing Model*, Cryptology ePrint Archive, Report 2019/1005, https://eprint.iacr.org/2019/1005, 2019.

[36] Zakir Durumeric et al., « The Matter of Heartbleed », *in*: Nov. 2014, pp. 475–488, DOI: 10.1145/2663716.2663755.

[37] Chucky Ellison and Grigore Roşu, « An executable formal semantics of C with applications », *in*: *POPL*, ACM, 2012.

[38] Úlfar Erlingsson et al., « XFI: Software Guards for System Address Spaces », *in*: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, Seattle, Washington: USENIX Association, 2006, pp. 75–88, ISBN: 1-931971-47-1, URL: http://dl.acm.org/citation.cfm?id=1298455.1298463.

[39] John Fisher-ogden, *Hardware support for efficient virtualization*, 2006.

[40] Bryan Ford and Russ Cox, « Vx32: Lightweight User-level Sandboxing on the x86 », *in*: *USENIX 2008 Annual Technical Conference*, ATC'08, Boston, Massachusetts: USENIX Association, 2008, pp. 293–306, URL: http://dl.acm.org/citation.cfm?id=1404014.1404039.

[41] J. A. Goguen and J. Meseguer, « Security Policies and Security Models », *in*: *1982 IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–11, DOI: 10.1109/SP.1982.10014.

[42] Louis Goubin and Jacques Patarin, « DES and Differential Power Analysis (The "Duplication" Method) », *in*: *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, London, UK, UK: Springer-Verlag, 1999, pp. 158–172, ISBN: 3-540-66646-X, URL: http://dl.acm.org/citation.cfm?id=648252.752372.

[43] M. Gruhn and T. Müller, « On the Practicability of Cold Boot Attacks », *in*: *2013 International Conference on Availability, Reliability and Security*, Sept. 2013, pp. 390–397, DOI: 10.1109/ARES.2013.52.

[44] M. R. Guthaus et al., « MiBench: A free, commercially representative embedded benchmark suite », *in*: *2001 IEEE International Workshop on Workload Characterization, WWC 2001*, United States: Institute of Electrical and Electronics Engineers Inc., 2001, pp. 3–14.

[45] J. Alex Halderman et al., « Lest We Remember: Cold Boot Attacks on Encryption Keys », *in*: *17th USENIX Security Symposium (USENIX Security 08)*, San Jose, CA: USENIX Association, July 2008, URL: https://www.usenix.org/conference/17th-usenix-security-symposium/lest-we-remember-cold-boot-attacks-encryption-keys.

[46] Chris Hathhorn, Chucky Ellison, and Grigore Roşu, « Defining the Undefinedness of C », *in*: *PLDI*, ACM, June 2015, pp. 336–345.

[47] *How to zero a buffer - Erratum*, https://www.daemonology.net/blog/2014-09-05-erratum.html, 2014.

[48] Sebastian Hunt and David Sands, « Just Forget It: The Semantics and Enforcement of Information Erasure », *in*: *ESOP'08*, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 239–253, ISBN: 3-540-78738-0, 978-3-540-78738-9, URL: http://dl.acm.org/citation.cfm?id=1792878.1792903.

[49] Yuval Ishai, Amit Sahai, and David Wagner, « Private Circuits: Securing Hardware against Probing Attacks », *in*: *Advances in Cryptology - CRYPTO 2003*, ed. by Dan Boneh, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 463–481, ISBN: 978-3-540-45146-4.

[50] ISO, *ISO C Standard 1999*, tech. rep., 1999.

[51] J.Daemen and V.Rijmen, « Resistance Against Implementation Attacks: A Comparative Study of the AES Proposals », *in*: *AES'99* (Mar. 1999).

[52] Jeremy Jacob, « On the Derivation of Secure Components », *in*: IEEE, 1989, pp. 242–247.

[53] Drew Dean Joshua A. Kroll, *BakerSFIeld: Bringing software fault isolation to x64*, Nov. 2009.

[54] Jeehoon Kang et al., « Lightweight verification of separate compilation », *in*: *POPL*, ACM, 2016, pp. 178–190.

[55] Paul C. Kocher, « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems », *in*: *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, London, UK, UK: Springer-Verlag, 1996, pp. 104–113, ISBN: 3-540-61512-1, URL: http://dl.acm.org/citation.cfm?id=646761.706156.

[56] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun, « Differential Power Analysis », *in*: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, Berlin, Heidelberg: Springer-Verlag, 1999, pp. 388–397, ISBN: 3-540-66347-9, URL: http://dl.acm.org/citation.cfm?id=646764.703989.

[57] Paul Kocher et al., « Spectre Attacks: Exploiting Speculative Execution », *in*: *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[58] Boris Köpf and Markus Dürmuth, « A Provably Secure And Efficient Countermeasure Against Timing Attacks », *in*: *IACR Cryptology ePrint Archive* 2009 (July 2009), p. 89, DOI: 10.1109/CSF.2009.21.

[59] Robbert Krebbers, « An operational and axiomatic semantics for non-determinism and sequence points in C », *in*: *POPL*, ACM, 2014.

[60] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel, « Portable Software Fault Isolation », *in*: *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, IEEE Computer Society, 2014, pp. 18–32, ISBN: 978-1-4799-4290-9, DOI: 10.1109/CSF.2014.10, URL: http://dx.doi.org/10.1109/CSF.2014.10.

[61] Ramana Kumar et al., « CakeML: A Verified Implementation of ML », *in*: *Principles of Programming Languages (POPL)*, ACM Press, Jan. 2014, pp. 179–191, DOI: 10.1145/2535838.2535841, URL: https://cakeml.org/popl14.pdf.

[62] Chris Lattner, *What Every C Programmer Should Know About Undefined Behaviour*, May 2011, URL: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[63] K Rustan M Leino, « Dafny: An Automatic Program Verifier for Functional Correctness », *in*: Dec. 2010, pp. 348–370, DOI: 10.1007/978-3-642-17511-4_20.

[64] Sorin Lerner et al., « Automated Soundness Proofs for Dataflow Analyses and Transformations Via Local Rules », *in*: *In Proc. of the 32nd Symposium on Principles of Programming Languages*, ACM Press, 2005, pp. 364–377.

[65] Xavier Leroy, « A formally verified compiler back-end », *in*: *Journal of Automated Reasoning* 43.*4* (2009), pp. 363–446.

[66] Xavier Leroy, « Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant », *in*: *POPL'06*, ACM, 2006, pp. 42–54, ISBN: 1-59593-027-2, DOI: 10.1145/1111037.1111042, URL: http://doi.acm.org/10.1145/1111037.1111042.

[67] Xavier Leroy, « Formal Verification of a Realistic Compiler », *in*: *Commun. ACM* 52.*7* (2009), pp. 107–115, ISSN: 0001-0782, DOI: 10.1145/1538788.1538814, URL: http://doi.acm.org/10.1145/1538788.1538814.

[68] Xavier Leroy and Sandrine Blazy, « Formal verification of a C-like memory model and its uses for verifying program transformations », *in*: *Journal of Automated Reasoning* 41.*1* (2008).

[69] Xavier Leroy et al., « The CompCert memory model », *in*: *Program Logics for Certified Compilers*, Cambridge University Press, 2014, ISBN: 9781107048010.

[70] Kyung-suk Lhee and Steve Chapin, « Buffer Overflow and Format String Overflow Vulnerabilities », *in*: *Electrical Engineering and Computer Science* 33 (Apr. 2003), DOI: 10.1002/spe.515.

[71] Moritz Lipp et al., « Meltdown: Reading Kernel Memory from User Space », *in*: *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[72] H. Mantel, « Preserving information flow properties under refinement », *in*: *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, May 2001, pp. 78–91, DOI: 10.1109/SECPRI.2001.924289.

[73] Stephen Mccamant and Greg Morrisett, « Evaluating SFI for a CISC architecture », *in*: *In 15th USENIX Security Symposium*, 2006, pp. 209–224.

[74] Barton Miller, Lars Fredriksen, and Bryan So, « An Empirical Study of the Reliability of UNIX Utilities. », *in*: *Commun. ACM* 33 (Dec. 1990), pp. 32–44, DOI: 10.1145/96267.96279.

[75] J Strother Moore, « A Mechanically Verified Language Implementation », *in*: *Journal of Automated Reasoning* 5 (1989), pp. 461–492.

[76] Greg Morrisett et al., « RockSalt: Better, Faster, Stronger SFI for the x86 », *in*: *PLDI*, ACM, 2012, pp. 395–404, ISBN: 978-1-4503-1205-9.

[77] Greg Morrisett et al., « RockSalt: Better, Faster, Stronger SFI for the x86 », *in*: *SIGPLAN Not.* 47.*6* (June 2012), pp. 395–404, ISSN: 0362-1340, DOI: 10.1145/2345156.2254111, URL: http://doi.acm.org/10.1145/2345156.2254111.

[78] Leonardo de Moura and Nikolaj Bjørner, « Z3: An Efficient SMT Solver », *in*: *Tools and Algorithms for the Construction and Analysis of Systems*, ed. by C. R. Ramakrishnan and Jakob Rehof, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[79] George C. Necula, « Proof-Carrying Code », *in*: *POPL*, ACM Press, 1997, pp. 106–119.

[80] George C. Necula and Peter Lee, « Safe Kernel Extensions Without Run-Time Checking », *in*: *OSDI*, ACM, 1996, pp. 229–243.

[81] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999, ISBN: 3540654100.

[82] Svetla Nikova, Christian Rechberger, and Vincent Rijmen, « Threshold Implementations Against Side-channel Attacks and Glitches », *in*: *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, Raleigh, NC: Springer-Verlag, 2006, pp. 529–545, ISBN: 3-540-49496-0, 978-3-540-49496-6, DOI: 10.1007/11935308_38, URL: http://dx.doi.org/10.1007/11935308_38.

[83] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Jan. 2002, DOI: 10.1007/3-540-45949-9.

[84] Michael Norrish, « C formalised in HOL », PhD thesis, University of Cambridge, 1998.

[85] *Optimizer Removes Code Necesseray for Security*, https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537, 2002.

[86] Amir Pnueli, Michael Siegel, and Eli Singerman, « Translation Validation », *in*: *Tools and Algorithms for Construction and Analysis of Systems*, vol. 1384, LNCS, Springer, 1998, pp. 151–166, DOI: 10.1007/BFb0054170, URL: https://doi.org/10.1007/BFb0054170.

[87] Thomas Popp, Stefan Mangard, and Elisabeth Oswald, « Power Analysis Attacks and Countermeasures », *in*: *IEEE Design & Test of Computers* 24 (Nov. 2007), pp. 535–543, DOI: 10.1109/MDT.2007.200.

[88] Emmanuel Prouff and Matthieu Rivain, « Masking against side-channel attacks: A formal security proof », *in*: *EUROCRYPT, volume 7881 of LNCS*, Springer, 2013, pp. 142–159.

[89] Ludo Van Put et al., « Diablo: A reliable, retargetable and extensible link-time rewriting framework », *in*: *In IEEE International Symposium On Signal Processing And Information Technology*, 2005.

[90] Joel Reardon et al., « 50 Ways to Leak Your Data: An Exploration of Apps' Circumvention of the Android Permissions System », *in*: *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620, ISBN: 978-1-939133-06-9, URL: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon.

[91] *Reimplement non-asm OPENSSL_cleanse()*, https://github.com/openssl/openssl/pull/455, 2015.

[92] Silvain Rideau and Xavier Leroy, « Validating Register Allocation and Spilling », *in*: *Compiler Construction, 19th International Conference, CC 2010*, vol. 6011, LNCS, Springer, 2010, pp. 224–243.

[93] Ryan Roemer et al., « Return-Oriented Programming: Systems, Languages, and Applications », *in*: *ACM Transactions on Information and System Security - TISSEC* 15 (Mar. 2012), pp. 1–34, DOI: 10.1145/2133375.2133377.

[94] Andrei Sabelfeld and David Sands, « Declassification: Dimensions and principles », *in*: *Journal of Computer Security* 17.*5* (2009), pp. 517–548, DOI: 10.3233/JCS-2009-0352, URL: https://doi.org/10.3233/JCS-2009-0352.

[95] Kai Schramm and Christof Paar, « Higher Order Masking of the AES », *in*: *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, San Jose, CA: Springer-Verlag, 2006, pp. 208–225, ISBN: 3-540-31033-9, 978-3-540-31033-4, DOI: 10.1007/11605805_14, URL: http://dx.doi.org/10.1007/11605805_14.

[96] Robert C. Seacord, *Dangerous Optimizations and the Loss of Causality*, CERT, 2010.

[97] Fredrik Seehusen and Ketil Stølen, « Maintaining Information Flow Security Under Refinement and Transformation », *in*: vol. 4691, Aug. 2006, pp. 143–157, DOI: 10.1007/978-3-540-75227-1_10.

[98] David Sehr et al., « Adapting Software Fault Isolation to Contemporary CPU Architectures », *in*: *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, Washington, DC: USENIX Association, 2010, pp. 1–1, URL: http://dl.acm.org/citation.cfm?id=1929820.1929822.

[99] David Sehr et al., « Adapting Software Fault Isolation to Contemporary CPU Architectures », *in*: *19th USENIX Security Symposium*, USENIX Association, 2010, pp. 1–12.

[100] Selvaraj and Surenthar, *Overview of Intel Software Guard Extensions Enclave Life Cycle*, Dec. 2016, URL: https://software.intel.com/en-us/blogs/2016/12/20/overview-of-an-intel-software-guard-extensions-enclave-life-cycle.

[101] Rui Shu et al., « A Study of Security Isolation Techniques », *in*: *ACM Comput. Surv.* 49.*3* (Oct. 2016), 50:1–50:37, ISSN: 0360-0300.

[102] Joseph Siefers, Gang Tan, and Greg Morrisett, « Robusta: Taming the Native Beast of the JVM », *in*: *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, Chicago, Illinois, USA: ACM, 2010, pp. 201–211, ISBN: 978-1-4503-0245-6, DOI: 10.1145/1866307.1866331, URL: http://doi.acm.org/10.1145/1866307.1866331.

[103] Patrick Simmons, « Security Through Amnesia: A Software-Based Solution to the Cold Boot Attack on Disk Encryption », *in*: *CoRR* abs/1104.4843 (2011), arXiv: 1104.4843, URL: http://arxiv.org/abs/1104.4843.

[104] Laurent Simon, David Chisnall, and Ross J. Anderson, « What You Get is What You C: Controlling Side Effects in Mainstream C Compilers », *in*: *EuroS&P*, IEEE, 2018, pp. 1–15.

[105] Laurent Simon, David Chisnall, and Ross J. Anderson, « What You Get is What You C: Controlling Side Effects in Mainstream C Compilers », *in*: *EuroS&P*, IEEE, 2018, pp. 1–15.

[106] Rohit Sinha et al., « A Design and Verification Methodology for Secure Isolated Regions », *in*: *PLDI*, ACM, 2016, pp. 665–681.

[107] Christopher Small and Margo Seltzer, *Abstract MiSFIT: A Tool for Constructing Safe Extensible C++ Systems*, 1997.

[108] M. Stump, *[patch] C undefined behavior fix*, Jan. 2002, URL: https://gcc.gnu.org/ml/gcc/2002-01/msg00518.html.

[109] Mengtao Sun et al., « Bringing Java's Wild Native World Under Control », *in*: *ACM Trans. Inf. Syst. Secur.* 16.*3* (Dec. 2013), 9:1–9:28, ISSN: 1094-9224, DOI: 10.1145/2535505, URL: http://doi.acm.org/10.1145/2535505.

[110] *Supplementary material*, https://www.irisa.fr/celtique/ext/compcertsfi.

[111] Scut from TESO, *Exploiting format strings vulnerabilities*, Sept. 2001.

[112] The Coq development team, *The Coq proof assistant reference manual*, Version 8.7, 2017, URL: http://coq.inria.fr.

[113] L. Torvalds, *Linux bug in cfs*, May 2007, URL: https://lkml.org/lkml/2007/5/7/213.

[114] Robert Wahbe et al., « Efficient Software-based Fault Isolation », *in*: *SIGOPS Oper. Syst. Rev. 27.5* (1993), pp. 203–216, ISSN: 0163-5980, DOI: 10.1145/173668.168635, URL: http://doi.acm.org/10.1145/173668.168635.

[115] Xi Wang et al., « Undefined behavior: what happened to my code? », *in*: *APSys '12, Seoul*, ACM, 2012, p. 9.

[116] Zhaomo Yang et al., « Dead Store Elimination (Still) Considered Harmful », *in*: *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, USENIX Association, 2017, pp. 1025–1040, ISBN: 978-1-931971-40-9, URL: http://dl.acm.org/citation.cfm?id=3241189.3241269.

[117] Bennet Yee et al., « Native Client: A Sandbox for Portable, Untrusted x86 Native Code », *in*: *Commun. ACM* 53.*1* (2010), pp. 91–99, ISSN: 0001-0782, DOI: 10.1145/1629175.1629203, URL: http://doi.acm.org/10.1145/1629175.1629203.

[118] Bin Zeng, Gang Tan, and Greg Morrisett, « Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing », *in*: *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, Chicago, Illinois, USA: ACM, 2011, pp. 29–40, ISBN: 978-1-4503-0948-6, DOI: 10.1145/2046707.2046713, URL: http://doi.acm.org/10.1145/2046707.2046713.

[119] Jianzhou Zhao et al., « Formalizing the LLVM Intermediate Representation for Verified Program Transformations », *in*: *SIGPLAN Not.* 47.*1* (Jan. 2012), pp. 427–440, ISSN: 0362-1340, DOI: 10.1145/2103621.2103709, URL: http://doi.acm.org/10.1145/2103621.2103709.

[120] Lu Zhao et al., « ARMor: Fully Verified Software Fault Isolation », *in*: *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, Taipei, Taiwan: ACM, 2011, pp. 289–298, ISBN: 978-1-4503-0714-7, DOI: 10.1145/2038642.2038687, URL: http://doi.acm.org/10.1145/2038642.2038687.

Our papers

[19] Frédéric Besson, Alexandre Dang, and Thomas P. Jensen, « Information-Flow Preservation in Compiler Optimisations », *in*: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, 2019, pp. 230–242, DOI: 10.1109/CSF.2019.00023, URL: https://doi.org/10.1109/CSF.2019.00023.

[20] Frédéric Besson, Alexandre Dang, and Thomas P. Jensen, « Securing Compilation Against Memory Probing », *in*: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 29–40, DOI: 10.1145/3264820.3264822, URL: https://doi.org/10.1145/3264820.3264822.

[22] Frédéric Besson et al., « Compiling Sandboxes: Formally Verified Software Fault Isolation », *in*: *Programming Languages and Systems*, ed. by Luís Caires, Cham: Springer International Publishing, 2019, pp. 499–524, ISBN: 978-3-030-17184-1.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Compilation pour la protection de la mémoire

Mot clés : Compilation Sécurisé, Isolation, Canaux Cachés

Résumé : Notre société est de plus en plus dépendante de services informatiques et cette tendance est à la hausse. De ce fait, la sécurité de nos systèmes est également devenu incontournable afin d'éviter les conséquences désastreuses que peuvent provoquer d'éventuelles attaques. Cette thèse porte sur la sécurité des programmes et particulièrement en utilisant la compilation pour parvenir à ses fins. La compilation est correspond à la traduction des programmes sources écrits par des humains vers du code machine lisible par nos systèmes. Nous explorons les deux manières possible de faire de la compilation sécurisée : la sécurisation et la préservation. Premièrement, nous avons dévelopé CompCertSFI, un compilateur qui sécurise des modules en les isolant dans des zones mémoires restreintes appelées *bac à sable*. Ces modules sont ensuite incapables d'accéder à des zones mémoires hors de leur bac à sable, ce qui empêche un module malveillant de corrompre d'autres entités du système. Sur le sujet de la préservation, nous avons défini une notion de *Préservation de Flot d'Information* qui s'applique aux transformations de programme. Cette propriété, lorsqu'elle est appliquée, permet de s'assurer qu'un programme ne devienne moins sécurité durant sa compilation. Notre propriété de préservation est spécifiquement conçus pour préserver les protections contre les attaques de type canaux cachés. Cette nouvelle catégorie d'attaque utilise des médiums physique comme le temps ou la consommation d'énergie qui ne sont pas pris en compte par les compilateurs actuels.

Title: Compilation for memory protection

Keywords: Secure Compilation, Isolation, Side-Channels

Abstract: Our society has been growingly dependent on computer systems and this tendency will not slow down in the incoming years. Similarly, interests over cybersecurity have been increasing alongside the possible consequences brought by successful attacks on these systems. This thesis tackles the issue of security of systems and especially focuses on compilation to achieve its goal. Compilation is the process of translating source programs written by humans to machine code readable by our systems. We explore the two possible behaviours of a secure compiler which are enforcement and preservation. First, we have developed CompCertSFI, a compiler which enforces the isolation of modules into closed memory areas called *sandboxes*. These modules are then unable to access memory regions outside of their sandbox which prevents any malicious module from corrupting other entities of the system. On the topic of security preservation, we defined a notion of *Information Flow Preserving* transformation to make sure that a program does get less secure during compilation. Our property is designed to preserve security against side-channel attacks. This new category of attacks uses physical mediums such as time or power consumption which are taken into account by current compilers.