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 : Informatique

Par

« Julien LEPILLER »

« Vérification d'isolation de fautes logicielle »

« Verifying Software Fault Isolation »

Thèse présentée et soutenue à « Rennes », le « 11/12/2019 »
Unité de recherche : IRISA

REMERCIEMENTS

Je tiens à remercier Thomas Jensen et Frédéric Besson pour avoir encadré cette thèse. Ils ont su être présents aux moments cruciaux pour m'aiguiller dans mes réflexions et mon travail. Plus généralement, je voudrais remercier toute l'équipe Celtique qui m'a accueillie pendant ces trois dernières années et avec qui j'ai eu plaisir à travailler.

J'aimerais remercier mes rapporteurs, Marie-Laure Potet et Antoine Miné pour avoir accepté de relire le présent manuscrit. Je veux aussi remercier mon jury, composé de Sandrine Blazy et de Gustavo Petri.

J'adresse également toute ma reconnaissance à Lydie Mabil et à Stéphanie Lemaile, nos deux merveilleuses assistantes d'équipe qui m'ont aidé et aiguillé dans mes différentes démarches administratives.

Je voudrais aussi remercier toutes les personnes qui m'ont écouté et soutenu durant cette thèse, mes parents, Sylvie et Guillaume, et mes sœurs, Oriane et Éloïse (ils n'ont pas eu le choix). Enfin, je remercie Amaury Jacques, les hackers de GNU Guix et les libristes croisés pendant ma thèse, qui ont eu la gentillesse de m'écouter et de m'aider à mieux présenter mes idées.

TABLE OF CONTENTS

Vérification d'isolation de fautes logicielle

Les gros systèmes logiciels sont complexes à maintenir et à personnaliser. De nombreux projets populaires s'appuient alors sur des modules développés par des programmeurs tiers. C'est le cas des modules du noyau Linux, des extensions des navigateurs, des bibliothèques natives de certains langages de programmation ou de toutes sortes de greffons dans toutes sortes de logiciels. Bien que le logiciel hôte soit de bonne qualité, régulièrement entretenu et testé, ses modules n'ont aucune garantie de ce genre et peuvent même être malicieux.

Si l'on veut garantir la sécurité de l'hôte face à ses modules, une solution est d'isoler ces modules. Seulement, s'ils sont complètement isolés, ils sont inutiles. On leur donne donc une interface de programmation qui leur permet d'interagir avec l'hôte. Cette interface devra être indépendante du module et correctement sécurisée. La sureté d'un module peut alors se résumer au fait que ce module n'utilise que cette interface pour interagir avec l'hôte ou d'autres modules.

La sureté doit être assurée dans le cas de certains scénarii d'attaque. Nous définissons deux scénarii : soit l'attaquant est capable de proposer un module arbitraire, soit l'attaquant est capable d'exploiter une faille dans un module.

Software fault isolation (isolation de fautes logicielle, ou SFI) est une excellente technique pour garantir la sécurité des module. Elle se base sur un vérifieur qui s'assure, avant de lancer un module, qu'il est sûr. Le vérifieur peut rejeter le module si celui-ci n'est pas sûr, ou si le vérifieur ne sait pas conclure. Le module n'est alors pas exécuté.

La technique se base sur un système de bac à sable : une zone mémoire est dédiée au module, une autre à son code. L'emplacement de ces zones est déterminé par le programme hôte et passées au module via quelques registres. Si les zones mémoire en question sont correctement alignées, il est certain qu'en réécrivant les bits de poids fort de chaque adresse mémoire auxquelles le module souhaite accéder, celui-ci ne pourra pas accéder à de la mémoire en dehors de ses zones dédiées, et donc interagir avec le module autrement qu’en appelant son code.

La réécriture est introduite par le compilateur, mais il n’est pas nécessaire de faire confiance au compilateur utilisé : un attaquant qui n’utiliserait pas le compilateur pourrait fournir un binaire sans réécriture. C’est le rôle du vérifieur que de s’assurer que le module réécrit correctement ses adresses. Pour cela, il vérifie aussi d’autres propriétés sur le binaire. Par exemple, l’alignement de blocs de codes, la cible des sauts et l’emplacement des réécritures par rapport à l’utilisation des adresses.

Dans cette thèse, nous proposons une nouvelle définition de SFI, basée non pas sur un critère purement syntaxique, mais sur une propriété sémantique. Il s’agit de définir la propriété le plus largement possible pour en extraire précisément les points qui définissent la sécurité, et non pas seulement des critères suffisants. Ainsi, nous continuons d’allouer une zone mémoire dédiée au bac à sable, mais le code n’est plus séparé de celui de l’hôte. La pile est partagée entre l’hôte et le module. Le module a alors accès en lecture et en écriture au bac à sable, à ses trames de pile et en exécution à son code et aux fonctions de l’interface de programmation, que nous appelons la *bibliothèque de confiance*, dans la tradition de SFI. La pile en particulier est un problème, car l’appartenance d’une trame à tel ou tel « domaine » (hôte ou module) n’est pas une donnée statique, mais dynamique, et est susceptible de changer au cours du temps.

Nous proposons de définir la sécurité d’un module au moyen d’une *sémantique défensive*, c’est-à-dire une sémantique qui ressemble à la sémantique standard du langage, mais qui effectue des vérifications supplémentaires à l’exécution. Un module est alors dit sûr si sa sémantique défensive n’est pas bloquée.

Nous définissons une théorie simplifiée de l’interprétation abstraite, qui permet d’obtenir un analyseur statique à partir d’une sémantique abstraite. Cette sémantique abstraite définie une abstraction de l’exécution d’un programme, examiné par l’analyseur statique. Avec la sémantique abstraite, nous devons définir une concrétisation, qui à partir d’un état abstrait renvoie un ensemble d’états concrets. En analysant le programme, on associe à chaque point de programme un état abstrait. Si l’abstraction est correcte, la concrétisation de ces états donne un sur-ensemble des états qui sont effectivement atteints à ces points de programme durant l’exécution du module. Nous suivons ensuite une méthodologie qui nous permet de découper la preuve de correction de l’abstraction en plusieurs preuves de correction plus petites, en passant par des sémantiques intermédiaires qui présentent chacune un aspect important de l'abstraction.

Suite à cela, nous avons implémenté un prototype d'analyseur en se basant sur bincat, un outil d'analyse binaire existant, basé lui aussi sur l'interprétation abstraite. Nous y avons apporté nos propres domaines abstraits et modifié la fonction principale et le domaine concret pour correspondre à nos besoins d'analyse. Nous avons effectué des expérimentations pour valider nos attentes : avec des programmes spécifiquement non sûrs, des programmes sûrs et des programmes plus gros, qui utilisent un compilateur qui produit des modules sûrs. Nous avons aussi mesuré le temps d'exécution de l'analyseur, qui varie essentiellement en fonction de la quantité de boucles imbriquées dans les fonctions.

La dernière partie de cette thèse s'articule autour de l'extension du travail déjà effectué au cas d'une exécution parallèle du programme hôte et du module. Dans ce nouveau contexte, le programme exécute plusieurs fils d'exécution qui commencent tous dans le programme hôte. Chacun de ces fils d'exécution peut alors exécuter du code de l'hôte ou du module. Il convient de redéfinir la propriété de sécurité qui ne convient plus : le blocage d'un fil d'exécution n'entraine pas le blocage des autres fils. Pire encore : si notre langage proposait des fonctionnalités de synchronisation, il pourrait y avoir des interblocages, mais ce ne sont pas des problèmes de sécurité. Si le programme ne s'exécute plus, il n'y a pas de vulnérabilité.

On pourrait penser qu'une sémantique d'entrelacement suffirait à représenter tous les aspects de l'exécution parallèle d'un programme. Une sémantique d'entrelacement est une sémantique où à chaque pas, c'est un fil d'exécution choisi arbitrairement qui fait un pas d'exécution de son côté, et où la mémoire est partagée entre tous les fils. Malheureusement, l'exécution de programmes sur du vrai matériel ne fonctionne pas de cette manière : chaque cœur du processeur a son propre cache, peut éventuellement réordonner des instructions, etc. En utilisant un modèle mémoire faible, il est possible de représenter, de manière abstraite, ces comportements du matériel en représentant plus de comportements possibles du programme. Plus un modèle mémoire permet de comportements, plus on dit qu'il est faible. On peut les comparer en comparant les comportements autorisés ou non par ces modèles.

Nous avons proposé un modèle mémoire faible suffisamment faible pour représenter plus de comportements que ceux observables sur un multi-processeur x86 ou arm par exemple. Ce modèle est utilisé avec une sémantique axiomatique (ce qui est habituel pour les sémantiques de programmes dans des modèles mémoire faibles) et la propriété de sécurité est définie en fonction des événements qui ont lieu dans cette sémantique. Chaque événement est soit un calcul local, une lecture en mémoire, une écriture en mémoire, un saut ou l'exécution de code de l'hôte. La propriété de sécurité définit ce qu'est un événement sûr : par exemple, une écriture doit avoir lieu soit dans le bac à sable, soit dans la trame de pile actuelle, de même pour l'écriture en mémoire.

Comme précédemment, nous abstrayons cette sémantique concrète pas à pas, en plusieurs sémantiques intermédiaires. À chaque étape, nous montrons que la sécurité du module dans la nouvelle sémantique, plus abstraite, entraine la sécurité du module dans l'ancienne, plus concrète. Ainsi, nous abstrayons d'abord le bac à sable, en permettant la lecture d'une valeur arbitraire, puis nous abstrayons le modèle mémoire par un modèle d'entrelacement, puis nous abstrayons le tout en une sémantique non parallèle.

Enfin, cette thèse se termine là où elle a commencé : en montrant que la sécurité de cette dernière sémantique se réduit à la sécurité d'une sémantique défensive. Précisément, la première sémantique intermédiaire après la sémantique défensive qui définit la sécurité d'un module dans une exécution non parallèle. Cela signifie alors que le même analyseur est capable de déterminer la sureté d'un module, indépendamment du modèle mémoire considéré.

Ce travail pourra être étendu ensuite à plusieurs modules. Les preuves actuelles se font sous l'hypothèse qu'un seul module sera exécuté. S'il y a plusieurs modules, cela changerait un peu la sémantique défensive, et donc le contenu des preuves. Il est cependant probable que la sémantique abstraite et l'analyseur n'auront pas besoin de changements pour pouvoir vérifier un nombre arbitraire de modules. Il pourrait être intéressant d'implémenter cet analyseur et les preuves de correction dans un assistant à la preuve comme Coq, pour assurer la fiabilité des résultats. L'implémentation pourrait aussi être améliorée, entre autres pour raffiner les conditions sur les gardes, et pour exécuter une version plus grossière mais plus rapide et toujours correcte avant de lancer une version plus subtile si la première n'a pas fonctionné. Enfin, il serait intéressant d'étudier le lien entre ces deux définitions sémantiques de SFI et la définition syntaxique d'autres implémentation, comme NaCl. Nous pouvons nous demander si, et dans quelle mesure, l'analyseur que nous avons développé serait capable de vérifier ces modules. Enfin, on pourrait s'intéresser au contenu du bac à sable, qui permettrait d'avoir une analyse plus fine et plus précise, qui tient compte par exemple du comportement de certains modules standards, qui créent leur pile dans le bac à sable. Cela représenterait un défi important, puisque l'abstraction complète du bac à sable est un élément central du raisonnement dans le cas parallèle.

INTRODUCTION

We are now used to have computers with multiple software installed, from different software vendors. Some of the software comes from the same company or group who published the operating system you are running, and some of it comes from other groups. Although some operating systems used to rely on software cooperation for them to run together, modern operating systems and hardware architectures are able to manage uncooperative software. In fact, software are now mostly designed to be uncooperative and act as if they were the only software running on the machine.

One of the techniques to force uncooperative software to cooperate and not step on one another is the use of virtual memory spaces. The memory of a computer is a memory space that maps addresses to values. When two programs run at the same time, and they are not cooperating, it is not guaranteed that they will not try to access the same memory location for different purposes. Modern hardware have an *MMU* (Memory Management Unit) that allows an operating system to set up virtual memory for processes. An MMU is tasked with the mapping of virtual memory addresses of a process to actual memory addresses. The operating system selects a free memory zone for each new process, and creates a memory map for each process, where actual addresses are in this free memory zone. With the help of the MMU, the software can now access the same address as others, while clashes are prevented by the mapping of this address for each program to completely different addresses in the real memory.

Failing to do so can have damaging consequences. In 11th grade, we had to have and use an advanced calculator during some math and science classes. There were different brands and models, but they were all effectively a small computer with a calculator keyboard and a simple operating system. The calculator was even able to load programs from a computer, and exchange files and programs with other calculators of the same model. I was intrigued and quickly found how to create programs for the calculator. It was possible to use a computer to program in C and compile for the calculator's processor. One of my first achievements was a snake game, where a snake has to eat dots that appear at random on the screen and grows with each dot eaten, but must not cross itself.

I showed the game to comrades and immediately became very popular. Some of them had the same model of calculator and we copied the executable to their calculator. I'm very bad at games, so I could not score more than 20 or maybe 30 points. Some of my friends though managed to be way better than me and get more than 100 points! All went well for a time, until the game ended. Friends who made more than 100 points quickly found that, when quitting the game, the operating system had crashed. Thankfully, a hard reset (which made all files and custom programs disappear) was able to fix it.

The reason for the crash was simple, although it took me some time to understand it. My snake game registered the positions of the snake with an array: at each tick, it would update every position in the array with the position of the element after it, or if it was the head, with the new position of the head. Unfortunately, if you were too good at the game and your snake managed to grow beyond 100, the array overflowed... into a memory region that was part of the operating system's memory.

This example shows the importance of using virtual memory, or at least some sort of memory segmentation or isolation. It was a genuine mistake that had almost no bad long-term consequences. But what if the program came from a malevolent programmer? It could have made much more harm, taking over the entire operating system.

Even more recently, we are seeing more and more programs allowing third-party programmers to add features through a module system. These modules are loaded inside the program that hosts them. We can draw a parallel between the operating system and the host program, and between the snake game and the module. At this level, there is no support from the operating system for memory isolation, nor does the hardware help. If the module is not carefully written or the host program does not take specific actions to prevent a disaster, the same kind of outcome may happen.

When the hardware cannot help with isolation of processes or modules (either because there is no MMU or because no hardware really supports isolation of modules in the same address space), we can only rely on software techniques. Chapter 1 introduces such a technique, called *Software Fault Isolation*.

Although we will see there are multiple variants of the technique, it is mostly a syntactic technique. In order to better understand how Software Fault Isolation works, this thesis started by working on a semantic definition of it. We tried to really understand what it meant for a module to be isolated from its host program. We started by designing a small language that acted like an assembly language. This language is presented in

Chapter 3. We came up with a *defensive semantics*, which means a semantics that acts as a normal semantics with additional checks, to model isolated modules. We refined it multiple times to arrive at the current form that we will present in this thesis.

Once we had grasped what it meant for a module to be isolated, we ended up with a semantics property that allowed for more behaviors than other implementations. In order to check whether a particular module respects the property, a single pass to verify syntactic properties was not enough any more. Based on abstract interpretation and a methodology for decomposing a proof of correctness in smaller steps, with intermediate semantics, that we present in Chapter 2, we developed a static analyzer. We present in Chapter 5 the implementation we wrote and experiments we ran on it. Chapter 4 presents the defensive semantics, the intermediate semantics and the abstract semantics the analyzer is based on.

This work has a fundamental flaw: it assumes a single-threaded application. Modern software however often take advantage of the multithreaded hardware we have now. In order to solve this flaw, we studied multithreading models. We wanted to stay as close as possible to actual hardware behaviors, so we decided to study SFI under a weak-memory model, which had never been done before.

A weak-memory model is a memory model where some assumptions about memory in a multithreading context are not held anymore. The reason is that these assumptions are not guaranteed by actual hardware. We will see in Chapter 6 what these assumptions are, and how to define a weak-memory model. Chapter 7 will later define the semantics of our language under this weak-memory model and present a proof that the analyzer we defined before is correct even under this new model, using multiple intermediate semantics.

PART I

Context

CHAPTER 1

SFI: A SECURE EXTENSION MECHANISM

This chapter is derived from a survey written in collaboration with Alexandre Dang.

1.1 Secure Extension by Untrusted Modules

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 in an interpreted language, or any kind of plugin or module in any software. They extend the possibilities of their host software.

Although the host software is usually of good quality, carefully developed and extensively tested, its modules can lack such quality or be downright malicious.

This thesis starts with the question of securing such modules, even if we do not trust them. Expressing a security property for such a system is difficult: sending a user file to a webmail server is expected when the user clicks on the attachment icon. This behavior is normal, and certainly not considered malicious. The same behavior from a disk device driver however, is not expected and can be considered malicious.

So, we can intuitively express the security of an extension in terms of "user expectation": a secure module cannot do something the user does not expect. This definition however is too broad and not formal enough for our purposes. Let's first examine an example of extension usage and then come back to the security property, with a better definition.

Some video games allow their players to add content to the game through "mods". Content added this way are typically assets (game data) and code in a simple extension language, such as lua or python for instance. The code itself can interact with the rest of the virtual environment through predefined functions.

This example shows a very interesting technique: since the language is interpreted, the extension can only interact with specific parts of the host's code, that is especially designed to interact with extensions. This interface between the host code and the extension code is key to the definition of the security property.

Definition 1 (Extension Security)**.** *Given a programming interface between the host program and its modules, defined by the host, a secure module is a module that only uses this interface to interact with the rest of the host program and system.*

Additionally, we need to define a threat model that motivates us to create a security mechanism in the first place. Intuitively, we want an attacker to be able to do anything, except directly compromising the host program:

Definition 2 (Threat Model)**.** *Attackers might distribute malicious extensions to users (the code might not follow our security property), or compromise a vulnerable extension running on a user's computer by controlling any data read by the extension.*

To illustrate this threat model, a first attack scenario could be that the attacker crafts a malicious extension that is installed by a user. That extension does not comply with the security property and interacts directly with the system or parts of the host software that is not in the programming interface. A second scenario is a vulnerable extension, asking for data coming from an untrusted source, such as a website content. A maliciously crafted data (which is allowed by our threat model since attackers control data read by the extension) triggers a buffer overflow that ultimately overrides a return address. Later on, on return of the function, the module jumps to an unexpected address, breaking the security property.

In addition to security, we are also interested in performance. With two secure implementations of an extension mechanism, we would prefer to use the fastest one. However, naive implementations of faster extension mechanisms are also more vulnerable. The challenge is to combine extensibility, security and speed.

In the video game example, extensions are written in a high-level programming language, which gives some guarantees on what it can do or not. This solution however is rather slow, and requires us to believe the interpreter is bug-free, or at least not exploitable. [1] In short, we have an extension mechanism, but not at full speed and with

1. and that is not always the case. For instance, see a recent vulnerability in Firefox's JavaScript engine that was used to distribute malware: `https://www.mozilla.org/en-US/security/advisories/mfsa2019-18/`

security that is difficult to prove. In software where speed is more important such as in kernels, it would be very inconvenient to use a slow interpreted extension language. Therefore, these programs are extended with binary modules.

For the Linux kernel, these are 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.

To further increase the speed of the module, software designers may want to load the module directly in the address space of their software, especially when the software and modules communicate frequently. When two separate software want to communicate, they have to use costly context switching. Kernel modules reside in kernel space while other software would use the `dlopen()` function to load a binary extension.

However, when we gain the speed of native code, we lose the guarantees of an interpreted language, where it's not possible to express code that interacts with parts of the system that were not designed for that purpose. Since every native code is not secure, the challenge is to find a procedure to decide whether a specific extension can be run safely or not.

Software Fault Isolation (SFI) is an excellent candidate to solve that issue of having extensions, speed and security. It was originally presented in the work of Wahbe et al. [41] It supports the idea that third-party programmers are not trusted and provides a verifier, that decides whether an extension can be run safely. We will see in the next section why it is a good candidate and how it works.

This chapter is an overview of different techniques used in different implementations. We will first see in section 1.2 the principles of SFI as defined in [41], with naive implementation details to get a first intuition of the way it works. In section 1.3, we will see and compare implementation choices of different papers, in terms of security, efficiency and practicality when applied to extensions. In section 1.4, we will see and compare different optimizations that can be used to improve the speed of an implementation. Finally, in section 1.5, we will see how to write a verifier for SFI.

1.2 Principles of SFI

We introduce here the main ideas of *Software Fault Isolation*. All recent implementations are derived from these principles. The goal of SFI is to allow a host program to safely execute potentially dangerous modules in its own address space. To accomplish that, these modules are isolated in specific memory areas called *sandboxes*. In fact, each module has two sandboxes: one for its code, and the other for its data.

The SFI approach has two main components: the first one is the rewriting of the untrusted module to prevent it from accessing any memory out of its sandbox. The second component is the verification of the module's code before loading it into memory. This step checks whether the rewriting done in the previous part is still present and valid in the code.

1.2.1 Foundations of SFI

The main principle behind SFI was first presented in the work of Wahbe et al. [41]. The implementation described in their paper was made for a RISC architecture like MIPS or Alpha.

We consider that an untrusted code is effectively contained in the sandbox if the following three security properties are true:

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

— **Memory safety**: untrusted modules will not do any `write` operation out of the sandbox

— **Control flow**: every control flow transfer from the untrusted module to the host program is identified and verified

The first property protects the host against self-modifying code which can bypass the SFI measures. *Memory safety* prevents any illegal access to the memory of the host program. The last property allows only licit interactions between the host and its modules. SFI forbids any call from untrusted modules that could modify the control flow of the program. If the control flow is compromised, it can lead to an unexpected behavior of the program which we want to avoid.

The whole SFI chain of execution is presented in Figure 1.1: a module code is written and compiled by a third-party developer. The developer may comply with the requirements of SFI and use a specific generator, that generates an effectively contained binary module. As we have seen before, the threat model allows an attacker to be a third-party developer, so we cannot assume all extensions are generated with the generator. Then, the extension is distributed in some way to end-users. For instance, through an app store or by a direct download link from the developer's website. Here, an attacker could potentially act and replace the downloaded code with its own. Finally, once the user downloaded an extension, they run a verifier on their computer to check that the code they downloaded is effectively sandboxed.

If the verification fails the module is rejected and is not executed. With this model, we only have to trust the verifier in order to trust that accepted modules are effectively contained. This is one advantage of SFI: only the verifier needs to be in the *Trusted Computing Base* (TCB). The TCB is the set of computer hardware and software whose bugs may lead to a security issue. If we trust that the TCB does not have bugs, then the rest of the system is secure. In particular, the TCB ensures that the rest of the system does not have bugs, or cannot exploit them to gain privileges, etc.

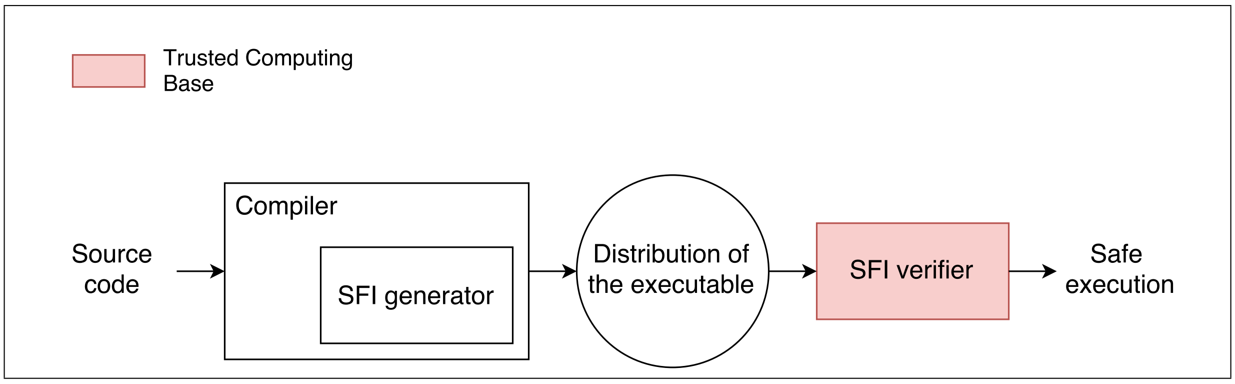


Figure 1.1 – SFI chain

1.2.2 Code Generation

To protect a program from its modules, the generator will restrict every write and jump instructions of the modules to addresses of their sandbox.

The generator has to face three issues to do so. The first one is to introduce protection mechanisms before every potentially dangerous instruction. For example it prevents any destination address of jump instructions to be located out of the sandbox.

Secondly, we have to make sure that these protection mechanisms cannot be avoided.

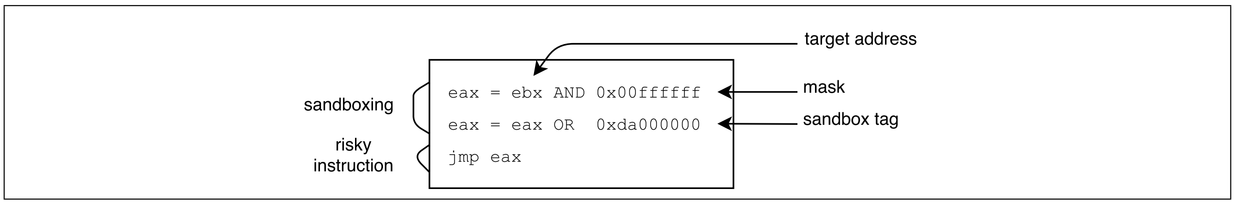
Finally, the transformations injected have to authorize only legal calls from the sandbox to the host program by using entry points specified by the latter. This is the interface we talked about in the introduction of this chapter.

Confining memory accesses. The sandbox is a contiguous memory area reserved for a module's use. The confinement to the sandbox is done by a rewriting process: any address used in the module as a target to a read, a write or a control flow jump is rewritten at run-time to an address that is guaranteed to be inside the sandbox. The sandbox size is chosen to be a power of two. This requirement eases the confinement of the modules in their sandbox by allowing the use of bit arithmetic which accelerates the rewriting process. In fact, we only need to rewrite the most significant bits of addresses to ones that match the sandbox area.

In future examples, we will limit ourselves to the addresses whose most significant bits are `0xda`. This sequence of bits is also called a *tag*. Each tag is specific to a unique sandbox and this designation will be used repeatedly in this thesis.

The transformation is simple in theory: the targeted addresses simply have their most significant bits replaced by the tag of the sandbox. In the case of direct addressing, compilers can easily rewrite the address, or fail to compile an explicitly incorrect program. In the case of indirect addressing, things are more complicated. The destination address is stored in a register and its value cannot always be known at compile time.

To address these situations, SFI injects runtime modifications in the code of the untrusted module. These modifications consist in the rewriting of the address before the jump or store, and are called *sandboxing*.



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

Figure 1.2 – Pseudo code of the sandboxing operation

Figure 1.2 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 reinitialized before. Hence we are sure that the `jmp` instruction will target a location in the sandbox of the untrusted module. Note that the sandboxing does not change the behavior of the module if the targeted address was already in the sandbox. For the write instructions the principle is the same. Sandboxing instructions are injected before every write that can endanger the program.

Protection of the sandboxing mechanism. We made sure in the previous section that any untrusted module cannot either jump or write on a location out of its sandbox. But this first implementation was very naive: we also need to protect the sandboxing operations, to prevent any malicious code to bypass the runtime checks inserted by SFI. Using the example in Figure 1.2, we could imagine code which directly jumps on the `jmp eax` instruction. To protect the sandboxing, the solution found by SFI is to reserve dedicated registers exclusively used for sandboxing. These registers will not be available anymore for the rest of the code. Naive sandboxing requires three dedicated registers for each sandbox. A first register is used to keep the mask value (we used the immediate value `0x00ffffff` in Figure 1.2). A second register is reserved to store the tag of the corresponding sandbox (we used the immediate value `0xda000000` in Figure 1.2). The third dedicated register is used to manage the operations contained in the sandboxing, in our example in Figure 1.2 it is the `eax` register. The first two registers are necessary in implementations where the tag and sandbox size is not known at compile time, because there might be more than one module for instance. The third register (`eax` in our example) is ensured to only store addresses that are inside the sandbox during the whole execution. Then even if malicious code can jump directly to the instruction `jmp eax`, we will still be sure that the next instruction stays in the sandbox.

The worst case scenario would be that the value stored in `eax` was wrong and the untrusted module crash or has unexpected behavior, but always contained in the sandbox. Note that since the security property is that the untrusted module stays contained in its sandbox, crashes and unexpected behaviors are considered safe.

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. SFI manages to reduce the number of dedicated registers to five by sharing the same mask for both sandboxes, and we could reduce the number to four by using a statically-defined mask.

Allocating five registers for the sandboxing means that they cannot be used by the program. This is not a problem on RISC architectures like MIPS, Alpha or RISC-V that have 32 general-purpose registers. Indeed, experiments in the original paper [41] shows that removing five registers from GCC's register allocation phase did not significantly impact performance of test programs. For other architectures, we will see that there are different mechanisms to work around this limitation, without sacrificing security.

Controlled interactions with the protected program. It is necessary for SFI to also control the different interactions that modules have with the main program. Without restrictions, malicious modules could, for example, make function calls with wrong parameters which could compromise the state of the main program.

To avoid such a situation, SFI needs the host program to define an interface which describes all the authorized entry points available to external modules. This interface is assumed to be composed of *stubs*, i.e. small functions that execute checks on their parameters, before calling the actual function from the host program. It is assumed that these checks are enough to ensure that the module cannot corrupt its host program by calling stubs of the interface.

Any other means of interacting with the host module or the rest of the system is forbidden. In particular, system calls are forbidden in modules: instead, the module must use a stub that then calls a function from the main program that implements the system call. An implementation could for instance add a stub for a system call. The module would call that stub to perform the system call. Another implementation could refuse to add stubs for system calls, but allow some high-level functions that will, in its expected behavior, make some system calls.

As a nice side effect, such modules are perfectly portable across operating systems on which the host program can run, although they still of course depend on the processor architecture.

1.2.3 Verification

As we have seen before, an attacker could provide a malicious module to a user either as being a developer of that module or by substituting a legitimate module with its own. Because of this attacker model, the user needs a protection mechanism. We could think of a central authority delivering certificates, but our experience in, for instance, mobile application stores, show that it does not work well in practice. Instead, the user is provided with a verifier, a trusted piece of software that checks that all the necessary sandboxing is indeed present in the untrusted module.

The verifier is part of the Trusted Computing Base. This means that it must not have bugs. Indeed, if it works well, it is able to reject incorrectly sandboxed untrusted modules, which could result from a faulty implementation of the generator, or from attacker provided modules. The verifier still relies on the generator, but not critically. The generator produces a binary with a certain format that should be easy to analyze and the verifier simply checks that this format is respected (e.g. that all writes and jumps are sandboxed).

We will see in section 1.5 some verifier designs. They mostly work the same way though: a first pass disassembles the binary. A second pass checks that the SFI properties are respected and that no code that was not disassembled is reachable from the untrusted module, except for a trusted library outside of the sandbox.

		nacl	misfit	pittsfield	ARMor	xfi	psfi	fbgi	bakersfield
References		[42][33]	[38]	[27]	[45]	[19]	[25]	[12]	[22]
security model	malicious module [2]	✓	×	✓	×	×	✓	×	✓
architecture	ARM	✓	×	×	✓	×	✓	×	×
	x86	✓	✓	✓	×	✓	✓	✓	×
	x86_64	✓	×	×	×	×	×	×	✓
transformation	IR	×	×	×	×	×	✓	✓	×
	assembly	✓	✓	✓	×	✓	×	×	✓
	binary	×	×	×	✓	×	×	✓	
Techniques	Bundles	✓	× [3]	✓	×	× [3]	×	× [4]	✓
	Return	pop+jmp	outside	× [5]	shadow	scoped	shadow	no write	× [5]
	Self modi-fying code	✓	×	×	×	×	×	×	×
application	kernel	×	×	×	×	✓	×	✓	×
	web browser	✓	×	×	×	×	×	×	×
	language extension	✓ [37][39]	×	×	×	×	×	×	×
	general purpose	×	✓	✓	✓	×	✓	×	✓
Certified		✓ [30]	×	×	✓	×	✓	×	×

Table 1.1 – Summary of techniques

2. if ×, the module is still protected against an external attacker that interacts with it
3. Indirect calls are checked dynamically
4. This technique does not reserve memory blocks, but uses access checks on every byte read, written or executed. The return address is not allowed to be modified.
5. Secure on single-threaded applications

1.3 Implementations

In this section, we will show the major implementation techniques used in different papers. Instead of showing each individual paper and explaining the techniques they used, this section is organized around the techniques themselves, while citing and explaining differences and common points between implementations that use the technique. You can refer to table 1.1 to get a summary of techniques and applications used in different implementations.

1.3.1 Splitting the Code into Constant Bundles

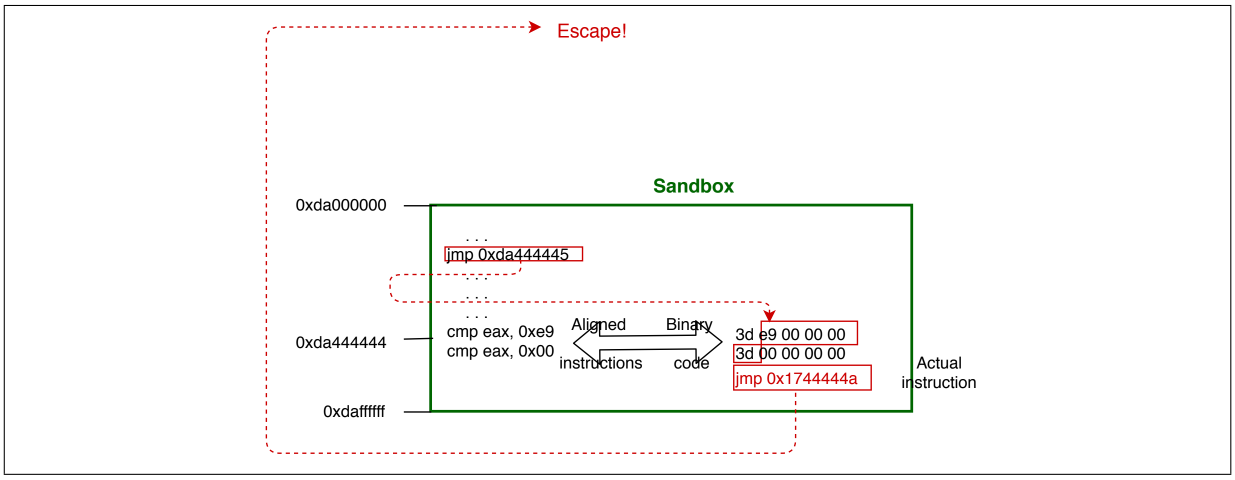


Figure 1.3 – Jumping on an unaligned instruction to bypass SFI

In CISC architectures, instructions are not of a constant size, and the huge number of different possible instructions makes it very likely that reading an instruction starting at the middle of another will execute something meaningful for the processor. In these architectures, it is perfectly possible to hide an arbitrary jump inside a seemingly innocent instruction, as shown in Figure 1.3: by decoding the instructions one by one, an analyzer would be lead to decode the instruction at 0xda444444, which is a simple comparison instruction, just as the next instruction. However, in reality, this instruction is machine code, shown on the right of the figure. The jump before them is misaligned:

it does not target 0xda444444, but one byte later, which happens to be decoded by the processor as another jump instruction, which exits the sandbox.

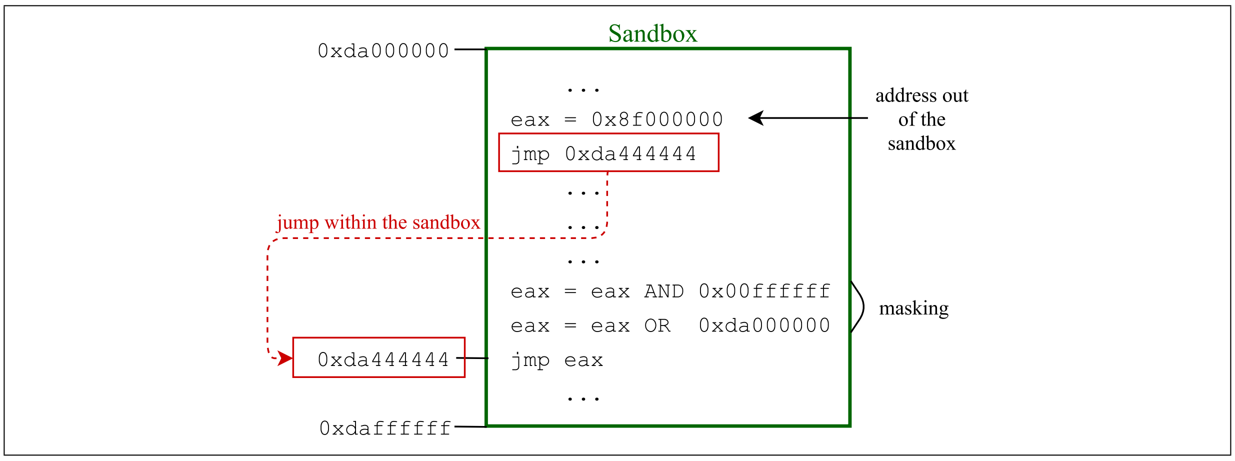


Figure 1.4 – Vulnerability of basic SFI

Additionally, it is easy to imagine how a malicious module could present sandboxing instructions before every jump, and yet be able to jump to arbitrary addresses. As shown in Figure 1.4, one could write the target address to a register and jump directly to the jump instruction after the sandboxing, bypassing it entirely. The malicious `jmp` allows the attacker to bypass the masking operation put by the techniques of SFI and then, reach illegal code.

To avoid this, Pittsfield [27] suggests to divide 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 conditions 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 aligned with 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.

Drawbacks of this approach are the increased size of the code but also the overhead due to added `nop` instruction. Indeed the Pittsfield paper[6] judged that between 25 and 50 percent of the slowdows encountered was due to the additional `nop`s used to pad instruction chunks.

1.3.2 Protecting the Return Address

The return address is a difficult case for SFI. Indeed 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 modifies the return address in the stack after the masking operation, this could disrupt the flow of the host program.

Since this issue only appears in the context of a multithreaded program, some implementations have choosen not to implement counter-measure, which is insecure in the general case, but is still secure in the case of a mono-threaded execution. In this thesis, we are interested in multithreaded programs, so we now examine different ways to prevent this time-of-check to time-of-use race.

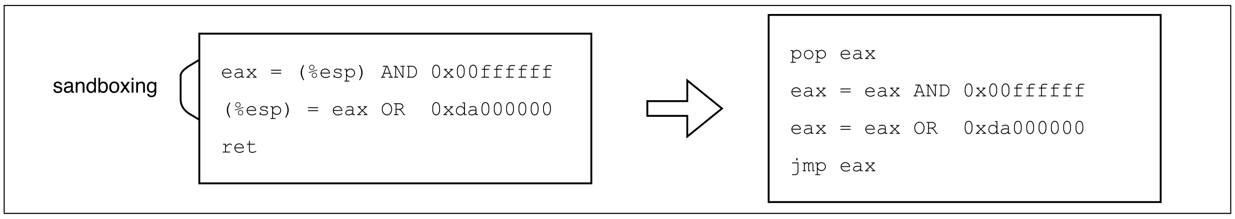
Changing the Return Instruction

A common technique to tackle this issue is to replace the `ret` instruction with a `pop+jmp` combination presented in Figure 1.5. The pseudo-code on the left masks the return address directly on the stack. Since stack memory is reachable from other threads of the module, it is possible that the value pointed to 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`.

McCamant and Morrisett [27] have done exactly that in their initial implementation, but decided to go back to a simple `ret` because of performance issues. Indeed, this technique does not make any use of jump target caches. The technique is also used in

6. See [27], page 8

Zeng et al [43], although they also insert an additional check to ensure they return to the caller, and not only to a module's instruction.



```
eax = (%esp) AND 0x00ffffff
(%esp) = eax OR  0xda000000
ret
```

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

Figure 1.5 – Transforming `ret` into `pop+jmp`

Scoped Stack

A scoped stack [19] is a slightly more restrictive stack than the usual one. A module can access data on this stack only statically. That means that the only way to read and write to the stack, is via a constant positive offset of the stack pointer, at each program point. A different stack called an access stack can be used when it is necessary to have a stack that can be accessed via computed addresses.

At each point in the program, the depth of the scoped stack is known statically, because only `push`, `pop`, `call` and `ret` can modify it. Direct modification of the stack pointer is forbidden. Combined with the fact that a write is done at a statically known offset from the stack pointer, it is easy to see when a write occurs outside the stack or on the return address, and reject such modules.

Control Stack

A control stack (or shadow stack) is a special stack that records return addresses of functions. In ARMor [45], a routine copies and restores return addresses from and to the normal stack upon function call and return. The shadow stack is protected against access by the sandbox, 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. This technique requires a dedicated register to point to the shadow stack.

MiSFIT [38] uses a per-thread global state outside of the sandbox to record the return addresses of module functions that were called, and a special procedure is used to read it and jump to it on function return. This is not exactly a scoped stack, since this data is not necessarily stored in a stack, but the technique and usage is very similar. Having the return address outside of the sandbox ensures that the module cannot modify it. Compared to the shadow stack technique, this looses processor optimizations and caches for the return address, but it does not require an additional dedicated register.

1.3.3 Interfacing with Trusted Libraries

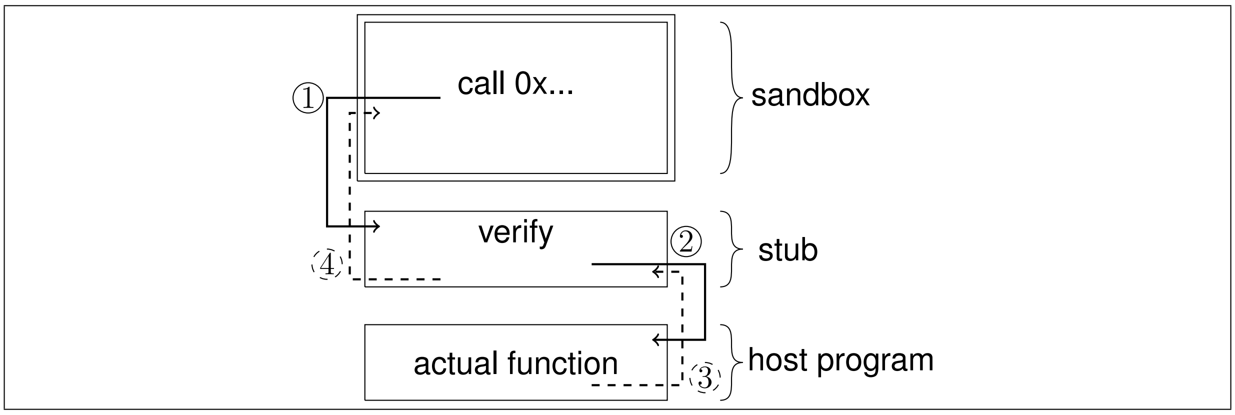


Figure 1.6 – Overview of the work of a stub.

Wahbe et al. [41] introduce the concept of stubs in the context of SFI. Stubs are small, trusted pieces of code, available in a statically known location outside the sandbox. Module code is allowed to call them directly, and their role is to dynamically check the arguments passed to the host function or system call they protect. Figure 1.6 summarizes the role of the stub:

— **1**: Direct call to the stub;

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

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

The multiple implementations seen are quite similar to these stubs but hold different names. Nacl [42] uses the concept of trampoline and springboard which are also part of the sandbox. On x86, trampolines are also used to execute a `far call` to a library function that is in another segment, while springboards restore the sandbox environment before returning to the isolated module.

PSFI [25] and MiSFIT [38] both allow function pointers to be used to call the trusted library. To quickly check that the module is correctly sandboxed, SFI uses a sandboxing operation. Similarly, instead of statically verifying the possible value of a function pointer, these implementations verify the content of the function pointer at runtime. They have a hash table containing the set of authorized function pointers outside of the sandbox and allow only a static call to a procedure that checks whether a function pointer is in the hash table. If not, the pointer is rejected and the function is not called.

1.3.4 Transformation Point

In order to ensure safety of a module, SFI implementations need to apply transformations (mainly adding sandboxing) to the code. Different implementations choose a different place in the build toolchain to apply its transformations, with different benefits and drawbacks. The vast majority of implementations choose to modify the compiler to output sandboxed assembly code. Here, we highlight two outliers.

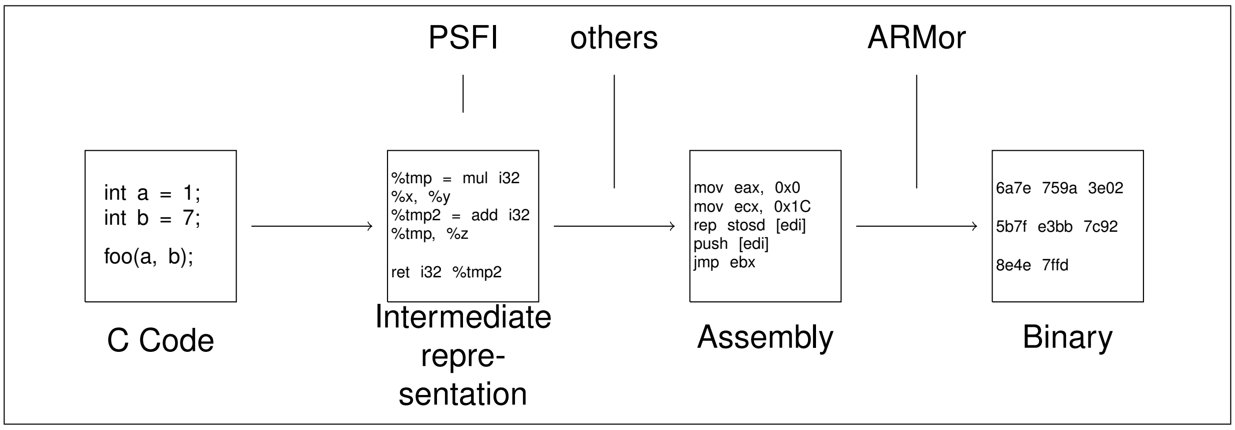


Figure 1.7 – Transformation point in different implementations

Binary Rewriting

This method consists in rewriting the module at link-time, between the assembly and the binary representation. Link-Time Optimization can be used to decrease the overhead. ARMor [45] uses Diablo, a link-time binary rewriting framework to rewrite the binary output by GCC.

High-level Transformation

Psfi [25] is the only implementation which rewrites the code at a higher level than assembly. They are taking advantage of the main theorem of CompCert [26] to produce isolated modules. CompCert is a formally verified C compiler. It is provided with a main theorem which says that, if it produces a result, the behavior of the resulting binary is a behavior of the initial sources, if it does not have undefined behaviors. CompCert is written and proved with the Coq [40] proof assistant. Coq is a programming environment in which one can express programs, mathematical properties and proofs. The assistant is able to follow a proof development and validate every step of reasoning, which gives a high assurance that a mathematical property proved in Coq is actually true.

In PSFI, the transformations are done at the CMinor level, an intermediate language that can be targeted by many languages, but that is still high-level enough to then be compiled to assembly for different architectures.

At this level, they isolate modules by targeting all pointer dereferences for writes and calls and add the sandboxing mechanism with simple `if..then` conditionals. Additionally, since CMinor is the language for optimizations, they benefit from it: runtime checks can be optimized away by the compiler.

The benefits are that they take advantage of the optimizations of the compiler which can reduce the sandboxing overhead and also that the code transformations introduced by SFI are easily portable. A drawback is that the security property is not easily provable by a verifier after the compilation phase since the compiler may modify the sandboxing instructions during the code optimizations. More details are given on the verification of a high-level code generation in Section 1.5.3.

1.3.5 Self-modifying Code

One of the fundamental requirements of SFI is that the verifier checked every instruction that is going to be executed by the module. Self-modifying code can therefore not be allowed, since the generated instructions have not been verified statically before the module's execution.

However, some applications, such as JIT compilers, need to modify the code they execute for better efficiency. In NaCl [4], 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 1.3.1.

— 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 1.8 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 semantic of that instruction is unknown.

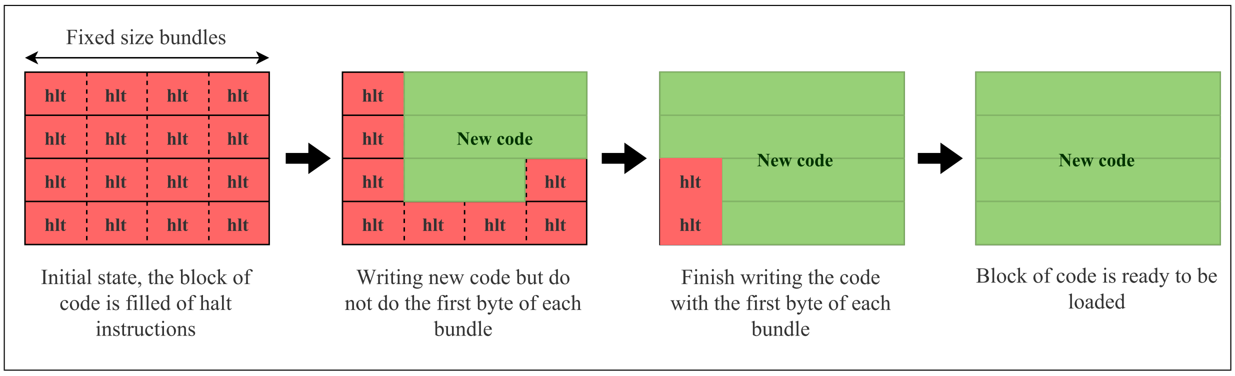
— To unload a code, the implementation writes back the `hlt` instructions, starting with the first byte of each affected bundle. In this way, any execution in this invalidated region will crash the module. Then, marking the region free can happen only when the implementation knows no thread is currently executing in the affected bundle. For that purpose, it waits for each thread to enter a trampoline (Section 1.3.3), 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 using an eight-byte atomic write (because 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.

1.4 Optimizations

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


Fixed size bundles
hlt
New code
Initial state, the block of code is filled of halt instructions
Writing new code but do not do the first byte of each bundle
Finish writing the code with the first byte of each bundle
Block of code is ready to be loaded
Figure 1.8 – Loading code during runtime with SFI

transformed binary.

Most architectures have a mechanism to give read, write and execution privileges to memory areas. For instance on x86, segments can have a NX flag, forbidding execution, and memory pages can be read-only or read-write. On x86_64, memory pages can handle execution as well as write privileges. This mechanism is exploited in most operating systems to disable execution on the stack, and the heap, and writes to the code of a program[7]. This is especially useful to prevent writes to the code section from self-modifying code, and execution of arbitrary code from the data section.

Moreover x86-32's segment mechanism can prevent jumping, reading or writing outside designated areas. For instance, NaCl [42] and Pittsfield [27] 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.

1.4.2 The Art of Choosing the Mask

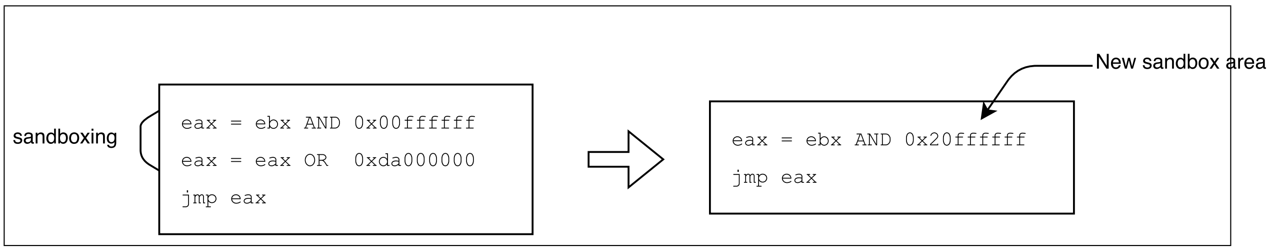
The sandboxing operation usually uses two instructions as shown on the left side of Figure 1.9. A first AND instruction to turn off the bits matching the tag of the SFI memory region, then an OR instruction to set these bits to the sandbox tag.

Pittsfield [27] suggests using tags with only one bit set for the sandboxed code and sandboxed data. The reason is that with a one-bit tag, we can replace the sandboxing

7. As long as the binary has metadata to request this protection mechanism, see the Executable and Linking Format specification [13], at page 2-3. The program header (used to specify a list of memory regions that must be loaded to prepare the program for execution) contains a flag to specify read, write and execution privileges.

operation with a simple `and`: by setting every tag bit to zero, except the bit that makes up the tag, the resulting address has either the correct tag (with one bit set), or the zero-tag (with no bit set). If the tag is correct, execution continues safely, and if it is incorrect, execution stops, since data is read/written or code is executed from a memory that does not have the corresponding priviledges.

For example `[0x00000000 - 0x0fffffff]` is the zero-tagged region, `[0x10000000 - 0x1fffffff]` is the sandboxed data and `[0x20000000-0x2fffffff]` is the sandboxed code. Pointers are rewritten either to the sandbox' tag, or to the zero tag which was reserved for this situation. We see in Figure 1.9 that by placing the sandbox areas cleverly, the AND and OR operations reduce to a single AND operation for the sandboxing.



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

```
eax = ebx AND 0x20ffffff
jmp eax
```

Figure 1.9 – Reducing the sandboxing to a single instruction

This optimisation has also been reused in NaCl [42] and BakerSFIeld [22].

1.4.3 Register Management

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 1.10(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) Classic sandboxing

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

(b) SFI with dedicated registers

Figure 1.10 – Register management for sandboxing

Dedicated Registers

The naive implementation main issue is obvious: the sandboxing operation needs at least four instructions! The cost of sandboxing becomes quite high.

An idea already used in the work of Wahbe et al. [41] is to use dedicated registers for the sandboxing operations. In the previous example in Figure 1.10(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 1.10(b), reduced to two sandboxing instructions. Additionally, using a dedicated register means that, the register will always contain an address in the sandbox, reducing the number of sandboxing instructions when the same address is used multiple times.

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 some architectures with few general purpose registers like x86-32 which only has eight.

Register Management with Liveness Analysis

Another possibility for register management is possible thanks to the use of static analysis and especially liveness analysis [43]. Liveness analysis allows the system to know whenever a register is considered dead which means that the value stored in the register will not be used in the future instructions. In our case knowing the dead registers allows the SFI techniques to use these registers for the sandboxing since these registers' values can be modified without hindering the program. Thus, when there are enough dead registers, the sandboxing operations will be similar to the one with dedicated registers as in Figure 1.10(b). But when available registers are not enough, the classic implementation will take place as in Figure 1.10(a). This implementation can be considered a compromise between the two previous solutions: it both optimizes the use of registers and avoids the overhead due to register pressure in the dedicated registers proposition, at the cost of not being able to skip a sandboxing safely.

1.4.4 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. Since these registers are repeatedly used after being set, it might be ineffective to check their value before each use. To optimize this case, similarly to loop optimization in Section 1.4.6, these registers are checked only once, when their value is modified. Afterwards they do not need to be checked when used with a small offset.

This optimization is implemented with so-called guard zones around the sandboxes (Figure 1.11). A guard zone is an unallocated memory region: a read, write or jump to that memory region will lead to a crash of the program. By checking that a register is inside the sandbox and that the offset is smaller than the size of the guard zone, the verifier can check that the actual value (value of register + offset) is either inside the sandbox, or inside the guard zone. Execution therefore either continues as expected, or aborts immediately. If the offset is bigger than the guard zone however, the address could be outside of the sandbox and guard zones, so the generator should reapply a sandboxing operation, or the verifier will reject the program as potentially unsafe.

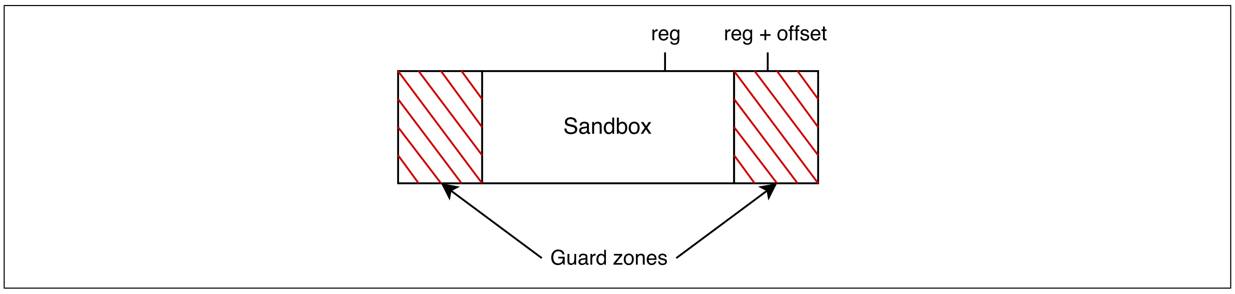


Figure 1.11 – Guard zones

1.4.5 Protecting the Control Flow Graph

In SFI, indirect jumps are required to point to aligned addresses in the sandbox. A more fine-grained technique is called Control Flow Integrity [1]. This technique inserts run-time checks that the computed jump follows a certain policy. In particular, this technique is able to discriminate between function start and the middle of a function. It is then able to block jumps that do not target the beginning of a function. With this technique, bundles are not required anymore, and their overhead can be spared. However, the threat model is different, as the control flow is given by the programmer. CFI requires trust on the programmer to provide adequate control flow information.

CFI works by indicating the beginning of functions by a four-byte constant, that cannot be found anywhere else in the code, except for the start of a function. Indirect jumps and calls are rewritten to execute a small routine that checks the presence of the constant. The routine itself must recompute the constant to prevent embedding it, outside the beginning of a function. Otherwise, somewhere inside the routing would be considered a safe place to call. If the routine finds the constant just before the jump target, it executes the jump, otherwise it prevents it.

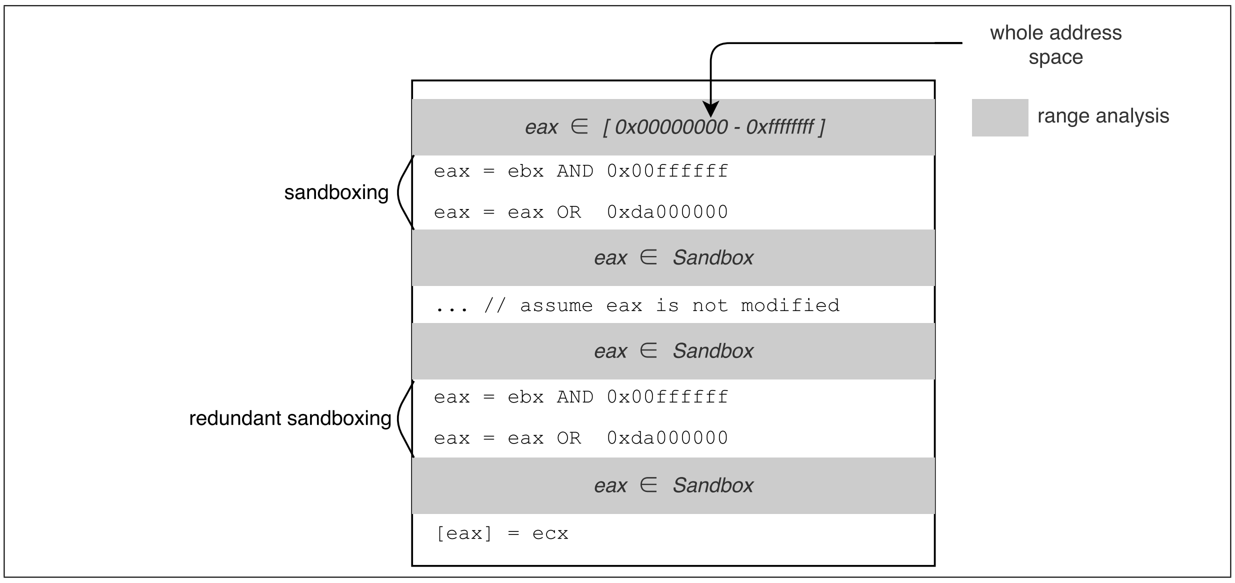
XFI [19] uses CFI to create a sandbox similar to SFI. In this implementation, data access is sandboxed exactly like in [41], while code is sandboxed using CFI. It assumes the code producer is not malicious and provides correct control flow information too. In general, this technique is more precise on what is allowed, but provides different guarantees than SFI.

1.4.6 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 has just been sandboxed will have its range equal to the sandbox memory area. These information allow two new optimizations presented in this section. These techinques are presented in Zeng et al [43].

Redundant Checks

This optimization takes place after the transformation of the code by the SFI techniques. It aims at removing sandboxing operations which are redundant and slow the untrusted module down. To detect such occurrences, a range analysis is performed for every register. An example can be seen in Figure 1.12, 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.



```
eax = ebx AND 0x00ffffff
eax = eax OR  0xda000000
... // assume eax is not modified
eax = ebx AND 0x00ffffff
eax = eax OR  0xda000000
[eax] = ecx
```

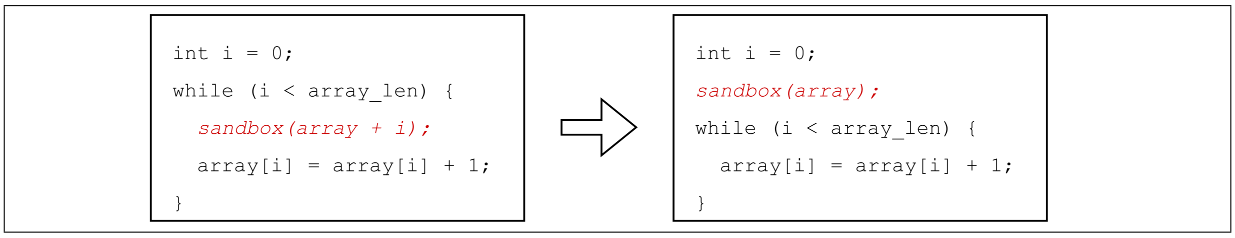
Figure 1.12 – Redundant check

Loop Check Hoisting

This optimization 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 1.13 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 1.4.4. 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 1.13.



```
int i = 0;
while (i < array_len) {
  sandbox(array + i);
  array[i] = array[i] + 1;
}
```

```
int i = 0;
sandbox(array);
while (i < array_len) {
  array[i] = array[i] + 1;
}
```

Figure 1.13 – Loop check hoisting

1.5 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 checks if the code transformation done previously are enough to guarantee the security property of SFI. Hence all executables accepted by the verifier are safe. However that also means that a safe program, which does not fulfill the criteria of the verifier will not be accepted either.

In SFI, most of the time, the verification consists in checking that every dangerous instruction is preceded by a sandboxing operation. To execute this step multiple solutions are available, RockSalt [30] for example directly uses regular expressions on binary files. But most implementations of verifiers start by disassembling the executables then check that the program follows the procedures of SFI. Unfortunately disassembling is a non computable problem for generic programs and is still widely searched, which is problematic for this verification step. Most of the time the verification of assembly code is much simpler than the disassembly phase so the amount of trusted code can generally be reduced to the disassembler.

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.

1.5.1 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 for being undecidable for arbitrary input program, but it is possible when we make some assumptions on the binary module that has to be disassembled:

— 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 1.3.1 help disassembly, because no instruction crosses a chunk boundary and control flow only targets the beginning of a chunk

If these assumptions are true, disassembly is reliable, and the verifier can do its job. If these assumptions are false, the disassembler or the verifier will notice and the module will be rejected. In the end, only reliably disassembled and checked binaries will be run, which means that only safe binaries will be run. Although this might sound simple, some researches have worked on making the verifier even more trustworthy thanks to formal methods.

1.5.2 Certified Verification

Proof assistants such as Coq or HOL are great tools to gain confidence that a given implementation actually does what it is meant to do. In ARMor [45] for instance, 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 [30] uses the Coq proof assistant to build a proved correct alternative verifier for NaCl sandboxes. 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.

Usually the TCB of SFI implementations is equal to the code of the verifier. However in ARMor and RockSalt both use formal methods in order to reduce the amount of code which needs to be trusted. Indeed the verifiers are both proved to be correct which means that if they deem that a binary is safe, it can not break the policy of SFI when loaded and executed. Therefore the TCB of these implementations can be summarized to the amount of trust one puts in their respective proof assistant: HOL and Coq.

1.5.3 Certified Sandboxing

Previously in 1.3.4 we talked about Psfi [25] which does all the SFI code transformation in a high-level language. This choice allows the sandboxing mechanism to have better portability since it becomes architecture independent, and better speed can be achieved because it benefits from the compiler optimizations. 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 optimization, so, even if the binary is secure, it is harder to be sure of it.

To solve these constraints, Psfi choses to use a certified compiler called CompCert [26]. CompCert was written using the Coq proof assistant and was proved 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 proved to meet this two criterias:

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 1.14. Starting with a source program, one only needs to compile it with SFI-CompCert to obtain a SFI-safe executable. We can notice in Figure 1.14 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.

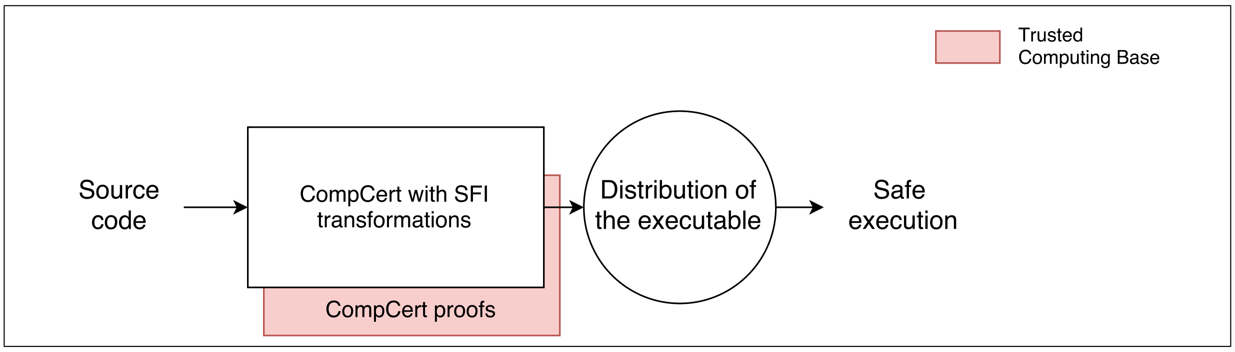

Trusted Computing Base
Source code
CompCert with SFI transformations
CompCert proofs
Distribution of the executable
Safe execution

Figure 1.14 – 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.

1.5.4 Static Analysis

Some optimizations we have presented, *redundant checks elimination* and *loop check hoisting*, have the peculiarity that they can remove unnecessary sandboxing checks in the code. This kind of optimizations however makes the structure of the untrusted module less clear. With the usual optimizations, the structure is preserved and the verification can still be a single linear pass on the code. Unless they are very local (e.g. in the same code bundle for x86), they cannot be checked with a single pass, since the verifier now needs to check control flow. It needs to do a more complex static analysis [43].

Chapter 2

Abstract Interpretation as a Tool for Analysis

As we have seen in the previous chapter, we want to verify a security property, that is a property on states of the program. This property is not trivial, so according to Rice's theorem, the security of a program is undecidable.

Even though we would like to decide whether every accessible state of a program is safe or not, which is impossible, we have to find another way to ensure we run only safe programs. In SFI, the verifier may reject perfectly safe modules, but it must not accept unsafe modules. In the same way, abstract interpretation [15] creates an over-approximation of the set of accessible states on which the safety is decidable. There are then three cases: either the program was insecure, in which case the over-approximation is still insecure and the program is rejected, either the program was secure but the over-approximation includes insecure states that are not reachable by the program, and it is rejected, or the program was secure and the over-approximation stays secure too, in which case the program is accepted.

In this thesis, we will show how to verify a more complex and complete security property than the standard SFI implementations, that better reflects the intuition when we talk about isolation. So I will be going the first route: constructing a decidable over-approximation on which it will be easier to check the security property. For that I will be using Abstract Interpretation, a framework that allows one to create such approximations programatically and to verify properties on them.

2.1 A Minimal Theory of Abstract Interpretation

2.1.1 Abstract Domains

Programs manipulate data such as integers, strings, or complex records. During a program analysis, we do not want to manipulate possible values one by one: we want to manipulate them in sets of possible values. A set of values is represented by an abstract value. For instance, to represent a set of integers, we can use the abstract set $\{positive, negative, any\}$. An abstract value is one of these three cases: positive represents the set of numbers from 0 to $+\infty$, negative from $-\infty$ to 0, and any represents any number. Or we can use the abstract set of every interval. In that case, an abstract value is an interval, which represents every value in the interval. The set of actual values manipulated by programs is called the concrete set and is noted D and values are called concrete values, while the set of abstract values is called the abstract set and is noted $D^\sharp$. We will always use the $\sharp$ notation for abstract objects, as opposed to no special notation for the corresponding concrete objects.

We do not want to choose any kind of abstract values, though: we want to represent sets of values. So, we define an operation, called the concretization and noted γ, that takes an abstract value and returns the set of values it abstracts. For instance, the sign abstraction will have the following concretizations:

$$\gamma(positive) = \{x \| x \geq 0\}$$

$$\gamma(any) = D$$

$$\gamma(negative) = \{x \| x \leq 0\}$$

Because programs manipulate data, we want our analysis to also manipulate them in the same way. For each manipulation of data (the computation of some result over variables), we want to have a matching manipulation of abstract data. In the concrete world, we compute a new value from a set of values. In the abstract world, we compute a set of possible values that result from the same computation over a set of abstract values. Let's take a simple example first: the computation $a + b$. In the concrete world, the operation $+$ is well defined. If a is equal to 5 and b to 7, the result is 12. Now in the abstract world, if a is in the range $[1; 5]$ and b is in the range $[4; 12]$, then the result must be in the range $[5; 17]$. And we can check that indeed, for any value a and b in these ranges, the result is between 5 and 17.

Operation Soundness

In the previous small example, we already get the intuition of the property that we will present next. We want the abstract result to represent any concrete computation that uses one of the possible values described by the abstract values of the initial variables. Mathematically, we write, for a concrete operation $\diamond$ and the corresponding abstract operator $\diamond^\sharp$ acting on the same number of arguments:

Definition 3 (Abstract Operation Soundness)**.** *An abstract operation $\diamond^\sharp$ is sound if and only if $\forall i \in [0;n], a_i \in \gamma(a_i^\sharp) \implies \diamond(a_0, \ldots, a_n) \in \gamma(\diamond^\sharp(a_0^\sharp, \ldots, a_n^\sharp))$.*

Following this principle, here are a few abstract operations on the interval domain:

$$[a;b] +^\sharp [c;d] = [a+c; b+d]$$
$$[a;b] -^\sharp [c;d] = [a-d; b-c]$$
$$[a;b] \times^\sharp [c;d] = [min(ac, ad, bc, bd); max(ac, ad, bc, bd)]$$

We can check that these definitions respect the property we want to have:

Lemma 1 (Abstract Addition Soundness)**.** $\forall a \in \gamma(a^\sharp), b \in \gamma(b^\sharp), a+b \in \gamma(a^\sharp +^\sharp b^\sharp)$.

Proof. $a^\sharp$ and $b^\sharp$ are intervals, so we can rewrite the first conditions on a and b as: $a \in [c;d]$ and $b \in [e;f]$. In particular, we have $a \geq c$ and $b \geq e$, so $a+b \geq c+e$ and similarly, we have $a \leq d$ and $b \leq f$, so $a+b \leq d+f$. So we have $c+e \leq a+b \leq d+f$ which can be rewritten as $a+b \in [c+e; d+f] = a^\sharp +^\sharp b^\sharp$. $\square$

Partial Order

Abstract domains are partially ordered sets. That means that we need to equip our domain with a comparison operation, which is noted $\sqsubseteq$. Two elements do not always need to be comparable (that's why it's only a partial order), but the order must have the following three properties:

Definition 4 (Partially Ordered Set)**.** *$\sqsubseteq$ is a partial order on $D^\sharp$ if and only if it is:*

— *reflexive:* $\forall a \in D^\sharp, a \sqsubseteq a$

— ***antisymmetric:*** $\forall(a,b) \in D^{\sharp 2}, a \sqsubseteq b \wedge b \sqsubseteq a \implies a = b$

— ***transitive:*** $\forall(a,b,c) \in D^{\sharp 3}, a \sqsubseteq b \wedge b \sqsubseteq c \implies a \sqsubseteq c$

We can now state a first property related to order and concretization:

Definition 5 (Partial Order Soundness). $\sqsubseteq$ *is a sound partial order if and only if* $\forall(a^\sharp, b^\sharp) \in D^{\sharp 2}, a^\sharp \sqsubseteq b^\sharp \implies \gamma(a^\sharp) \subseteq \gamma(b^\sharp)$*.*

We also equip the abstract domain with a union operation, noted $\sqcup$. The union is the smallest set that is bigger than both elements of the union.

Definition 6 (Union). $a \sqcup b = c \equiv a \sqsubseteq c \wedge b \sqsubseteq c \wedge (\forall d, (a \sqsubseteq d \wedge b \sqsubseteq d) \implies c \sqsubseteq d)$*.*

2.1.2 Abstract State

In this minimal theory, we model programs as a transition system between states: a program starts with an initial state and transitions from state to state. For nondeterministic programs, a state can have more than one following state. The program may block when there is no possible next step. From this definition of a program, we take the set of reachable states to be the closure of all possible states, reachable from the initial state. Formally, if we use $s \to s'$ to represent a possible transition from state s to state s', and i the initial state. The set of reachable states, $Reachable$, is defined as:

$$\frac{}{i \in Reachable} \qquad \frac{s \in Reachable \quad s \to s'}{s' \in Reachable}$$

The previous notation is an inference system: if you can prove what is over the horizontal bar, then you are allowed to conclude what is below the horizontal bar. In the previous system, we can always conclude that the initial state is reachable, because there is no condition, or premise. The second rule says that, if we can conclude that s is a reachable state, and it transitions to another state s', then that other state is also reachable.

We also suppose that our program is a set of instructions, that are executed one after the other. The execution of an instruction is exactly what is represented by the transition: the instruction is executed in a state, modifies the state, and the state after the execution is the new state after the transition. Each instruction is associated with a location, and the location of the next instruction to execute is stored in the state.

The program is allowed to modify the next instruction location (for instance to call code elsewhere, to jump around in a conditional or in a loop).

The set of reachable states at a specific location is simply the subset of the reachable states whose location is equal to that specific location.

Property

We also have a property P on states, and the goal of the analysis is to decide whether the property holds on every concrete reachable state. Here, we introduce $P^\sharp$, an abstract property on abstract states. The following property states that, if the abstract property holds on an abstract state, the concrete property holds on its concretization.

Definition 7 (Abstract Property Soundness)**.** *The abstract property $P^\sharp$ is sound if and only if $P^\sharp(s^\sharp) \implies \forall s \in \gamma(s^\sharp), P(s)$.*

The result of our analysis will be an over-approximation of the set of reachable states for each location of the program, by means of an abstract state. Each location is associated with an abstract state, that represents a set of reachable states at this location. The abstract state domain is also associated with a transition system, such that:

Property 1 (Abstraction Soundness)**.**
$P^\sharp(s^\sharp) \implies s \in \gamma(s^\sharp) \implies s \to s' \implies \exists s'^\sharp, s^\sharp \to^\sharp s'^\sharp \wedge s' \in \gamma(s'^\sharp)$.

The first condition is unusual: proper abstract interpretation does not require it. In fact, we can see the transition as the application of a function, and without the initial condition, this soundness property is exactly an operation soundness property. The reason why we need the first condition here is that our analysis might not be able to properly analyze a program after it reached an abstract state on which the property does not hold. After reaching such a state, the analysis terminates without necessarily returning an over-approximation of the reachable states. However, this property states that, if the property holds on the result of the analysis, then the result is indeed an over-approximation of the concrete reachable states.

Now, we can define the analysis. It is an iteration of the same steps: for every location associated with an abstract state, compute the set of next abstract states. Merge these new states with the states that have already been computed before, and start again if anything changed. If nothing changed, we stop the analysis and return that result. Although we can think of the merging step as a simple union of the old state and the newly computed one(s), it is not always practical, since it may lead to the analysis running forever. We will for now use the union anyway, since it is sound and simpler to understand. However, in reality we use a *widening* operation, that we introduce later in this chapter.

Example

To better understand this, let us consider the simple example in Figure 2.1, in a pseudo-code syntax, close to C.

Sound Analysis

All we are left with now, is to prove that this process terminates and that it indeed computes an over-approximation of the reachable states. As we defined it before, using the union of the old and new states, the analysis does not always terminate: if the loop in the example was infinite (with an always true condition), there would have been an infinite number of steps: in location 2, the abstract state would successively be computed as $[0; n]$, $[0; n+1]$, etc. Since at every step, the abstract state grows, there needs to be another step, and that new step continues to change the abstract state, forever. In order to solve that issue, and get a terminating analysis, we introduce the widening operator, noted $\triangledown$. It has the following properties:

Definition 8 (Widening)**.** *An operator on an abstract domain $D^\sharp$ is a widening if and only if it respects the two following conditions:*

— $a \sqcup b \sqsubseteq a \triangledown b$

— $\forall (y_0, \ldots, y_n)$*, we define* $x_0 = \bot$ *and* $x_{n+1} = y_n \triangledown x_n$*. Then* (x_n) *is ultimately stationary, i.e.* $\exists n, \forall m \geq n, x_n = x_m$.

With this widening operation, we can now define the actual algorithm: it is the same algorithm as before, except that we use the widening operator in the merging step instead of the union. Now, we are able to prove the termination of the algorithm.

Theorem 1 (Termination)**.** *The analysis terminates.*

```
// ∅
1: int a = 0;
   while(a < 10) {
2:       a++;
3:       nop;
   }
4: return a;
```

(a) There are only 4 instructions to analyze. At first, we have no abstract state, except for the initial one, which is empty.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;0]
2:       a++;
3:       nop;
   }
4: return a;
```

(b) During the first step of our analysis, we see that there is only one abstract transition from the initial state to the state 2, because the condition is always satisfied.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;0]
2:       a++;
// a ∈ [1;1]
3:       nop;
   }
4: return a;
```

(c) On the next step, the initial state transitions again to that same state, and the state we just introduced can only flow to 3 while executing the instruction in 2.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;0] ⊔ [1;1] = [0;1]
2:       a++;
// a ∈ [1;1]
3:       nop;
   }
4: return a;
```

(d) Then, again, but this time there is a transition from 3 back to 2 so we need to merge the state we knew about and the newly computed state.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;1].
2:       a++;
// a ∈ [1;1] ⊔ [1;2] = [1;2]
3:       nop;
   }
4: return a;
```

(e) We again continue: this time only the newly merged states give new abstract states.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;9]
2:       a++;
// a ∈ [1;9] ⊔ [1;10] = [1;10]
3:       nop;
   }
4: return a;
```

(f) After doing this a few times, we end up in this state.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;9] ⊔ [1;9] = [0;9]
2:       a++;
// a ∈ [1;10]
3:       nop;
   }
// a ∈ [10;10]
4: return a;
```

(g) Only the abstract state at 3 has an interesting behavior: its next state is either 2 because the value of a might be below 10, or 4 because the value might be bigger or equal to 10.

```
1: int a = 0;
   while(a < 10) {
// a ∈ [0;9] ⊔ [1;9] = [0;9]
2:       a++;
// a ∈ [1;10]
3:       nop;
   }
// a ∈ [10;10]
4: return a;
```

(h) There is no new computation step: if we compute the next abstract states for each location, they are already included in the abstract state we already know. So, the computation ends here, and we have some information about the possible reachable states.

Figure 2.1 – Abstract Interpretation Example

Proof. Let's call each step of the analysis x_n. Then, the initial step of the analysis is $x_0 = \bot$. Then, at each step, we compute the set of next states, that we are going to call y_n. Then, we use the widening operator to compute the next step of the analysis, $x_{n+1} = x_n \nabla y_n$. With this notation, it is easy to see that we are exactly in the configuration of the definition above, so we can conclude that x_n is ultimately stationary, and that is our halting condition. □

Finally, we can prove the soundness of the analysis. Here again, the first part of the theorem is not usual: this time, it is simply a consequence of the change in the abstraction soundness property.

Theorem 2 (Soundness)**.**
$\forall s^\sharp \in \mathit{Reachable}^\sharp, P^\sharp(s^\sharp) \implies \forall s \in \mathit{Reachable}, \exists s^\sharp \in \mathit{Reachable}^\sharp, s \in \gamma(s^\sharp).$

Proof. This proof is by induction on s. There are two cases:

— First, if s is the initial state, we have by definition both $i \in \gamma(i^\sharp)$ and $i^\sharp \in \mathit{Reachable}^\sharp$.

— Second, we suppose by induction that we have $s \to s'$, $s \in \gamma(s^\sharp)$ and $s^\sharp \in \mathit{Reachable}^\sharp$. Because of the property on the transition system we have presented at the beginning of this section, we can conclude on the existence of $s'^\sharp$ such that $s^\sharp \to s'^\sharp$ and $s' \in \gamma(s'^\sharp)$. Because of the way the analysis works, this new abstract state must be smaller (and comparable) than the one that was computed by the analysis: otherwise the widening would have been taken and the analysis would not have stopped yet. s' is therefore also in the concretization of the bigger state that was computed by the analysis.

□

2.2 A Methodology for Showing Property Preservation

Because of the previous soundness theorem, if the abstract property is true on the result of the analysis, the soundness properties ensure that the concrete property P is also true on every reachable state. On the other hand, if the property does not hold on the result of the analysis, we cannot conclude: since the concretization of the result is bigger than the set of reachable states, maybe the concrete property does not hold only on some unreachable states (a false alarm), or maybe it does not hold on some reachable states (the program is not safe).

For our purposes, an abstract interpretation analysis is therefore composed of a concrete state domain, an abstract state domain, a concrete property, an abstract property and a concretization. Additionnaly, they have four soundness properties:

— Sound initial state: $i \in \gamma(i^\sharp)$

— Sound order: $a^\sharp \sqsubseteq^\sharp b^\sharp \implies \gamma(a^\sharp) \subseteq \gamma(b^\sharp)$

— Sound abstraction: $P^\sharp(s^\sharp) \implies s \in \gamma(s^\sharp) \wedge s \to s' \implies \exists s'^\sharp, s^\sharp \to^\sharp s'^\sharp \wedge s' \in \gamma(s'^\sharp)$

— Sound property: $\forall s^\sharp, P^\sharp(s^\sharp) \implies \forall s \in \gamma(s^\sharp), P(s)$

Concretization Compositionality

Suppose we have an intermediate domain, $D^\natural$. Instead of using a direct concretization, we define a concretization from the abstract to the intermediate domain, γ_1 and from the intermediate domain to the concrete domain, γ_2. The concretization from the abstract domain to the concrete domain is now defined as:

$$\gamma(a) = \bigcup_{x \in \gamma_1(a)} \gamma_2(x)$$

With this definition, we also have the following lemma, which we will use in the proofs of the next four theorems:

Lemma 2. *$s \in \gamma(s^\sharp) \implies \exists s^\natural, s \in \gamma_2(s^\natural) \wedge s^\natural \in \gamma_1(s^\sharp)$.*

If we suppose we have the four soundness properties on the two pairs of domains (and respective properties), we have the following four compositionality theorems:

Theorem 3 (Initial State Soundness Compositionality)**.**
$i \in \gamma(i^\sharp)$.

Proof. By initial state soundness between the three domains, we have $i^\natural \in \gamma_1(i^\sharp)$ and $i \in \gamma_2(i^\natural)$. We can easily conclude by using the definition of γ: $\gamma(i^\sharp)$ is a union of at least $\gamma_2(i^\natural)$, which contains i. □

Theorem 4 (Order Soundness Compositionality)**.**
$a^\sharp \sqsubseteq b^\sharp \implies \gamma(a^\sharp) \subseteq \gamma(b^\sharp)$.

Proof. By order soundness between the abstract and intermediate domains, we have $\gamma_1(a^\sharp) \subseteq \gamma_1(b^\sharp)$. A union on $\gamma_1(a^\sharp)$ is therefore necessarily smaller than a union on $\gamma_1(b^\sharp)$, and since γ is exactly this kind of union, we can conclude. □

Theorem 5 (Property Soundness Compositionality)**.**
$\forall s^\sharp, P^\sharp(s^\sharp) \implies \forall s \in \gamma(s^\sharp), P(s).$

Proof. By Lemma 2, we can take $s^\natural$ such that $s^\natural \in \gamma_1(s^\sharp)$ and $s \in \gamma_2(s^\natural)$. By property soundness between the abstract and intermediate domains, we have $P^\natural(s^\natural)$, and by property soundness between the intermediate and concrete domains, the property holds on s. □

Theorem 6 (Abstraction Soundness Compositionality)**.**
$P^\sharp(s^\sharp) \implies s \in \gamma(s^\sharp) \implies s \to s' \implies \exists s'^\sharp, s^\sharp \to^\sharp s'^\sharp \wedge s' \in \gamma(s'^\sharp).$

Proof. By Lemma 2, we can take $s^\natural$ such that $s^\natural \in \gamma_1(s^\sharp)$ and $s \in \gamma_2(s^\natural)$. By property soundness, we have: $P^\natural(s^\natural)$. By abstraction soundness between the intermediate and concrete domains, we have: $\exists s'^\natural, s^\natural \to^\natural s'^\natural$ and $s' \in \gamma_2(s'^\natural)$. We can also take such an $s'^\natural$. The first element of this property and the second element of the application of Lemma 2 ($s^\natural \to^\natural s'^\natural$ and $s \in \gamma_2(s^\natural)$) are the conditions of the abstract soundness between the abstract and intermediate domains. Therefore, we have $s'^\sharp$, such that $s^\sharp \to^\sharp s'^\sharp$ and $s'^\natural \in \gamma_1(s'^\sharp)$. The first expression is the first part of the property we want to prove, while the second expression ($s'^\natural \in \gamma_1(s'^\sharp)$) can be used with $s' \in \gamma_2(s'^\natural)$ to get the second part of the property we wanted to prove. □

These compositionality theorems can then be applied recursively as many times as we want, to have as many intermediate steps in the soundness proofs. For each intermediate step, we will prove the four properties.

In Chapter 4, we will introduce the security property on the concrete domain. Then, we will prove the soundness properties on more and more abstract semantics, until we reach the abstract semantics that we are going to use in the implementation shown in Chapter 5. The verifier implements the abstract interpretation and, if the abstract property holds, it accepts the untrusted module and allows it to run. If the abstract does not hold however, the verifier will reject the untrusted module and will not allow it to run. In any case, the verifier will never allow an untrusted module that does not respect the concrete security property to run.

PART II

Modular Software Fault Isolation as Abstract Interpretation

CHAPTER 3

AN INTERMEDIATE LANGUAGE TO REPRESENT ASSEMBLY

3.1 A Semantics for Running SFI Modules

We have just defined a methodology for proving semantic property preservation between two semantics, by abstracting further and further a concrete semantics, and using abstract interpretation to prove a property holds on the concrete semantics, by analysing the most abstract semantics. In this second part of this thesis, we will first introduce the language we are going to use for our analysis, its concrete semantics and its security property. This is what this chapter is about. In the following chapter, we will introduce the different intermediate semantics and show that at every step, the four properties of concretization we defined in the previous section are respected. The last chapter of this second part is dedicated to the actual implementation of a verifier that uses this technique.

For the purpose of simplicity, we are not going to use the full semantics of an actual processor, because it would require us to explain hundreds of different rules. Instead, we introduce an intermediate language that is a minimal assembly language that implements all relevant features of these languages: the language has a global memory, registers, temporaries, computations, function management and halting.

Here is the complete syntax of the language:

e	$::=$	r	a register
		$\mid n$	an immediate value
		$\mid e_1 \bowtie e_2$	arithmetic operations
i	$::=$	$r = e$	register affectation (or STR: Store To Register)
		$\mid [e_1] = e_2$	write to memory (or STM: Store To Memory)
		$\mid r = [e]_n$	read from memory (or LDM: Load Data from Memory)
		$\mid \mathbf{jmpif}\ e_1\ e_2$	conditional jump
		$\mid \mathbf{call}\ e$	function call
		$\mid \mathbf{ret}\ e$	function return
		$\mid \mathbf{hlt}$	halt

The language features expressions e made of registers r, numeric constants n and binary operators $\bowtie$. Binary operators range over typical arithmetic operators e.g. $+$, $\times$, bitwise operators e.g. `xor` and logical operators e.g. $<$. An instruction i consists of assignments of an expression to a register ($r = e$) or from a memory location e of size n bytes ($r = [e]_n$); storing in memory the value e_2 at the address e_1 ($[e_1] = e_2$). A conditional jump $\mathbf{jmpif} e_1, e_2$ jumps to the computed address e_2 if the condition holds ($e_1 \neq 0$). The instruction call is equivalent to the computed jump $\mathbf{jmpif} 1 e$ but identifies a function call ; $\mathbf{ret} e$ is also equivalent to a computed jump but identifies a function return. The instruction $\mathbf{halt}$ immediately stops the program.

The language manipulates bitvectors: a value has a size in bits. We will use the notation T_* to denote a list of elements of type T. We use this notation in $\mathbb{B}_*$ to mean a list of booleans, and that is the type for values. The notation $\mathbb{B}_n$ denotes the set of n-bits values. A register is not typed, in the sense that it is not limited to a specific size (it contains a value of type $\mathbb{B}_*$), so for instance a register that holds a 32-bits value can later hold a 1-bit value or a 64-bits value. Additionally, registers are present in unlimited numbers, so they can be used to model actual registers of a machine, and for representing temporary values (intermediate results of an instruction). In the rest of this thesis, we suppose our programs are "well-typed", that is, no size discrepancies can happen: computations and operations always act on operands and expressions of appropriate size.

The operational semantics operates over a state $\langle \rho, \mu, \iota \rangle$ where ρ is an environment ($Env = Reg \to \mathbb{B}_*$), μ is the whole memory of the process and ι is the current instruction pointer. Memory is divided into regions that are granted different access rights among read and write. The two predicates, respectively $Writable(a, n)$ and $Readable(a, n)$, hold when a is an address in a region that is granted write access, respectively read access, and there is at least n bytes available in that region after a.

The semantics is fairly standard and is given below. First, the ISTR rule shows that a register assignation updates the register in the environment with its new value. This first rule also uses a few notations. First, we note ι^+ for the address of the following instruction. We also note $instr(\iota)$ for the instruction at address ι. This function is partial and may not always return anything. $\lfloor i \rfloor$ is our notation for when the partial function is defined and returns a value, here an instruction. We also note environment updates with the syntax $\rho[r \mapsto v]$ for the same environment as ρ, except r is now associated with the value v. The same notation is used for updating the memory. $[\![e]\!]_\rho$ is our notation for evaluating the expression e in the environment ρ. Remember that an expression does not use the memory at all.

$$\text{ISTR}\frac{instr(\iota) = \lfloor r = e \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho[r \mapsto [\![e]\!]_\rho], \mu, \iota^+ \rangle}$$

The memory write can only happen when a write is allowed, and updates the global memory with the computed value.

$$\text{ISTM}\frac{instr(\iota) = \lfloor [m] = e \rfloor \quad Writable([\![m]\!]_\rho, size(e))}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu[[\![m]\!]_\rho \mapsto [\![e]\!]_\rho], \iota^+ \rangle}$$

The memory read can only happen when a read is allowed, and updates the environment with the value that is being read. We use the function notation $\mu(a, s)$ for a read at address a of size s bytes in the memory μ.

$$\text{ILDM}\frac{instr(\iota) = \lfloor r = [m]_n \rfloor \quad Readable([\![m]\!]_\rho, n)}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho[r \mapsto \mu([\![m]\!]_\rho, n)], \mu, \iota^+ \rangle}$$

The conditional jump can either not jump, and continue to the rest of the program if the condition does not hold (i.e. , is 0), or jump to the computed address.

$$\text{IJCCNO}\frac{instr(\iota) = \lfloor \mathbf{jmpif}\ cond\ addr \rfloor \quad [\![cond]\!]_\rho = 0}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, \iota^+ \rangle}$$

$$\text{IJCC}\frac{instr(\iota) = \lfloor \mathbf{jmpif}\ cond\ addr \rfloor \quad [\![cond]\!]_\rho \neq 0)}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, [\![addr]\!]_\rho \rangle}$$

Calls and returns are treated no different from a conditional jump whose condition is verified. In particular, they do not update the stack as the same-named instructions would on `x86` for instance. In fact, the the equivalent of a call in `x86` is composed of multiple instructions that make explicit the effects of the instruction. This is also the case for a `ret`.

$$\text{ICALL}\ \frac{instr(\iota) = \lfloor \mathbf{call}\ addr \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, [\![addr]\!]_\rho \rangle}$$

$$\text{IRET}\ \frac{instr(\iota) = \lfloor \mathbf{ret}\ addr \rfloor}{\langle \rho, \mu, \iota \rangle \longrightarrow \langle \rho, \mu, [\![addr]\!]_\rho \rangle}$$

The following shows what these two instructions can be translated to in our language to have the same effects:

Listing 3.1 – Intermediate Representation of an X86 Call

```
0x40800000:
  # We record the address of the
  # following instruction
  [esp] := 0x40800005
  # and we push it to the stack
  esp := esp + 0x00000004
  # then, we jump to the callee
  call 0x40123456
```

Listing 3.2 – Intermediate Representation of an X86 Ret

```
0x408123456:
  # We pop the return address from
  # the stack
  esp := esp - 0x000004
  # and record it in a temporary
  tmp := [esp]
  # then, we jump to it
  ret tmp
```

3.2 Execution Model

Our small language is supposed to model the execution of an actual binary on an actual hardware. In this part of the thesis, we only consider a program running on a single processor, with a single core and no thread. Before its execution, the host program (including the trusted library) is loaded with the untrusted module. The host program and the untrusted module are both composed of a code section, containing the code to be executed, and a data section, containing initial values for some memory regions that are distinct from the memory region that contains the code. We assume the memory region described by the untrusted module is the sandbox. In fact, if it is not the case, it is simple to detect and reject the module before doing any analysis of the code itself.

The initial state is in the host program, which is responsible for creating a first stack frame and calling the untrusted module. The initial state also has an arbitrary initial environment and an arbitrary memory, except for memory regions that were loaded from the host program or the untrusted module. The untrusted module then executes and modifies memory according to its instructions. However, we do not model self-modifying programs, as we assume the code is immutable. The module can either return to the trusted library, or call functions from the trusted library. Once control has returned to the trusted library, it is again allowed to call an untrusted function, or return to the untrusted module if it was called from there.

CHAPTER 4

ANALYZING A SINGLE-THREADED MODULE

We have seen in chapter 1 that most SFI implementations consider a memory region, called the sandbox, outside of which the untrusted module is not allowed to read or write. However, our execution model imposes another memory region with which the module has to interact: the stack. The stack needs to be written to by functions in order to register e.g. return addresses when calling other functions, local variables, etc. This way of doing software fault isolation creates new challenges that this thesis attempts to address.

The back and forth between the trusted library and the untrusted module creates an alternating pattern of trusted and untrusted stack frames: we cannot allow the untrusted module to interfere with the stack frames of the trusted library, but we still need to allow it to access and modify its own stack frames. Usually, SFI is a method for doing static verification of isolation in a memory region. In the case of the stack however, there are many memory regions. The dynamic nature of these regions also creates a new challenge for verification, as a simple arithmetic and logical operation cannot guarantee that an access is in a stack frame at runtime.

Additionally, we must ensure the untrusted module cannot create a stack overflow, which would potentially interfere with the rest of the memory (including the private memory of the trusted library).

4.1 Security Property as a Defensive Semantics

We define SFI and its sandboxing property operationally, as a *defensive semantics*, which means a standard semantics for the language, which also includes a series of additional (dynamic) verifications. These dynamic checks express what it means for code to be properly sandboxed, and what a properly sandboxed program can execute in this semantics. In particular, since the semantics is blocked when a check is not verified, the sandboxing property (i.e. the security property) is defined as a simple *progress* property, that is, a program is secure if its reachable states can progress.

It is however not enough to add conditions to the previous semantics because of undefined behaviors. A reachable state may not progress for two reasons: either the program does not respect the security property, or it is trying to do something that is undefined, such as a division by zero, halting or trying to access a memory region that is not associated with the corresponding right.

The language is designed to prevent undefined behaviors such as divisions by zero, because the evaluation of an expression is a total function. A division by zero is defined, first because some hardware define it, and second because for those who do not, the division instruction can be modeled by a code that first checks the denominator.

We do not wish to model the content of the trusted library, nor do we want to explicit the rights on any memory region. The only exceptions are for the sandbox and the stack which must be readable and writable, as well as the two regions around the stack that are called “guard zones” and are associated to no rights at all.

We use an explicit $\blacksquare$ state for situations where the program tries to access the guard zones and where the program halts, to prevent these situations from blocking the semantics progress. This ensures that the semantics blocks only when the module would have executed an instruction that violates the security.

Figure 4.1 shows what rights are associated to what in the sandbox (in white) and the guard zones (in light gray). bp is the current base pointer. Above bp is the backtrace of the current function, below bp is the current stack frame. The untrusted module is allowed to read anywhere in the stack (so it can read arguments passed through the stack and dereference pointers to the stack), but it is allowed to write only in its own stack frame. Writing to parents' stack frames is forbidden and a security violation, so the semantics must reflect this. Calling a function (of the trusted library or the untrusted module) sets a new bp which is allowed to be any point below the current base pointer, and is either in the stack or in the guard zone at the bottom.

The reason for introducing a guard zone and these specific security rules for stack management is not immediately obvious. Remember however that during the analysis, the actual position of the base pointer is unknown. We must ensure that the initial assumptions about the possible values of the base pointer are always true after a function

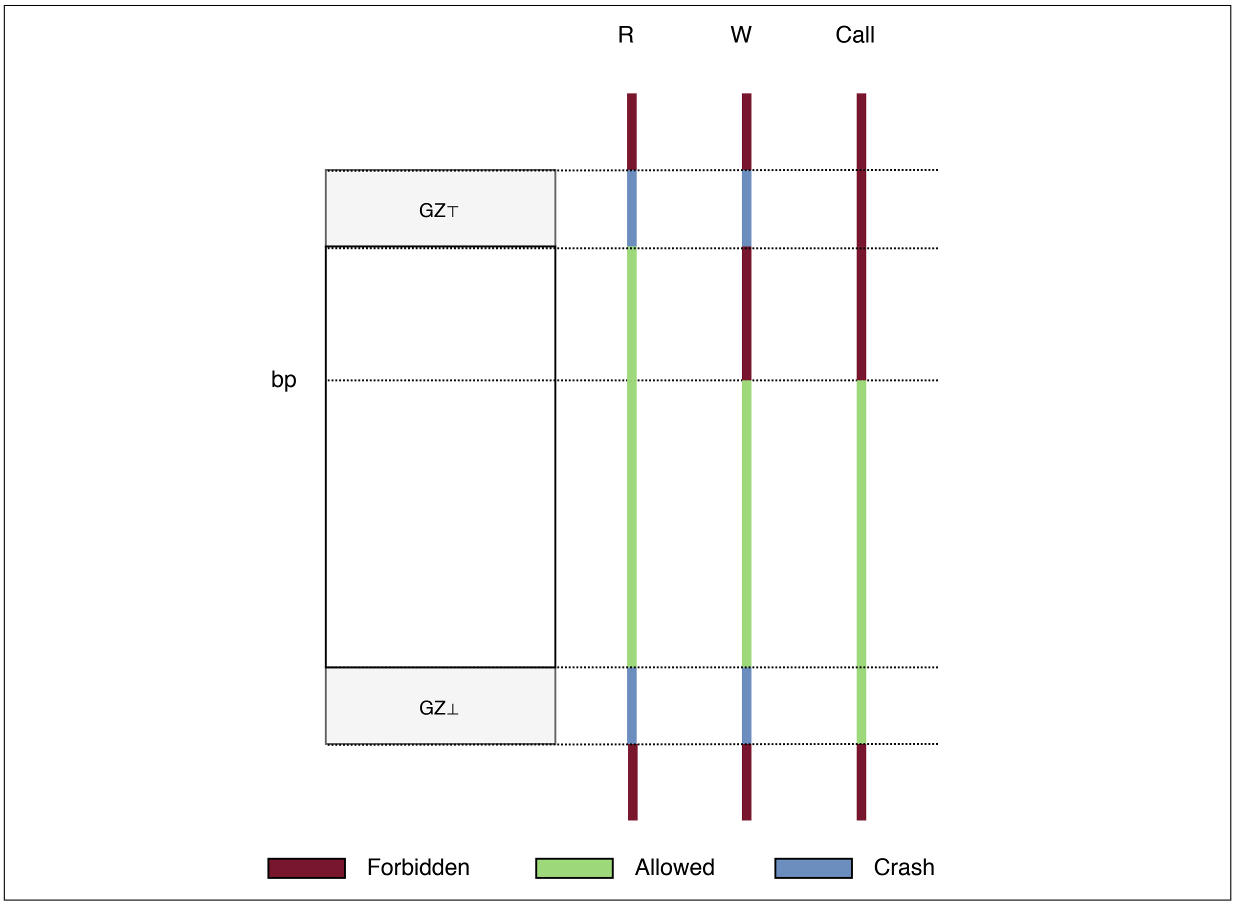


Figure 4.1 – Allowed and Forbidden Actions in the Stack

call, otherwise we would accept modules for which some behaviors were not analyzed. Using a guard zone of sufficient size ensures that, if the stack starts to overflow, it is detected by the program crashing. The actual implementation of this mechanism is described in section 4.3

This technique is not new and used already as a cooperation between the kernel, the C library and the compiler. At load time, a guard zone is inserted between the stack and the heap. When compiled with gcc's `-fstack-check` option, programs will attempt a write every page from the stack pointer when the stack grows, ensuring that if the stack grows beyond the guard zone, a write to the guard zone happens and creates a segmentation fault. We will see our solution is slightly different: we only allow a stack frame to grow as much as a limited size, and instead of checking at every stack allocation, we only impose that a check happens before any function call (which is part of the normal workflow for most functions under most architecture), which creates no runtime penalty, as no specific code is needed.

We also assume given a set $\mathcal{T} \subseteq \mathit{Trust}$ of trusted functions that form the only authorized entry points of the trusted library, that the untrusted module can call. We assume a mechanism to get a set $\mathcal{F} \subseteq \mathit{Code}$ of function entry points. This set could be obtained by metadata given by the programmer, or by reading debugging information, or even by a previous static analysis pass. Since the mechanism to get the set of entry points is not trusted, the analysis will have to verify that functions that are called are always inside this set. Otherwise, the program may execute functions that were not analysed.

4.1.1 Defensive Semantics

With these assumptions, we can now define the defensive semantics, whose progress property defines the security property.

Since we do not want to model all memory, the defensive semantics will only model the relevant parts of the memory for the untrusted module: the sandbox and the stack. Instead of a single memory block that covers all of the available memory, we use two blocks that cover only a limited portion of the memory.

Definition 9 (Defensive State)**.** *A state is defined as* $\langle \rho, \delta, \phi, \iota \rangle \in \langle \mathit{Reg} \to \mathbb{B}_*, \mathbb{B}_{ptr} \hookrightarrow \mathbb{B}_1, \mathbb{B}_{ptr} \hookrightarrow \mathbb{B}_1, \mathbb{B}_{ptr} \rangle$ *and is made of an environment* ρ *and an address* ι *as with the concrete semantics, as well as two memory blocks: the sandbox* δ *and a stack frame* ϕ*.*

This stack frame corresponds to memory from the base pointer of the current function to the bottom of the stack and includes the guard zone at the bottom. As in the standard semantics, we use the function notation $\phi(addr, n)$ for a memory read at address $addr$ of size n and since we always have a condition to check the read is inside the memory region in the semantics, we will not bother with the fact that the memory is a partial function.

The reason why we record only a stack frame in the state is because every semantic derivation occurs in an inter-procedural context.

Definition 10 (Defensive Context)**.** *A context is defined as* $\langle cs, bp, \rho_i \rangle \in \langle (\mathit{IMMptr} \times \mathbb{B}_{ptr} \hookrightarrow \mathbb{B}_1)_*, \mathbb{B}_{ptr}, \mathit{Reg} \to \mathit{IMMB} \rangle$*, where* cs *is a call stack, a list where each element is composed of the base pointer of the function and the stack of the function, from its* *base pointer to the following base pointer (current or of the following function in the stack).* bp *is the current base pointer and* ρ_i *is the environment as it was when the current function was called.*

We will use the notation $::$ for the concatenation of an element to a list: $\langle bp, \phi \rangle :: cs$ is the list whose first element is the tuple $\langle bp, \phi \rangle$ and whose tail is cs. By an abuse of notation, we will also write $cs :: \langle bp, \phi \rangle$ for a list composed of the elements of cs to which we add the tuple at the end.

From both the intra-procedural state and the inter-procedural context, we can read memory from anywhere in the stack, but it also ensures that we cannot overwrite a value in the call stack.

Definition 11 (Memory Read)**.** *Reading a value in the stack is done with the following derivations: under a call stack, reading a value at address* $a \in \mathbb{B}_{ptr}$ *of size* n *returns a value. Either the value is in the current stack frame (rule* READFRAME*) or it is in another frame (recursive rule* READSTACK*).*

$$\textit{ReadFrame}\frac{bp- \mid \phi \mid \leq a \leq bp}{cs :: \langle bp, \phi \rangle \vdash_{a,n} \phi(a, n)} \qquad \textit{ReadStack}\frac{cs \vdash_{a,n} v}{cs :: \langle bp, \phi \rangle \vdash_{a,n} v}$$

A defensive judgment is of the form $\Gamma \vdash s \longrightarrow s'$ where Γ is an inter-procedural context and s, s' are either intra-procedural states or the crash state $\blacksquare$.

The ASSIGN rule assigns a new value to a register. This is always possible without violating the SFI property and no extra check is needed.

$$\text{ASSIGN}\frac{instr(\iota) = \lfloor r = e \rfloor}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto [\![e]\!]_\rho], \delta, \phi, \iota^+ \rangle}$$

The rule STOREDATA describes the execution of the statement $[e_1] = e_2$ for the case where e_1 evaluates to a memory address within the sandbox. The value of e_1 is computed and the start address of the data segment (the sandbox) d_0 is subtracted from it to obtain an offset o into the data segment. It is then verified that this offset is indeed smaller than the size of the data segment. If this verification succeeds, the location at offset o in the sandbox is updated with the value of e_2.

$$\text{STOREDATA}\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = [\![e_1]\!]_\rho - d_0 \quad 0 \leq o \leq d_s - \mid e_2 \mid}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho, \delta[o \mapsto [\![e_2]\!]_\rho], \phi, \iota^+ \rangle}$$

The rule STOREFRAME similarly makes the checks necessary for storing securely into the run-time stack. Here, the value of e_1 is supposed to be a valid reference into the current stack which starts at the address designated by the base pointer bp. Because the stack grows towards smaller addresses, the relative offset o into the current stack frame is computed as $bp - [\![e_1]\!]_\rho$. In order for the store to proceed normally, this offset must point into that part of the stack frame that is *not* making up the guard zone ($0 \leq o <\mid \phi \mid - \mathsf{GZ}_\perp$). It is also checked that the offset does not point into the guard zone at the beginning of the stack segment ($bp - o \leq s_0 - \mathsf{GZ}_\top$).

$$\textsc{StoreFrame}\ \frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = bp - [\![e_1]\!]_\rho \\ 0 \leq o \leq \mid \phi \mid - \mathsf{GZ}_\perp - \mid e_2 \mid \quad bp - o \leq s_0 - \mathsf{GZ}_\top}{\langle cs, bp, R \rangle \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho, \delta, \phi[o \mapsto [\![e_2]\!]_\rho], 1 \rangle \iota^+}$$

The rule LDSTCRASH describes what happens on an attempt to write into or read from one of the guard zones. In that case, the program transits to the crash state ■ and stays there forever due to rule CRASH.

$$\textsc{LdStCrash}\ \frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \vee instr(\iota) = \lfloor [r = [e_1]_n \rfloor \quad [\![e_1]\!]_\rho = a \\ (s_0 - \mathsf{GZ}_\top < a \leq s_0) \vee (bp - \mid \phi \mid < a < bp - \mid \phi \mid + \mathsf{GZ}_\perp + n) \quad n = \mid e_2 \mid}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \blacksquare}$$

The two rules LOADDATA and LOADSTACK describe how data are read from the data segment and the run-time stack. Reading from the data segment uses verification similar to storing into it. Loading from the stack is, however, slightly different in that our version of SFI allows reads from all of the stack frames, and not just the current frame. This allows e.g. functions to read their arguments. It is still verified that the access does not fall in the guard zones, using checks similar to STOREFRAME.

$$\textsc{LoadData}\ \frac{instr(\iota) = \lfloor r = [e]_n \rfloor \quad [\![e]\!]_\rho = d_0 + o \quad 0 \leq o \leq d_s - n}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto \delta(o, n)], \delta, \phi, \iota^+ \rangle}$$

$$\textsc{LoadStack}\ \frac{instr(\iota) = \lfloor r = [e]_n \rfloor \quad [\![e]\!]_\rho = a \quad cs :: \langle bp, \phi \rangle \vdash_{a,n} v \\ bp - \mid \phi \mid + \mathsf{GZ}_\perp + n \leq a \leq s_0 - \mathsf{GZ}_\top}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto v], \delta, \phi, \iota^+ \rangle}$$

The rule CALL for the function call instruction $\mathbf{call}\ e$ first verifies that the value $[\![e]\!]_\rho$ belongs to the set of function entry points $\mathcal{F}$. The current stack frame is divided into two parts $\phi_1 \cdot \phi_2$ where ϕ_1 is the local data of the caller and ϕ_2 is the new stack frame for the function call, which starts at the address contained in register $\mathbf{esp}$. The offset o between the start of the old stack frame and the new is verified to be smaller than the maximal frame size f_s. Note that since ϕ always represents the rest of the stack (including the guard zone), we are sure that the new bp is always inside the stack or the guard zone. Figure 4.1 visualizes what actions are allowed where in the stack, and what actions crash where. The actual method call is modeled as an execution starting at address f with the same environment ρ, the same data segment δ, and a stack frame ϕ_2 The end of the call is identified by the execution reaching a $\mathbf{ret}\ e'$ instruction. The value of $[\![e']\!]_{\rho'}$ is verified to be the return address using the architecture-dependent predicate $isret$. The return address is the next instruction to execute. The semantics verifies that callee-saved registers are restored after the function call. This is the role of the architecture-dependent $\sim$ predicate. For instance on X86 assembly, registers **esp**, **ebx** etc. are saved, so $\rho \sim \rho'$ states that both ρ and ρ' have the same value associated to these registers. For this same architecture, $isret$ checks that the return address has indeed been pushed to the call stack.

$$\textsc{Call}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \quad \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \\ o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad instr(\iota') = \lfloor \mathbf{ret}\ e' \rfloor \quad [\![e']\!]_{\rho'} = \iota^+ \\ \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle \vdash \langle \rho, \delta, \phi_2, f \rangle \longrightarrow^* \langle \rho', \delta', \phi_2', \iota' \rangle \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi_1 \cdot \phi_2, \iota \rangle \longrightarrow \langle \rho', \delta', \phi_1 \cdot \phi_2', \iota^+ \rangle}$$

Calling a trusted function is modeled with the rule CallTrust. This rule follows the same pattern as the rule for ordinary calls, except that the trusted call is allowed to modify the sandbox data but should leave the callee's stack frame unchanged. We model this as a non-deterministic rule that can return any δ' in its resulting state.

$$\textsc{CallTrust}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \delta, \phi_1 \cdot \phi_2, \iota \rangle \longrightarrow \langle \rho', \delta', \phi_1 \cdot \phi_2', \iota^+ \rangle}$$

The rules Cont and Jump model the instruction $\mathbf{jmpif}\ e_1\ e_2$ for conditional jumps to a computed address. If the condition e_1 evaluates to zero, execution continues with the next instruction. Otherwise, the value $[\![e_2]\!]_\rho$ is computed and it is verified that this new jump target is in the code block $Code$.

$$\textsc{Cont}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad [\![e_1]\!]_\rho = 0}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho, \delta, \phi, \iota^+ \rangle}$$

$$\textsc{Jump}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad [\![e_1]\!]_\rho \neq 0 \quad [\![e_2]\!]_\rho \in Code}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \langle \rho, \delta, \phi, [\![e_2]\!]_\rho \rangle}$$

Finally, we use the (secure) crash state ■ to model program termination in the rule HALT. The rule CRASH states that once in a crash state the execution stays in this state forever. This semantic sleight of hand simplifies the statement of the overall security property, which becomes essentially a progress property. Employing a specific error state would be equivalent but slightly more cumbersome.

$$\textsc{Halt}\ \frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\Gamma \vdash \langle \rho, \delta, \phi, \iota \rangle \longrightarrow \blacksquare} \qquad \textsc{Crash}\ \frac{}{\Gamma \vdash \blacksquare \longrightarrow \blacksquare}$$

4.1.2 The Sandbox Property

The side-conditions of the rules performing memory accesses ensure that the defensive semantics gets stuck when memory accesses do not respect the sandboxing property. This means that we can state our sandbox property as a simple *progress* property of the defensive semantics: as long as the semantics can progress to a new state (possibly the crash state) no security violation has occurred.

There is one obstacle to this, though: due to our *big-step* modeling of function calls, the semantics also gets stuck as soon as a function call does not terminate. In other words, all infinite loops are deemed insecure, which is clearly not what we want. To remedy this, our sandbox property is defined over the set of reachable states induced by the defensive semantics where the transition relation $\rightarrow$ is extended with a relation $\triangleright$ which for each $\mathbf{call}$ instruction explicitly adds a transition to the callee state.

$$\textsc{CallAcc}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \end{array}}{\langle \langle cs, bp, \rho_i \rangle, \langle \rho, \delta, \phi_1 \cdot \phi_2, \iota \rangle \rangle \triangleright \langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle, \langle \rho, \delta, \phi_2, f \rangle \rangle}$$

Dually, we also add a transition stating that, for a $\mathbf{ret}$ instruction, the return state is not stuck provided that the calling conventions are respected. Since the next step is taken care of by the CALL rule, the resulting state is just a witness that the execution can proceed; we reuse for this purpose the state ■.

$$\textsc{RetAcc}\ \frac{\rho_i \sim \rho' \quad isret(ret, \phi_1, \rho_i) \quad instr(\iota) = \lfloor \mathbf{ret}\ e' \rfloor \quad [\![e']\!]_{\rho'} = ret}{\langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, \langle \rho', \delta, \phi_2', \iota \rangle \rangle \triangleright \langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, \blacksquare \rangle}$$

Definition 12 (Augmented defensive semantics)**.** *The augmented defensive semantics*

$\Rightarrow$ is given by the union of the relation $\rightarrow$ and $\triangleright$ such that:

$$\frac{\Gamma \vdash \Sigma_1 \longrightarrow \Sigma_2}{\langle \Gamma, \Sigma_1 \rangle \Rightarrow \langle \Gamma, \Sigma_2 \rangle} \qquad \frac{\langle \Gamma_1, \Sigma_1 \rangle \triangleright \langle \Gamma_2, \Sigma_2 \rangle}{\langle \Gamma_1, \Sigma_1 \rangle \Rightarrow \langle \Gamma_2, \Sigma_2 \rangle}$$

The SFI sandbox property can then be expressed as the progress property of the augmented defensive semantics.

Definition 13 (Correct Context)**.** *We define the set of correct contexts as:*

$$\frac{\mid \phi_i \mid = GZ_\top \quad isret(ret, \phi_0, \rho_0) \quad ret \in Trust}{\langle [\langle s_0, \phi_i \rangle], s_0 - GZ_\top, \rho_0 \rangle \in \textcircled{\Gamma}}$$

$$\frac{\begin{array}{c}\langle cs, \phi_1, \rho \rangle \in \textcircled{\Gamma} \quad isret(ret, \phi_2, \rho_i) \quad ret \in Trust \quad \mid \phi_1 \mid < f_s \\ \mid cs \mid + \mid \phi_1 \mid + \mid \phi_2 \mid < s_s + GZ_\top\end{array}}{\langle cs :: \langle bp, \phi_1 \rangle, \phi_2, \rho_i \rangle \in \textcircled{\Gamma}}$$

Definition 14 (Initial States)**.** *Let the set of initial states be the set of states that of the program when the trusted library calls the untrusted module (either because it calls the entry point after loading the module, or because one of the functions was registered as a callback). The initial states are defined as:*

$$Init = \left\{ \langle \langle cs, \phi_i, \rho \rangle, \langle \rho, \delta, \phi, \iota \rangle \rangle \;\middle|\; \begin{array}{l} \langle cs, \phi_i, \rho \rangle \in \textcircled{\Gamma} \wedge \iota \in \mathcal{F} \\ \mid \phi \mid = s_s + GZ_\top + GZ_\bot - \mid cs \mid - \mid \phi_i \mid \\ isret(ret, \rho, \phi_i) \wedge ret \in Trust \end{array} \right\}$$

Definition 15 (Sandboxing as progress)**.** *The program satisfies the SFI sandbox property if the set of reachable states $Acc = \{s \mid \langle \Gamma_0, \Sigma_0 \rangle \Rightarrow^* s \wedge \langle \Gamma_0, \Sigma_0 \rangle \in Init\}$ satisfies $\forall s \in Acc. \exists s'. s \Rightarrow s'$.*

We write $Safe(Acc)$ if this is the case.

4.2 Dataless Semantics

In order to derive a modular static analyzer, we abstract the defensive semantics into a dataless semantics where the sandbox (the data) is abstracted away. Abstracting the sandbox away is what makes the analysis independent of the initial content of the data, and to whatever the module might want to read or write. This is also the reason why the sandboxing instructions are crucial for a secure module, and, as we will see in the last technical part of this thesis, it is the reason why the analysis also works when there are multiple threads.

The dataless semantics abstracts away the sandbox. Reading in the sandbox is replaced by an nondeterministic read of any value of the specified size, and writing to the sandbox does not change the current state, apart from the instruction pointer.

Most of the rules do not change compared to the defensive semantics, except for the form of their states. For instance, the ASSIGN rule does not change except that it does not reference δ anymore.

$$\textsc{Assign}\ \frac{instr(\iota) = \lfloor r = e \rfloor}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto \llbracket e \rrbracket_\rho], \phi, \iota^+ \rangle}$$

This is also the case for the following rules, that we reproduce below:

$$\textsc{StoreFrame}\ \frac{\begin{array}{c} instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = bp - \llbracket e_1 \rrbracket_\rho \\ 0 \leq o \leq \mid \phi \mid -\mathsf{GZ}_\bot - \mid e_2 \mid \quad bp - o \leq s_0 - \mathsf{GZ}_\top \end{array}}{\langle cs, bp, R \rangle \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho, \phi[o \mapsto \llbracket e_2 \rrbracket_\rho], 1 \rangle\ \iota^+}$$

$$\textsc{LdStCrash}\ \frac{\begin{array}{c} instr(\iota) = \lfloor [e_1] = e_2 \rfloor \vee instr(\iota) = \lfloor [r = [e_1]] \rfloor \quad \llbracket e_1 \rrbracket_\rho = a \\ (s_0 - \mathsf{GZ}_\top < a \leq s_0) \vee (bp - \mid \phi \mid < a < bp - \mid \phi \mid + \mathsf{GZ}_\bot + n) \quad n = \mid e_2 \mid \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \blacksquare}$$

$$\textsc{LoadStack}\ \frac{\begin{array}{c} instr(\iota) = \lfloor r = [e]_n \rfloor \quad \llbracket e \rrbracket_\rho = a \quad cs :: \langle bp, \phi \rangle \vdash_{a,n} v \\ bp - \mid \phi \mid + \mathsf{GZ}_\bot + n < a \leq s_0 - \mathsf{GZ}_\top \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto v], \phi, \iota^+ \rangle}$$

$$\textsc{Call}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{F} \quad \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \\ o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad instr(\iota') = \lfloor \mathbf{ret}\ e' \rfloor \quad \llbracket e' \rrbracket_{\rho'} = \iota^+ \\ \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle \vdash \langle \rho, \phi_2, f \rangle \longrightarrow^* \langle \rho', \phi'_2, \iota' \rangle \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \phi_1 \cdot \phi_2, \iota \rangle \longrightarrow \langle \rho', \phi_1 \cdot \phi'_2, \iota^+ \rangle}$$

$$\textsc{CallTrust}\ \frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad \llbracket e \rrbracket_\rho = f \quad f \in \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle cs, bp, \rho_i \rangle \vdash \langle \rho, \phi_1 \cdot \phi_2, \iota \rangle \longrightarrow \langle \rho', \phi_1 \cdot \phi'_2, \iota^+ \rangle}$$

$$\textsc{Cont}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = 0}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho, \phi, \iota^+ \rangle}$$

$$\textsc{Jump}\ \frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho \neq 0 \quad \llbracket e_2 \rrbracket_\rho \in Code}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho, \phi, \llbracket e_2 \rrbracket_\rho \rangle}$$

$$\textsc{Halt}\ \frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \blacksquare} \qquad \textsc{Crash}\ \frac{}{\Gamma \vdash \blacksquare \longrightarrow \blacksquare}$$

The only noticeable changes are in the following two rules. The STOREDATA rule now does nothing to the state, but still applies the same verifications as the defensive semantics.

$$\textsc{StoreData}\,\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad o = [\![e_1]\!]_\rho - d_0 \quad 0 \leq o \leq d_s - \mid e_2 \mid}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho, \phi, \iota^+ \rangle}$$

The LOADDATA rule however, updates the state as in the defensive semantics, but this time the value read from the sandbox is arbitrary.

$$\textsc{LoadData}\,\frac{instr(\iota) = \lfloor r = [e]_n \rfloor \quad [\![e]\!]_\rho = d_0 + o \quad 0 \leq o \leq d_s - n}{\Gamma \vdash \langle \rho, \phi, \iota \rangle \longrightarrow \langle \rho[r \mapsto v], \phi, \iota^+ \rangle}$$

This abstract semantics has a concretization:

$$\gamma(\langle \rho, \phi, \iota \rangle) = \left\{ \langle \rho, \delta, \phi, \iota \rangle \middle| \mid \delta \mid = d_s \right\}$$

$$\gamma(\blacksquare) = \{\blacksquare\}$$

As for the defensive semantics, we introduce two new rules to model the small step progress and avoid blocking on function calls when the return instruction is never reached at runtime (because it crashed (safely) or because of an infinite loop).

$$\textsc{CallAcc}\,\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \end{array}}{\langle \langle cs, bp, \rho_i \rangle, \langle \rho, \phi_1 \cdot \phi_2, \iota \rangle \rangle \triangleright \langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle, \langle \rho, \phi_2, f \rangle \rangle}$$

Dually, we also add a transition stating that, for a $\mathbf{ret}$ instruction, the return state is not stuck provided that the calling conventions are respected. Since the next step is taken care of by the CALL rule, the resulting state is just a witness that the execution can proceed; we reuse for this purpose the state $\blacksquare$.

$$\textsc{RetAcc}\,\frac{\rho_i \sim \rho' \quad isret(ret, \phi_1, \rho_i) \quad instr(\iota) = \lfloor \mathbf{ret}\ e' \rfloor \quad [\![e']\!]_{\rho'} = ret}{\langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, \langle \rho', \phi_2', \iota \rangle \rangle \triangleright \langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, \blacksquare \rangle}$$

In the following, we will use the notation $\circ^{\downarrow}$ for anything related to the dataless semantics. However, it is just a notation and two things with and without the symbol ($\circ$ and $\circ^{\downarrow}$) have nothing in common unless it is explicit.

Definition 16 (Augmented dataless semantics)**.** *The augmented dataless semantics*

$\Rightarrow$ is given by the union of the relation $\rightarrow$ and $\triangleright$ such that:

$$\frac{\Gamma \vdash \Sigma_1 \longrightarrow \Sigma_2}{\langle \Gamma, \Sigma_1 \rangle \Rightarrow \langle \Gamma, \Sigma_2 \rangle} \qquad \frac{\langle \Gamma_1, \Sigma_1 \rangle \triangleright \langle \Gamma_2, \Sigma_2 \rangle}{\langle \Gamma_1, \Sigma_1 \rangle \Rightarrow \langle \Gamma_2, \Sigma_2 \rangle}$$

The SFI sandbox property can then be expressed as the progress property of the augmented dataless semantics.

Definition 17 (Initial State)**.** *Let the initial states be:*

$$\mathit{Init}^{\downarrow} = \left\{ \langle \langle cs, \phi_i, \rho \rangle, \langle \rho, \phi, \iota \rangle \rangle \;\middle|\; \begin{array}{l} \langle cs, \phi_i, \rho \rangle \in \textcircled{\Gamma} \wedge \iota \in \mathcal{F} \\ \mid \phi \mid = s_s + GZ_{\top} + GZ_{\bot} - \mid cs \mid - \mid \phi_i \mid \\ isret(ret, \rho, \phi_i) \wedge ret \in \mathit{Trust} \end{array} \right\}$$

Definition 18 (Sandboxing as progress)**.** *The program satisfies the SFI sandbox property if the set of reachable states $\mathit{D\text{-}Acc} = \{ s \mid \langle \Gamma_0, \Sigma_0 \rangle \Rightarrow^* s \wedge \langle \Gamma_0, \Sigma_0 \rangle \in \mathit{Init}^{\downarrow} \}$ satisfies $\forall s \in \mathit{Acc}. \exists s'. s \Rightarrow s'$.*

We write $\mathit{D\text{-}Safe}(\mathit{D\text{-}Acc})$ if this is the case.

We now prove the four properties for the abstraction, as seen in Chapter 2.2.

Lemma 3 (Sound Initial State)**.**

$$\forall s_0 \in \mathit{Init}, \exists s_0^{\downarrow} \in \mathit{Init}^{\downarrow}, s_0 \in \gamma(s_0^{\downarrow})$$

Proof. For s_0, we can construct the same $s_0^{\downarrow}$, with an abstracted sandbox. Its concretization contains at least a state whose sandbox is the same as s_0. □

Lemma 4 (Sound Order)**.**

$$\forall s^{\downarrow} s'^{\downarrow}, s^{\downarrow} \sqsubseteq s'^{\downarrow} \implies \gamma(s^{\downarrow}) \subseteq \gamma(s'^{\downarrow}).$$

Proof. The is no specific order on the dataless states, except that an element is always comparable to itself. The abstraction of an element is always included in itself, hence the result. □

Lemma 5 (Sound Property)**.**

$$\forall s^{\downarrow} \in \mathit{D\text{-}Acc}, \mathit{D\text{-}Safe}(s^{\downarrow}) \implies (\forall s \in \gamma(s^{\downarrow}), \mathit{Safe}(s)).$$

Proof. $D\text{-}Safe(s^{\downarrow})$ means that there exists $s'^{\downarrow}$ such that $s^{\downarrow} \to s'^{\downarrow}$. By case analysis on the semantics rule that allows the transition, we can prove the lemma. In every case, there is a same-named rule in the defensive semantics that has exactly the same preconditions. Since none of these conditions refer to the specific content of the sandbox (δ), any concretization of $s^{\downarrow}$ also verifies the same conditions, so it verifies the security property. □

Lemma 6 (Sound Abstraction, part 1)**.** *This lemma is an intermediate result that corresponds to a special case of the proof of the abstraction soundness. If the progress on the concrete semantics is due to an intra-procedural rule, we show that the abstraction is sound.*

$$\forall \Gamma, \forall s^{\downarrow}, \forall s', \forall s \in \gamma(s^{\downarrow}), \Gamma \vdash s \longrightarrow s' \implies \exists s'^{\downarrow}, \Gamma \vdash s^{\downarrow} \longrightarrow s'^{\downarrow} \wedge s' \in \gamma(s'^{\downarrow})$$

Proof. With these hypotheses, we reason by induction on $\Gamma \vdash s \to s'$. On the base cases (any rule except the call rule), we can note that the conditions are the same in both semantics, and since the states are the same (except for the sandbox) in both semantics and the conditions do not depend on the sandbox, the dataless semantics can progress to a new state $s'^{\downarrow}$, and we have $s' \in \gamma(s'^{\downarrow})$.

Only the LOADDATA rule can have more than one following state, and one of them is a state where the value read is the same as the one read in the defensive semantics.

For the inductive case, the induction hypothesis is that we have $\Gamma' \vdash s_1 \to^* s_2$ and $\Gamma' \vdash s_1^{\downarrow} \to^* s_2^{\downarrow}$ with $s_1 \in \gamma(s_1^{\downarrow})$ and $s_2 \in \gamma(s_2^{\downarrow})$. With this, the dataless semantics can also progress on the Call rule, and the new state is an abstraction of the concrete new state: $s' \in \gamma(s'^{\downarrow})$.

The reason why we can do this is because $\longrightarrow$ and $\longrightarrow^*$ are mutually inductive and we have an induction principle on them. □

Lemma 7 (Sound Abstraction)**.** *This is the full lemma that states the abstraction soundness for the dataless semantics.*

$$\forall \Gamma_1, \Gamma_2, \forall s^{\downarrow}, \forall s', \forall s \in \gamma(s^{\downarrow}), (\Gamma_1, s) \Rightarrow (\Gamma_2, s') \implies \exists s'^{\downarrow}, (\Gamma_1, s^{\downarrow}) \Rightarrow (\Gamma_2, s'^{\downarrow}) \wedge s' \in \gamma(s'^{\downarrow})$$

Proof. We have two cases: either $\Gamma_1 \vdash s \longrightarrow s'$, in which case we can apply the previous lemma, or $(\Gamma_1, s) \triangleright (\Gamma_2, s')$, in which case there are again two possible rules.

Both rule conditions are independent of the content of the sandbox, and are the same for the dataless semantics, so there is progress on the dataless semantics side. By case analysis on these rules, we can see that the states in both semantics which were related by the concretization both have a next state, and they are still related by the concretization, so we can conclude. □

4.3 Intra-procedural Semantics

In order to derive a modular static analyzer we abstract the dataless semantics into an intra-procedural semantics where the accessible states $I\text{-}Acc$ are computed for each function separately: $I\text{-}Acc = \bigcup_{f \in \mathcal{F}} I\text{-}Acc(f)$.

The intra-procedural semantics abstracts away all of the call stack, except a small portion of the stack that includes the frame of the caller. Thus the stack component is abstracted into two small zones of the stack above and below the current base pointer, representing the frames of the caller and the callee (the currently executing function).

Both are modeled by memory regions of size f_s where f_s is a chosen maximum size of these abstract stack frames. Fig. 4.2 illustrates the abstraction of the stack segment.

The current code can write to its own frame (ϕ) and read from both frames (both ϕ and ϕ_i). Note that ϕ_i may correspond to more than one frame in the dataless semantics, and does not always stop at a frame boundary either. The sizes of both guard zones are set such that the abstract frames are always contained in the stack, possibly overlapping a guard zone, as long as bp stays inside the writable part of the stack, or at most f_s below the stack, inside the guard zone.

The intra-procedural semantics only has an abstraction of the stack, where it can read and write to only a portion of it. This semantics is an abstraction of the dataless semantics, where the full stack is available. The intra-procedural semantics does not make any check that a write or a read happens inside the writable or readable space of the stack.

An initial state is a state at the beginning of a function, when it is called. We assume this initial state as a base pointer inside the stack, or at most f_s below the stack, inside the guard zone. This is assumed by the execution model for the initial function called in the untrusted module, but is it the case for other functions that are called afterwards? Without any check, the inter-procedural semantics would continue to execute happily after a function call, when the dataless semantics, which does a check, would have blocked.

We introduce β, a boolean value that records whether a write to the stack happened during the execution of the current function. The intuition is that, if the value is true, a write was attempted to the stack. In the intra-procedural semantics, the write was recorded and execution proceeded normally. In the dataless semantics however, since bp is still inside the stack or within f_s inside the guard zone, the write is either successful or in the guard zone. If the write was successful, bp could only have been above the guard zone, and it is possible to do a function call later, that respects the intra-procedural semantics invariant. If the write was not successful (because it was inside the guard zone), the semantics immediately crashes, and all later states in the intra-procedural semantics are irrelevant.

It is important to choose the size of the guard zone wisely: since a write can happen at most f_s below bp and bp can be as low as f_s below the stack, the bottom guard zone needs to be at least $2f_s$ in size. Since bp can be at most at the top of the stack, and a program can read at most f_s above bp, the top guard zone needs to be at least f_s in size.

4.3.1 Formal Proof of Security

The judgment of the intra-procedural semantics is of the form: $\Gamma \vdash s \to s'$ where the context $\Gamma = \langle \phi_i, bp, \rho_i \rangle \in \mathit{Ctx}^\natural$ is constant. Here, ϕ_i is the memory zone of size f_s above the base pointer, that includes the frame of the caller (because of the semantics invariant that a frame size is at most f_s), bp is the base pointer and ρ_i is the initial environment at function call. The states s and s' are either the crash state $\blacksquare$ or of the form $\langle \rho, \phi^{(\beta)}, \iota \rangle \in \mathit{State}^\natural$, where ρ and ι are the environment and the current instruction pointer as in the dataless semantics, ϕ is the current frame, limited to f_s bytes and β is the boolean that records whether a write happened in the stack.

The intra-procedural and dataless semantics are linked by the concretization function $\gamma : \mathit{Ctx}^\natural \times \mathit{State}^\natural \to \mathcal{P}((\mathit{Ctx}^\downarrow \times \mathit{State}^\downarrow) \cup \{\blacksquare\})$.

This link is mostly the identity (for the base pointer, the initial environment, the environment and the instruction pointer).

The concretization constructs the call stack and the current defensive frame in such a way that, once appended to one another, they form a memory region of size s_s, and both ϕ_i and ϕ are windows of size f_s around the address pointed to by the base pointer. To express the fact that, when the base pointer is in the guard zone and a write occurred in the current frame, the intra-procedural semantics continues, but the dataless semantics crashed, we only concretize to non-crash states when no write occurred or when the base pointer is in the stack. Formally, we have:

$$\gamma(\langle\phi_i, bp, \rho_i\rangle, \langle\rho, \phi^{(\beta)}, \iota\rangle) = \{\blacksquare\} \cup \left\{ \langle cs, bp, \rho_i\rangle, \langle\rho, \phi, \iota\rangle \;\middle|\; \begin{array}{l} \phi = \phi_{\big|[0,f_s-1]} \\ \phi_i = \overline{cs}_{\big|[|\overline{cs}|-f_s,|\overline{cs}|]} \\ \beta = 0 \vee bp \geq s_0 - s_s + \mathsf{GZ}_\perp \end{array} \right\}$$

where $\overline{cs}$ is obtained by concatenating the different stack frames of the call stack and $\phi_{\big|[a,b]}$ extracts the sub-list between the indexes a and b.

$$\overline{cs} = \begin{cases} [] & \text{if } cs = [] \\ \phi \cdot \overline{cs'} & \text{if } cs = \langle bp, \phi\rangle :: cs' \end{cases} \qquad \phi_{\big|[a,b]} = \begin{cases} [] & \text{if } a > b \\ \phi(a) :: \phi_{\big|[a+1,b]} & \text{otherwise} \end{cases}$$

Except for the handling of stack overflows and underflows, the rules for the intra-procedural semantics are fairly standard. Most of the rules are very similar to their dataless counterpart, such as the FUNASSIGN rule, that corresponds to the ASSIGN rule in the dataless semantics. Note that in most of these rules, β is preserved; it is only updated when writing to the stack.

$$\textsc{FunAssign}\;\frac{instr(\iota) = \lfloor r = e \rfloor}{\Gamma \vdash \langle\rho, \phi^{(\beta)}, \iota\rangle \to^{\natural} \langle\rho[r \mapsto [\![e]\!]_\rho], \phi^{(\beta)}, \iota^+\rangle}$$

The FUNSTD rule maps to the STOREDATA rule, with the same runtime verifications.

$$\textsc{FunStD}\;\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad [\![e_1]\!]_\rho = d_0 + o \quad 0 \leq o <| \delta |}{\Gamma \vdash \langle\rho, \phi^{(\beta)}, \iota\rangle \to^{\natural} \langle\rho, \phi^{(\beta)}, \iota^+\rangle}$$

Writing to the current frame requires less runtime verification: we only check that the write happens in the bounds of the frame (within f_s of the current base pointer).

$$\textsc{FunStF}\;\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad [\![e_1]\!]_\rho = bp - o \quad 0 \leq o <| \phi |}{\langle\phi_i, bp, \rho_i\rangle \vdash \langle\rho, \phi^{(\beta)}, \iota\rangle \to^{\natural} \langle\rho, \phi[o \mapsto [\![e_2]\!]_\rho]^{(1)}, \iota^+\rangle}$$

The FUNLDD rule maps to the LOADDATA rule, with the same runtime verifications.

$$\textsc{FunLdD}\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad [\![e]\!]_\rho = d_0 + o \quad 0 \leq o <\mid \delta \mid}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto v], \phi^{(\beta)}, \iota^+ \rangle}$$

The rule FUNLDARG shows how to access the arguments of the function that are placed in the stack frame of the caller modeled by ϕ_i. As with writing to the stack, there is not as much checks as in the dataless semantics: we merely check that the read is within the bounds of the caller's frame.

$$\textsc{FunLdArg}\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad [\![e]\!]_\rho = (bp + f_s) - o \quad 0 \leq o < f_s}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto \phi_i(o)], \phi^{(\beta)}, \iota^+ \rangle}$$

While the rule FULLDFRAME shows how to access the current frame of the function, ϕ. As with the previous rule, we merely check that we read within the bounds of the current frame, i.e. at most f_s below bp.

The base address of ϕ_i is obtained by incrementing the base pointer bp by the stack frame size f_s and checking that the offset o is in range $[0; f_s[$. The soundness of this rule exploits the fact that the defensive stack is guarded by GZ$_\top$. As a result, if FUNLDARG succeeds, either the memory access also succeeds in the dataless semantics (rule LOADSTACK) or it accesses the guard zone GZ$_\top$ and triggers a crash (rule LDSTCRASH).

$$\textsc{FunLdFrame}\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad [\![e]\!]_\rho = bp - o \quad 0 \leq o < f_s}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho[r \mapsto \phi(o)], \phi^{(\beta)}, \iota^+ \rangle}$$

Note that there is no equivalent to the LDSTCRASH rule of the dataless semantics, because this semantics simply continues its execution even if it accesses a guard zone. As we explained before, the concretization of this semantics will crash, so at any point in time, either the access was correct, or a crash occurred in the past.

Jumps perform the same runtime verifications as in the dataless semantics.

$$\textsc{FunCont}\frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad [\![e_1]\!]_\rho = 0}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{FunJump}\frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad [\![e_1]\!]_\rho \neq 0 [\![e_2]\!]_\rho \in Code}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \rightarrow^\natural \langle \rho, \phi^{(\beta)}, [\![e_2]\!]_\rho \rangle}$$

For the **call** instruction, the rule is similar to the dataless semantics rule CALL with the notable exception that no recursive call is made, and the position of bp is not checked; it is only checked that the new frame is within f_s of the old frame boundary.

$$\textsc{FunCall}\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \cup \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad 0 \leq o < f_s \quad \mid \phi_1 \mid = o \\ isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \end{array}}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle \to^\natural \langle \rho', \phi_1 \cdot \phi_2'^{(1)}, \iota^+ \rangle}$$

Function returns perform the same runtime checks as the dataless semantics, and go straight to the crash state, as in the dataless semantics.

$$\textsc{FunRet}\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{ret}\ e \rfloor \quad [\![e]\!]_\rho = ret \\ isret(\iota^+, \rho_i, \phi_i) \quad \rho_i \sim \rho \end{array}}{\langle \phi_i, bp, \rho_i \rangle \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \to^\natural \blacksquare}$$

The last two rules are the same as in the dataless semantics.

$$\textsc{FunHalt}\frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\Gamma \vdash \langle \rho, \phi^{(\beta)}, \iota \rangle \to^\natural \blacksquare} \qquad \textsc{FunCrash}\frac{}{\Gamma \vdash \blacksquare \to^\natural \blacksquare}$$

For each function entry $\iota \in \mathcal{F}$, the initial states $Init(\iota) \subseteq Ctx^\natural \times State^\natural$ are defined by:

$$Init(\iota) = \left\{ \langle \langle \phi_i, bp, \rho \rangle, \langle \rho, \phi^{(0)}, \iota \rangle \rangle \,\middle|\, \begin{array}{l} \mid \phi \mid = \mid \phi_i \mid = f_s \wedge \\ bp > s_0 - s_s + \mathsf{GZ}_\perp - f_s \end{array} \right\}$$

As we already discussed, the frames ϕ and ϕ_i have length f_s. At the function start, the environments of the caller and the callee are the same; the base pointer is so that there is below it a stack frame of size at least f_s and no memory write has been performed on the current frame ϕ. For a given function entry point $f \in \mathcal{F}$, the reachable states are defined as

$$I\text{-}Acc(f) = \{(\Gamma, s) \mid \Gamma \vdash s_0 \to^* s \wedge (\Gamma, s_0) \in Init(f)\}.$$

For the proofs, we need to define a transition system that does not have a context, and we call it $\Rightarrow$ as before:

$$\frac{\Gamma s \longrightarrow s'}{(\Gamma, s) \Rightarrow (\Gamma, s')}$$

We also add this transition on calls, which will be used to prove the abstraction soundness later:

$$\textsc{FunSub}\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \\ \rho(\mathbf{esp}) = bp - o \quad 0 \leq o < f_s \quad \mid \phi_1 \mid = o \\ isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad init \in Init(f) \end{array}}{(\langle \phi_i, bp, \rho_i \rangle, \langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle) \Rightarrow init}$$

Note that this rule does not add any new state to the set of reachable states.

The intra-procedural semantics is also defensive and gets stuck when abstract verification conditions are not met.

Definition 19 (Intra-procedural progress)**.** *The intra-procedural states are safe (written $I\text{-}Safe(I\text{-}Acc)$) iff* $\forall f \in \mathcal{F}, \forall (\Gamma, s) \in I\text{-}Acc(f). \exists s'. \Gamma \vdash s \rightarrow s'$.

The checked conditions are sufficient (but may not be necessary). For instance, the intra-procedural semantics gets stuck when an access is performed outside the bounds of the current stack frame ϕ. However, because ϕ only models a prefix of the frame of the dataless semantics, the dataless semantics may not be stuck. As a result, the usual result $Acc \subseteq \gamma(I\text{-}Acc)$ does not hold. The methodology however does not need the usual result and only requires that we prove the following four lemmas:

Lemma 8 (Sound Initial State)**.**

$$\forall s_0^\downarrow \in Init^\downarrow, \exists s_0^\downarrow \in Init^\natural, s_0^\downarrow \in \gamma(s_0^\natural).$$

Proof. By definition, $s_0^\downarrow = \langle \langle cs, \phi_i, \rho \rangle, \langle \rho, \phi, \iota \rangle \rangle$ where $\langle cs, \phi_i, \rho \rangle \in \textcircled{Γ}$, $\iota \in \mathcal{F}, isret(ret, \rho, \phi_i)$ and $ret \in Trust$. (Note: there is a condition on the size of the stack, but this is always respected by the concretization).

By construction, cs and ϕ both correspond to a part of the stack that is at least f_s in size. We can therefore select their restriction to the f_s bytes around bp. By choosing an intra thread state that has this partial stack, the same environment, instruction pointer, and a context with the same bp, environment and that restriction of cs as its previous frame, we have constructed an initial state for the intra-procedural semantics, whose concretization contains $s_0^\downarrow$. □

Lemma 9 (Sound Order)**.**

$$\forall s^\natural s'^\natural, s^\natural \sqsubseteq s'^\natural \implies \gamma(s^\natural) \subseteq \gamma(s'^\natural).$$

Proof. The is no specific order on the dataless states, except that an element is always comparable to itself. The abstraction of an element is always included in itself, hence the result. □

Lemma 10 (Sound Property)**.**

$$\forall s^\natural \in \mathit{I\text{-}Acc}, \mathit{I\text{-}Safe}(s^\natural) \implies (\forall s^\downarrow \in \gamma(s^\natural), \mathit{D\text{-}Safe}(s^\downarrow)).$$

Proof. If $s^\natural$ is secure, it means there is a transition from it to another state $s'^\natural$. Since $s^\natural$ is reachable from an initial state, its bp is also either in the stack or at most f_s below it.

The transition is due to a rule of the intra-procedural semantics. By case analysis on which rule occurs, we can conclude. In most cases, the conditions of the rule are the same as the conditions of the corresponding rule in the dataless semantics, and any concretization of $s^\natural$ can follow the rule and transition, which means they are secure.

The rules that we have to analyze more closely are:

— The `FunStF` rule: Since bp is in the stack or at least f_s in the guard zone, because of the conditions, the write occurs in the stack or in the guard zone. If it occurs in the stack, the conditions are verified for the `StoreFrame` rule in the dataless semantics, otherwise the conditions are verified for the `LdStCrash` rule in the dataless semantics.

— The `FunLdArg` rule: For the same reason, the read occurs either in the stack or in the top guard zone, which means the conditions for respectively the conditions of the `LoadStack` rule or the `LdStCrash` rule are verified.

— The `FunLdFrame` rule: Similarly, the read occurs either in the stack or in the bottom guard zone, which means one of the two rules has its conditions verified.

— The `FunCall` rule: because of the position of bp, the new bp is still inside the stack or the guard zone, which means the condition is verified for the `Call` or `CallT` rule (because in the dataless semantics, ϕ corresponds to the stack and both guard zones).

— The `FunSub` rule: the conditions are the same as for the `FunCall` rule, so the previous case applies.

□

Lemma 11 (Sound Abstraction)**.**

$$\forall \Gamma_1, \Gamma_2, \forall s^{\natural}, \forall s', \forall s^{\downarrow} \in \gamma(s^{\natural}), s^{\downarrow} \in D\text{-}Acc \implies (\Gamma_1, s^{\downarrow}) \Rightarrow (\Gamma_2, s'^{\downarrow}) \implies \exists s'^{\natural}, (\Gamma_1, s^{\natural}) \Rightarrow (\Gamma_2, s'^{\natural}) \wedge s'^{\downarrow} \in$$

Proof. Let's take a reachable intra-procedural state $s^{\natural}$ and one of its concretization, $s^{\downarrow}$. Suppose the dataless state is part of a transition to $s'^{\downarrow}$. We can use case analysis on the rule that leads to that new state. In most cases, the conditions for the transition in the dataless semantics are the same as the conditions that would allow a transition by the matching rule in the intra-procedural semantics (e.g. `Assign` and `FunAssign` both only need the instruction to be an assignation), and in these cases, the following state in the intra-procedural semantics admits $s'^{\downarrow}$ as one of its concretizations.

The rules that we have to analyze more closely are:

— The `StoreFrame` and `LoadStack` rules: these rules have a condition that the address is in the frame (resp. the stack), which might be bigger that the window we have in the intra-procedural semantics. However, since the reachable states are supposed to be safe, there has to be a following state. Therefore, because of the conditions and what rules can apply to get to a following state, the write (resp. read) must happen within f_s of the base pointer. With this condition, the rule in the intra-procedural semantics that applies updated the state in the same way as the dataless semantics.

— The `LdStCrash` rule: If a read or a write occurs in the guard zone, the corresponding rule in the intra-procedural semantics is one of the read rules in the frame or the stack, or a write to the frame, which was actually in the guard zone. The following state in the intra-procedural semantics is not the crash state, but its concretization contains the crash state by definition.

— The `Call` and `CallTrust` rules: Similar conditions are verified, and the intra-procedural rule does not even verify that the new bp is inside the stack or guard zone.

— The `CallAcc` rule: Since the intra-procedural state is safe, $\beta = 1$. bp is inside the stack (otherwise, the concretization would only contain ■ which cannot lead to the `CallAcc` rule), so there is a transition in the intra-procedural semantics through the `FunSub` rule to any initial state. In the dataless semantics, the next bp is inside the stack or within f_s in the guard zone because of the restriction on the position of the next bp, which makes the next dataless state a concretization of an initial state of the function.

□

4.4 Abstract Semantics

To get a modular executable verifier, we abstract further the intra-procedural semantics. The verifier needs to track numeric values used as addresses, in order to guarantee that memory accesses are within the sandbox or within the current stack frame, hence we need domains tailored for such uses of numeric values. The verifier also needs to gather some input-output relational information about the registers and verify that, at the end of the functions, callee-saved registers are restored to their initial values.

4.4.1 Abstract Domains

As pioneered by Balakrishnan and Reps [5], we perform a Value Set Analysis (VSA) where abstract locations (*a-locs*) are DATA for the sandbox region and CODE for the code region. We also introduce an abstract location for the function return RET, the base pointer BP and for each register e.g. EAX, EBX. To model purely numeric data, we have a dedicated *a-locs* ZERO with value 0:

$$\text{a-locs} = \{\mathsf{ZERO}, \mathsf{DATA}, \mathsf{CODE}, \mathsf{RET}, \mathsf{BP}, \mathsf{EAX}, \mathsf{EBX}, \ldots\}.$$

a-locs are equipped with an arbitrary non-relational numeric domain. The abstract value domain $\mathbb{B}^\sharp_*$ is therefore a pair (L, o) made of an abstract location L and a numeric abstraction $o \in D^\sharp$. For each concrete operation $\diamond$ on values, the transfer function on abstract (L, o)-values is using the corresponding operation $\diamond^\sharp$ of the abstract domain. For instance, for addition and subtraction, we get:

$$\begin{array}{ll} (L, o_1) +^\sharp (\mathsf{ZERO}, o_2) = (L, o_1 +^\sharp o_2) & (\mathsf{ZERO}, o_1) +^\sharp (L, o_2) = (L, o_1 +^\sharp o_2) \\ (L, o_1) -^\sharp (\mathsf{ZERO}, o_2) = (L, o_1 -^\sharp o_2) & (L, o_1) -^\sharp (L, o_2) = (\mathsf{ZERO}, o_1 -^\sharp o_2) \end{array}$$

When symbolic computations are not possible, it is always possible to abstract (L, o)

by $(\text{ZERO}, \top)$ and use numeric transfer functions. As the usual sandboxing technique consists in masking an address using a bitwise $\&$ ($[e_1] := e_2 \rightsquigarrow [d_0 + e_1 \& 1^k] := e_2$) [1] we opt, in our implementation, for the bitfield domain [29].

Furthermore, because the concretization of the $Zero$ label is known in advance to be 0, we can in certain cases use the operation on the offset and keep the label as $Zero$:

$$(Zero, o1) \diamond (Zero, o2) = (Zero, (o1 \diamond^\sharp o2))$$

If only one of the labels is different from $Zero$, we can still retain some precision in certain cases. For instance, an and operation between $(Zero, 0)$, whose concretization is 0, and $(Zero, top)$ is always $(Zero, 0)$. This is especially useful because sandboxing instructions use and operations to ensure the value is within the sandbox.

$$(L, o1) \diamond (Zero, o2) = (Zero, top) \diamond (Zero, o2)$$

$$(Zero, o1) \diamond (L, o2) = (Zero, o1) \diamond (Zero, top)$$

All other abstract operations return the top value $(Zero, top)$.

For instance, the sandboxing operation looks like $eax = eax \&\& 0x00ffffff + 0x11000000$. If before the execution of this instruction, eax is associated to $(Top, \top)$, the $\&\&$ operation results in $(Zero, 0x00\top\top\top\top\top\top)$ and the addition (of the sandbox mask) results in $(Data, 0x00\top\top\top\top\top\top)$, which is always inside the sandbox.

4.4.2 Abstract State

The abstract machine state at a program point is the product of an abstract environment $Env^\sharp$, an abstract frame $Frame^\sharp$, and a code pointer $\mathbb{B}_*$.

$$Env^\sharp = Reg \to \mathbb{B}_*^\sharp \quad Frame^\sharp = (\mathbb{B}_*^\sharp)^{f_s} \times \mathbb{B} \quad State^\sharp = Env^\sharp \times Frame^\sharp \times \mathbb{B}_*.$$

The abstract frame is annotated by a boolean indicating whether a memory write has definitively occurred in the stack frame.

The concretization function is parametrized by a mapping $\lambda : \text{a-locs} \to \mathbb{B}_*$ assigning a numeric value to abstract locations and the concretization function $\gamma : D^\sharp \to \mathcal{P}(\mathbb{B}_*)$ of

1. This exploits the property that the range of the sandbox is a power of 2.

the numeric domain. The concretization is then obtained using standard constructions:

$$\begin{array}{lcl}
\gamma_\lambda(L, o) & = & \{v + \lambda(L) \mid v \in \gamma(o)\} \\
\gamma_\lambda(\rho^\sharp) & = & \{\rho \mid \forall r.\rho(r) \in \gamma_\lambda(\rho^\sharp(r))\} \\
\gamma_\lambda(\phi^\sharp) & = & \{\phi \mid \forall i \in [0, f_s].\phi(i) \in \gamma_\lambda(\phi^\sharp(i))\} \\
\gamma_\lambda(\langle \rho^\sharp, \phi^{\sharp(b)}, \iota\rangle) & = & \left\{\langle \rho, \phi^{(\beta)}, \iota\rangle \middle| \beta \geq b \wedge \rho \in \gamma_\lambda(\rho^\sharp) \wedge \phi \in \gamma_\lambda(\phi^\sharp)\right\}
\end{array}$$

The mapping λ denotes a set of intra-procedural contexts such that a register r in the environment ρ_i has the value $\lambda(r)$ and the return address is constrained by the calling conventions.

$$\gamma(\lambda) = \left\{ \langle \phi_i, bp, \rho_i \rangle \middle| \begin{array}{l} \forall r, \rho(r) = \lambda(r), \\ bp = \lambda(\mathsf{BP}) = \lambda(\mathsf{ESP}), \quad isret(\lambda(\mathsf{RET}), \rho_i, \phi_i) \end{array} \right\}.$$

Finally, the whole concretization $\gamma : State^\sharp \to \mathcal{P}(Ctx^\natural \times State^\natural)$ is defined as:

$$\gamma(s^\sharp) = \{\langle \Gamma, \blacksquare \rangle\} \cup \{\langle \Gamma, s\rangle \mid \exists \lambda, \Gamma \in \gamma(\lambda) \wedge s \in \gamma_\lambda(s^\sharp)\}.$$

4.4.3 Abstract Semantics

The abstract semantics takes the form of a transition system that is presented in Fig. 4.3.

The rule AASSIGN abstracts the assignment to a register and consists in evaluating the expression e using the abstract domain of Section 4.4.1. A memory store is modeled by the rules ASTD and ASTF depending on whether the address is within the sandbox or within the current stack frame. Both rules ensure that the offset is within the bounds of the memory region. A memory load is modeled by the rules ALDD, ALDF or ALDS depending on whether the address is within the sandbox, the current stack frame or the caller stack frame. Each memory access is protected by a verification condition ensuring that the offset is within the relevant bounds. For the ALDF rule, the memory offset off is used to fetch the abstract value from the abstract frame ϕ. As the sandbox and the caller frame are not represented, we get the top element of the abstract domain i.e. $(\mathsf{ZERO}, \top)$. The rule ACALL models function calls. It checks whether the target of the call is a trusted ($f \in \mathcal{T}$) or untrusted function ($f \in \mathcal{F}$). For the call to proceed, the stack pointer $\mathbf{esp}$ must be within the bounds of the current stack frame and the return address ι^+ needs to be stored according to the calling conventions ($isret(\iota^+, \rho, \phi_1)$).

After the call, the resulting environment ρ' satisfies that the callee-saved registers are restored to their values in ρ ($\rho \sim \rho'$) and the suffix of the current frame ϕ_2' is arbitrary i.e. $\phi_2' = (\text{ZERO}, \top)^{|\phi_2|}$.

The rule ARET ensures that the expression e evaluates to the return of the current function ($[\![e]\!]_\rho = (\text{RET}, o) \quad \{0\} = \gamma(o)$), and also that the callee-saved registers are restored to their initial values. For instance, for $\mathbf{ebx}$, $preserve(\rho)$ ensures that $\rho(\mathbf{ebx}) = (EBX, o)$ with $\gamma(o) = \{0\}$.

The rules ACONT and AJUMP model control-flow transfer and check that the obtained code pointer is within the bounds of the code. The last two rules AHALT and ACRASH model the crash state that is produced by the $\mathbf{hlt}$ instruction and is its own successor.

Like the intra-procedural semantics, the abstract semantics is safe if it is not stuck (Definition 20).

Definition 20 (Abstract progress)**.** *The reachable intra-procedural states are safe (written $A\text{-}Safe(A\text{-}Acc)$) iff $\forall f \in \mathcal{F}, \forall s \in A\text{-}Acc(f).\exists s'.s \to s'$.*

The abstract semantics embeds abstract verification conditions that are only sufficient but not necessary for the intra-procedural semantics. As a result, it only computes a safe approximation under the condition that all the reachable abstract states are safe.

Lemma 12 (Sound Initial State)**.**

$$\forall s_0^\natural \in Init^\natural, \exists s_0^\natural \in Init^\sharp, s_0^\natural \in \gamma(s_0^\sharp).$$

Proof. Let's take $S \in Init(f) = \langle\langle \phi_i, bp, \rho\rangle, \langle \rho, \phi^{(0)}, f\rangle\rangle$.

We can construct $S^\sharp = \langle\langle \phi_i', bp, \rho'\rangle, \langle \rho', \phi'^{(0)}, f\rangle \in AInit(f)$ such that $S \in \gamma(S^\sharp)$, by making sure ρ', ϕ' and ϕ_i' have the same structure as their concrete counterpart, and associate a value to any location, whose concretization contains the value at that location in the counterpart. □

Lemma 13 (Sound Order)**.**

$$\forall s^\sharp s'^\sharp, s^\sharp \sqsubseteq s'^\sharp \implies \gamma(s^\sharp) \subseteq \gamma(s'^\sharp).$$

Proof. The concretization of an abstract value is the set of bitfields that have the same bits set, unset, and any bit for unknown bits. For two values to be comparable, the smallest must have the same bits set, unset and may set or unset bits that are unknown in the bigger value, so the bigger value has more concrete values than the smaller value. The order is sound for the abstract value.

For an abstract environment to be bigger than another, it must be the case that to each register, it associate a value bigger (and comparable) than the value associated to that register in the smaller environment. This represents more concrete environments, so the order is sound for environments.

The same reasonning can be applied to the stack, and then to the states. □

Lemma 14 (Sound Property)**.**

$$\forall s^\sharp \in \textit{A-Acc}, \textit{A-Safe}(s^\sharp) \implies (\forall s^\natural \in \gamma(s^\sharp), \textit{I-Safe}(s^\natural)).$$

Proof. When an abstract state has a following state, the conditions ensure that any concretization can progress using the corresponding intraprocedural rule. □

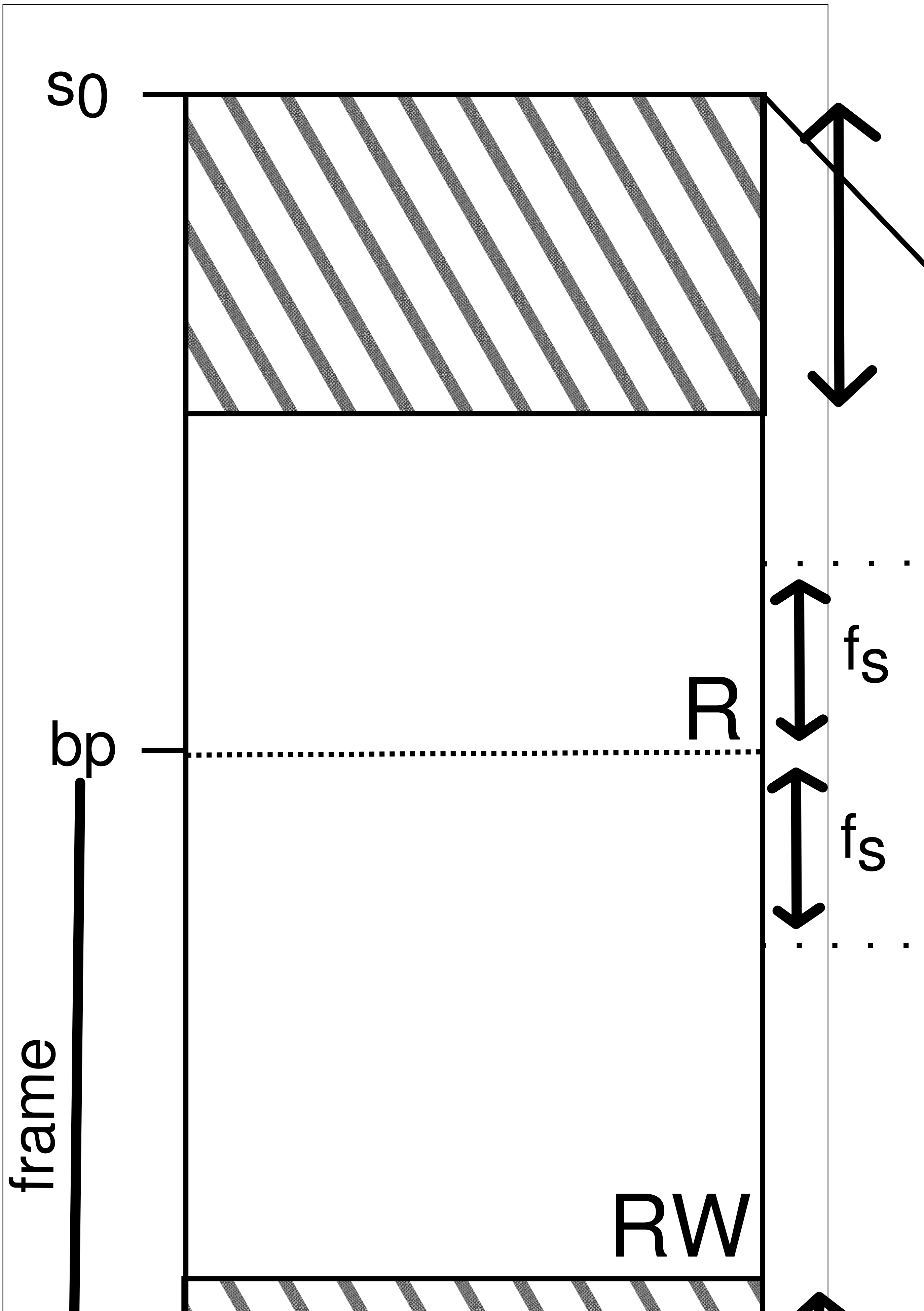
Lemma 15 (Sound Abstraction)**.**

$$\begin{aligned}&\forall \Gamma_1, \Gamma_2, \forall s^\sharp, \forall s', \forall s^\natural \in \gamma(s^\sharp), s^\natural \in \textit{I-Acc} \implies \\ &\quad (\Gamma_1, s^\natural) \Rightarrow (\Gamma_2, s'^\natural) \implies \exists s'^\sharp, (\Gamma_1, s^\sharp) \Rightarrow (\Gamma_2, s'^\sharp) \wedge s'^\natural \in \gamma(s'^\sharp)\end{aligned}$$

Proof. By case analysis on $\longrightarrow$, we can see that the preconditions of the abstract semantics are the same or more restrictive than those of the intra procedural semantics, so there is at least a following abstract state, that follows the corresponding rule. Still by case analysis, we can see that the new intraprocedural state is part of the concretization of the new abstract state. □

We now have proved the four properties on all the intermediate semantics. As we did in the methodology, we can deduce an abstract interpreter from the abstract semantics. This abstract interpreter is able to answer the question: can the defensive semantic be blocked? If the answer is no, the module is by definition secure, so it is accepted. Otherwise, the module might not be secure, so it is rejected.

So, we can now implement a verifier that uses the abstract semantics to check whether a module is safe or not. This is presented in the following chapter.



s0
bp
frame
f_s
R
f_s
RW

$$\textsc{AAssign}\;\frac{instr(\iota) = \lfloor r = e \rfloor}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho[r \mapsto \llbracket e \rrbracket_\rho], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{AStD}\;\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = (\mathsf{DATA}, o) \quad \gamma(o) \subseteq [0; d_s[}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{AStF}\;\frac{instr(\iota) = \lfloor [e_1] = e_2 \rfloor \quad \llbracket e_1 \rrbracket_\rho = (\mathsf{BP}, o) \quad \mathit{off} \in \gamma(o) \quad \gamma(o) \subseteq]- \mid \phi \mid ; 0]}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho, \phi[\mathit{off} \mapsto \llbracket e_2 \rrbracket_\rho]^{(1)}, \iota^+ \rangle}$$

$$\textsc{ALdD}\;\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = (\mathsf{DATA}, o) \quad \gamma(o) \subseteq [0; d_s[}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho[r \mapsto (\mathsf{ZERO}, \top)], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{ALdF}\;\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = (\mathsf{BP}, o) \quad \mathit{off} \in \gamma(o) \quad \gamma(o) \subseteq]- \mid \phi \mid ; 0]}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho[r \mapsto \phi(\mathit{off})], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{ALdS}\;\frac{instr(\iota) = \lfloor r = [e] \rfloor \quad \llbracket e \rrbracket_\rho = (\mathsf{BP}, o) \quad \gamma(o) \subseteq]0; \mid \phi \mid [}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho[r \mapsto (\mathsf{ZERO}, \top)], \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{ACall}\;\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad \gamma(\llbracket e \rrbracket_\rho) \subseteq \mathcal{F} \cup \mathcal{T} \quad \rho(\mathbf{esp}) = (\mathsf{BP}, o) \\ \mathit{off} \in \gamma(o) \quad \mid \phi_1 \mid = -\mathit{off} \quad \gamma(o) \subseteq] - f_s ; 0] \\ isret(\iota^+, \rho, \phi_1) \quad \rho \sim \rho' \quad \mid \phi_2 \mid = \mid \phi_2' \mid \end{array}}{\langle \rho, \phi_1 \cdot \phi_2^{(1)}, \iota \rangle \longrightarrow^\sharp \langle \rho', \phi_1 \cdot \phi_2'^{(1)}, \iota^+ \rangle}$$

$$\textsc{ARet}\;\frac{instr(\iota) = \lfloor \mathbf{ret}\ e \rfloor \quad preserve(\rho) \quad \llbracket e \rrbracket_\rho = (\mathsf{RET}, o) \quad \{0\} = \gamma(o)}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \blacksquare}$$

$$\textsc{ACont}\;\frac{instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad 0 \in \gamma(\llbracket e_1 \rrbracket_\rho)}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho, \phi^{(\beta)}, \iota^+ \rangle}$$

$$\textsc{AJump}\;\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{jmpif}\ e_1\ e_2 \rfloor \quad c \in \gamma(\llbracket e_1 \rrbracket_\rho) \quad c \neq 0 \\ \llbracket e_2 \rrbracket_\rho = (\mathsf{CODE}, o) \quad \iota_2 \in \gamma(o) + c_0 \quad \iota_2 \in \mathit{Code} \end{array}}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow^\sharp \langle \rho, \phi^{(\beta)}, \iota_2 \rangle}$$

$$\textsc{AHalt}\;\frac{instr(\iota) = \lfloor \mathbf{hlt} \rfloor}{\langle \rho, \phi^{(\beta)}, \iota \rangle \longrightarrow \blacksquare} \qquad \textsc{ACrash}\;\frac{}{\blacksquare \longrightarrow \blacksquare}$$

Figure 4.3 – Abstract semantics

Chapter 5

Experimenting with an Analyzer Implementation

5.1 REIL and Lifting

This thesis presents an analyzer of binary code and semantics of languages that describe how that binary code behaves. As a starting point, it is necessary to precisely define the semantics of a binary code, its behavior and what kind of behaviors are acceptable or not. However, the semantics of binaries is only documented in reference manuals written by processor vendors or manufacturers.

These manuals are huge (4618 pages for the x86_64 manual from Intel) and not necessarily complete: hidden instructions have been discovered through tools such as Sandsifter. [17] For instance, they found that `0f18xx` (where `xx` represents an arbitrary byte) was a valid instruction on a 2012 x86 processor, but was only documented after 2016. [16]

Since it is not realistic to write a complete semantics for an existing architecture by myself during my thesis, I decided to use existing semantics. And since my work requires that I write more than one semantics, I had to choose a concise one. Binary analysis tools often have the same problem and solve it by using an intermediate representation (IR) that has only a few instructions. Each processor instruction is decoded and *lifted* to a small snippet of IR code.

5.1.1 Intermediate Representations

Different tools use different intermediate representations. These representations are chosen to be as independent of the source binary code as possible, so as to support any processor type.

In binary analysis, the requirements of the intermediate language is that it should be short, make side effects visible and have the following commands: jumps, conditions, loops, storing in register, storing in memory and reading from memory. Each intermediate language also has a set of arithmetic operations, and might support more instructions (such as floating point instructions) or emulate them.

REIL

REIL (Reverse Engineering Intermediate Language) is an intermediate language designed by Zynamics for analyzing binary programs. It contains 17 different instructions that look like assembly language instructions. [32] An instruction can be an arithmetic operation, a conditional jump, a register or memory store or a load from memory. Each operand can be an immediate or the content of a register. The result of reading and operations is always stored in a register.

There is no limit to the amount of possible registers, and they are not typed. Some registers represent actual registers of the source architecture, and the rest are temporary registers, used for storing intermediate results during the execution of a source instruction.

Listing 5.1 shows the corresponding REIL program to the single `add eax, 0x400` (in binary: `05 00 04 00 00`) instruction on an x86 processor.

Listing 5.1 – Example REIL Lifting for `add eax, 0x400`

```
0.00  STR R_EAX:32,           ,  V_00:32
0.01  ADD  V_00:32,     400:32,  V_01:32
0.02  ADD  V_00:32,     400:32,  V_02:32
0.03   LT  V_02:32,    V_00:32,   R_CF:1 # Carry Flag is computed
0.04  AND  V_02:32,      ff:32,  V_04:32
0.05   OR  V_04:32,       0:32,   V_03:8
0.06  SHR   V_03:8,        7:8,   V_05:8
0.07  SHR   V_03:8,        6:8,   V_06:8
0.08  XOR   V_05:8,     V_06:8,   V_07:8
0.09  SHR   V_03:8,        5:8,   V_08:8
0.0a  SHR   V_03:8,        4:8,   V_09:8
0.0b  XOR   V_08:8,     V_09:8,   V_10:8
0.0c  XOR   V_07:8,     V_10:8,   V_11:8
0.0d  SHR   V_03:8,        3:8,   V_12:8
0.0e  SHR   V_03:8,        2:8,   V_13:8
0.0f  XOR   V_12:8,     V_13:8,   V_14:8
0.10  SHR   V_03:8,        1:8,   V_15:8
0.11  XOR   V_15:8,     V_03:8,   V_16:8
0.12  XOR   V_14:8,     V_16:8,   V_17:8
0.13  XOR   V_11:8,     V_17:8,   V_18:8
```

```
0.14  AND   V_18:8,          1:8,    V_20:8
0.15   OR   V_20:8,          0:8,    V_19:1
0.16  NOT   V_19:1,             ,    R_PF:1 # Parity Flag is computed
0.17  XOR  V_00:32,       400:32,  V_21:32
0.18  XOR  V_02:32,      V_21:32,  V_22:32
0.19  AND    10:32,      V_22:32,  V_23:32
0.1a   EQ     1:32,      V_23:32,    R_AF:1 # Adjust Flag is computed
0.1b   EQ  V_02:32,         0:32,    R_ZF:1 # Zero Flag is computed
0.1c  SHR  V_02:32,        1f:32,  V_24:32
0.1d  AND     1:32,      V_24:32,  V_25:32
0.1e   EQ     1:32,      V_25:32,    R_SF:1 # Sign Flag is computed
0.1f  XOR   400:32, ffffffff:32,  V_26:32
0.20  XOR  V_00:32,      V_26:32,  V_27:32
0.21  XOR  V_00:32,      V_02:32,  V_28:32
0.22  AND  V_27:32,      V_28:32,  V_29:32
0.23  SHR  V_29:32,        1f:32,  V_30:32
0.24  AND     1:32,      V_30:32,  V_31:32
0.25   EQ     1:32,      V_31:32,    R_OF:1 # Overflow Flag is computed
0.26  STR  V_01:32,             , R_EAX:32 # Result is computed
```

Vex

As REIL, Vex is an intermediate representation without side-effects, with a lifter that understands multiple architectures. The representation works with basic blocks that each correspond to an instruction in the binary. Each block is composed of statements, that modify the state of the machine. A statement manipulates expressions (constant values, results of operations, memory loads, register reads) and temporary variables (internal registers). Temporary variables are typed with the size, sign and kind (float, int) of its value.

Registers are modeled as a separate memory space, where a specific offset corresponds to a specific register in the source architecture. Compared to REIL, Vex has many more operations and statements, that make it more concise and easier to work with in terms of engineering, but it also makes it more difficult to reason about, since many cases must be taken into account.

It was developed by Valgrind and later used in angr [36], a binary analysis program.

Listing 5.2 shows the corresponding VEX program to the single `add eax, 0x400` instruction on an x86 processor, after being optimized.

Listing 5.2 – Example VEX Lifting for `add eax, 0x400`

```
t0:I32   t1:I32   t2:I32   t3:I32
```

```
------ IMark(0x0, 5, 0) ------
t0 = GET:I32(8)
t2 = Add32(t0,0x400:I32)
PUT(40) = 0x3:I32
PUT(44) = t0
PUT(48) = 0x400:I32
PUT(52) = 0x0:I32
PUT(8) = t2
PUT(68) = 0x5:I32
```

BAP IR

Bap [10], for Binary Analysis Platform is a tool for writing static analyzes on binaries. It uses its own intermediate representation, called the BAP IR. The BAP IR is close to VEX, but features explicit typing and register naming. As with VEX, temporaries and registers are separate.

Listing 6.1 shows the corresponding BAP IR program to the single `add eax, 0x400` instruction on an x86 processor.

Listing 5.3 – Example BAP Lifting for `add eax, 0x400`

```
var T_32t0:reg32_t;
var T_32t1:reg32_t;
var T_32t2:reg32_t;
var T_32t3:reg32_t;
T_32t0:reg32_t = R_EAX:reg32_t;
T_32t2:reg32_t = (T_32t0:reg32_t+1024:reg32_t);
// eflags thunk: add
var T_0:reg32_t;
T_0:reg32_t = (T_32t0:reg32_t+1024:reg32_t);
R_CF:reg1_t = (T_0:reg32_t<T_32t0:reg32_t);
var T_1:reg8_t;
T_1:reg8_t = cast(T_0:reg32_t)L:reg8_t;
R_PF:reg1_t = (!cast(((((T_1:reg8_t>>7:reg8_t)^(T_1:reg8_t>>6:reg8_t))^
  ((T_1:reg8_t>>5:reg8_t)^(T_1:reg8_t>>4:reg8_t)))^(((T_1:reg8_t>>3:reg8_t)^
  (T_1:reg8_t>>2:reg8_t))^((T_1:reg8_t>>1:reg8_t)^T_1:reg8_t))))L:reg1_t);
R_AF:reg1_t = (1:reg32_t==(16:reg32_t&(T_0:reg32_t^(T_32t0:reg32_t^1024:reg32_t))));
R_ZF:reg1_t = (T_0:reg32_t==0:reg32_t);
R_SF:reg1_t = (1:reg32_t==(1:reg32_t&(T_0:reg32_t>>31:reg32_t)));
R_OF:reg1_t = (1:reg32_t==(1:reg32_t&(((T_32t0:reg32_t^
  (1024:reg32_t^4294967295:reg32_t))&(T_32t0:reg32_t^T_0:reg32_t))>>31:reg32_t)));
R_EAX:reg32_t = T_32t2:reg32_t;
```

5.1.2 Comparison

Different intermediate languages exist for the purpose of analyzing binaries. They have different features, but some of them are common to every language. For each language, a *lifter* is used to translate assembly instructions into a list of one or more intermediate instructions. The intermediate language also has a way to represent actual registers, but also a way to represent temporary values that are neither registers nor the main memory.

A lifter can also add an optimization pass, to simplify the resulting instruction listing. For instance, a `xor eax, eax` can be translated as a simple assignment of 0 to `eax`, and of fixed values to flags. We will see in Chapter 7, that we later will analyze multithreaded modules. Optimizations do not always play well with multithreading as they may introduce races, or remove some behaviors that were possible before the optimization.

A language can either represent temporaries as a separate memory space, or as part of the registers. In our own language, we have decided to use the registers to represent both temporaries and actual hardware registers. This simplifies the assignment and evaluation logics.

A language is either dynamically or statically typed. Static typing improves the reliability of the lifting, because no type incompatibility can be undetected, but it has negligible impact on the analysis itself.

5.2 Implementation

A static analysis was implemented on top of the BinCAT binary code analysis toolkit [9]. BinCat is based on the theory of abstract interpretation, comes with a REIL lifter and provides support for forward and backward analysis, domains and it is easily extensible. Its main goal is to provide better analysis to IDA, a graphical binary disassembler. It is composed of a python server that communicates with IDA to get binaries to analyze and return analysis result so they can be shown to the user, and an OCaml backend that implements the analysis.

The OCaml backend is completely disconnected from the IDA frontend, which makes it easy to use it as a library and extend it with new domains, disconnect some domains (especially taint analysis that we do not need), and slightly change the elf decoder to recognize the use of the sfi external variable.

After changing the program to accept a single binary with no configuration as its only parameter, and run the analyzer once per function, we get a working SFI analyzer. First, the SFI analyzer reconstructs the structure of the binary and in particular partitions the code into separate functions, thanks to its elf decoding module, and transforms the binary instructions into the REIL [18] intermediate representation, thanks to the lifter. Second, each previously identified function is analyzed separately, using the abstraction described in Section 4.4. For each function, the analysis checks that all the intra-procedural jumps stay within the current function and that the abstract semantics never blocks.

The analysis also checks that all calls are towards previously identified entry points thus validating *a posteriori* that the initial partition of the code into distinct functions is indeed correct.

5.2.1 Relation between semantics and REIL

REIL maps back to the semantics of our languages very nicely: A REIL program can be transformed to our reduced semantics: any REIL instruction acts on a set of three registers: two inputs and one output. Some operations, STM, LDM, JMP, HLT map one-to-one with our language, while other arithmetic operations (ADD, SUB, etc) can be mapped to a register store. For instance, `ADD r1, r2, r3` can be translated into `r3 = r1 + r2`. Since REIL easily maps back to a subset of the language constructs we can use in our language, we can see our analyzer as an implementation of an analyzer of a subset of our language. Instead of analyzing any kind of program, we only analyze programs whose register store instruction has only one operation, and where jumps, memory loads and stores do not contain any operation.

5.2.2 Implementation Details

Compared to BinCat, the implementation has a few differences listed here. First, the main was modify to accept a binary file name as its only argument, assuming we were analyzing a ELF binary. Then, the REIL language was slightly modified to add a notion of a symbolic sandbox address, and domains were fixed to support it. New domains were added for the purpose of the analysis:

— The environment domain, as a Functor of a numerical domain, that represents the environment ρ of the program,

— A numerical domain interface,

— A symbolic domain, as a functor of a numerical domain, that represents our symbolic values (a pair of a label and a value),

— A value domain, that implements the bitfield domain.

The decoder was modified to detect the use of the sfi sandbox, which was implemented as a relocation to a separate section. A relocation is a mechanism used to get relocatable binaries: instead of directly storing a memory address as an immediate value in the code, a special table is used for the value to be set only at load time, when the loader has more information on actual section position in memory, or for dereferencing external functions and variables. Relocations are normally transformed in BinCat to an immediate value, but SFI relocations are transformed into an addition of the sfi symbolic address and an immediate offset.

The sandbox is not represented as a memory region. Instead, reads and writes to the sandbox are simply modeled by a constant function that returns $\top$ and a no-op, respectively. The analyzer already throws an error when the program might write or read an unmapped memory region.

5.2.3 Tests

The analysis has been tested on three test suites: correctly sandboxed programs, incorrectly sandboxed programs, and optimized, correctly sandboxed programs. The first test suite is built by hand and is composed of very small programs. Figure 5.1 shows how the sandboxing is implemented in practice: a memory region, `sfi$id` is declared and will be used as the sandbox region. We align it properly. The sandboxing operation consists of a logical and that drops the high-order bits (the sandbox tag) and adds the address of the sandbox (the correct tag). It is declared static inline to ensure that the compiler will always put the sandboxing operation in place, instead of calling a function. Indeed, our analyzer is intra-procedural, so it would not be able to find a pointer was sandboxed if this was in the body of a separate function.

The second test suite is similarly built by hand. It is composed of other very small tests that are supposed to be rejected. Each of them implements an incorrect security property for various reasons and they should all be rejected.

```
char sfi$id[0x1000000] __attribute__ ((aligned (0x1000000)));

static inline int sandbox(int ptr, int sz) {
  return (ptr & (0x1000000 - sz)) + (int)sfi$id;
}
```

Figure 5.1 – Common sandboxing code

Finally, the last test is built by compiling programs that are part of the CompCert test suite with a modified version of CompCert that includes sandboxing instructions. Because these binaries are correct by construction, the verifier should accept all of them.

In this last experiment, we have tested 10 programs, composed of 51 functions in total. 41 functions are verified in under 100ms, 9 functions are verified in less than 300ms and 1 function is verified in 3.5s. This last function occurs in sha3.c, and is responsible for the program being verified in 3.5s. This file is 200LoC long, while another file, aes.c, is 1.5KLoC long, composed of 7 functions and takes only 1s to validate. This suggests that the time complexity depends on the number of nested loops rather than on the size of the code to verify.

5.2.4 Verifying Incorrect Programs

The second test suite for catching incorrect programs has been obtained by compiling incorrectly sandboxed programs with gcc and verifying they do not pass our verification. Each test in the suite aims at a different error: returning before the end of a function, writing above and below the frame, stack or sandbox and bypassing the guard zones. Overall, the test suite contains 9 programs and all are correctly identified as violating the sandbox property. Some of these programs can be found in Fig. 5.2.

5.2.5 Verifying Redundant Sandboxing Elimination

We have also evaluated the ability of the analysis to verify programs where redundant sandboxing instructions have been optimized away. For instance, the sandboxing of consecutive accesses to an array can be factorized and implemented by a single sandboxing instruction. In addition to masking the most significant bits, this sandboxing instruction also zeroes out several least significant bits thus aligning the base address of the array. The reasoning is that if an address a of the sandbox is aligned on k bits we

```
asm("sub $5000000, %esp\n\t"
     "push $1");
```

(a) Attempt to write below the stack

```
data[-5] = 0;
```

(b) Attempt to write outside of the sandbox

```
int f(int *e) {
  int i;
  asm("push $main\n\t"
       "mov %1, %%ebp\n\t"
       "add %%ebp, (%%esp)\n\t"
       "ret"
       : "=r"(i)
       : "r"(*((int *)(sandbox(e, 4))))
       : "%ebp");
  return i+5;
}
```

(c) Attempt to return before the end

Figure 5.2 – Violation of sandboxing

have that $a+i$ for $i \in [0, 2^k-1]$ is also in the sandbox. We have sandboxed the programs manually and compiled them with gcc and verified whether they passed our verification. Since we use an intra-procedural analysis, no inter-procedural optimisation can be performed on sandboxing. Our numerical domain is able to model alignment constraints and the analysis accepts programs where consecutive writes are protected by the previous sandboxing operation. Yet, the analysis rejects programs where the sandboxing instruction is factored outside loops because the information inferred about the loop bound is currently not precise enough. More precision could be obtained by using more sophisticated numerical domains. An example program that fails our verification is given in Fig. 5.3.

```
char *a = (sfi + (t & 0b11111000)
for(char i=0; i<5; i++) {
  a[i] = i;
}
```

Figure 5.3 – Optimising array accesses in a loop

The last test suite evaluates the ability of the analysis to verify programs where redundant sandboxing has been optimized away. It has been obtained by compiling correctly sandboxed programs with gcc and verifying whether they pass our verification. Since we use an intra-procedural analysis, no inter-procedural optimisation can be performed on sandboxing. Our tests show that we are able to avoid sandboxing every write to an array. Because values may actually be bigger than one byte, the sand-

```
void f(char *a) {
    char *b = (char *)sandbox(
        (int)a, 32);
    b[0] = 1;
    b[1] = 1;
    b[2] = 7;
    b[5] = 3;
}
```

(a) One sandboxing for successive writes

```
void f(char *a) {
    // Ensures the array is
    // aligned to 32 bytes
    char *b = (char *)sandbox(
        (int)a, 32);
    for(int i=0; i<5; i++) {
        b[i] = i;
    }
}
```

(b) One sandboxing before a loop

Figure 5.4 – Examples of optimizations

boxing instructions align the address to a certain amount of bytes. By aligning arrays sufficiently, we can use just one sandboxing for a pointer to the array and later use offsets. Unrolled assignments are successfully detected as being sandboxed in that way. However, the current analyzer does not exploit the loop guards, so assignments in a loop are wrongly detected as unsafe.

Since we propose an intra-procedural analysis to ensure the security property, a fundamental weak point of our analyzer is its lack of knowledge about function calls. For instance, programmers sometimes use the `alloca(size_t size)` function. This function allocates `size` bytes on the stack and returns a pointer to this memory space.

The most common implementation of this function is an inline function that moves the stack pointer by the amount specified by its argument. The most common use for this function is to pass a statically unknown size though, so our analysis will most likely not be able to compute a bound on the new stack pointer, hence rejecting the module.

Finally, the following table shows time statistics for each function when running the analysis on a Intel(R) Core(TM) i7-6600U CPU at 2.60GHz processor. The analysis uses only one thread of the processor.

program	function	mean analysis time	standard deviation
aes	do_bench_749	26ms	3ms
aes	do_test_742	18ms	2ms
aes	main	8ms	1ms
aes	rijndaelDecrypt	139ms	14ms
aes	rijndaelEncrypt	200ms	17ms
aes	rijndaelKeySetupDec	240ms	13ms
aes	rijndaelKeySetupEnc	161ms	8ms
chomp	copy_data	4ms	0ms

chomp	dump_list	2ms	0ms
chomp	dump_play	2ms	0ms
chomp	equal_data	54ms	3ms
chomp	get_good_move	10ms	1ms
chomp	get_real_move	6ms	1ms
chomp	get_value	5ms	0ms
chomp	in_wanted	12ms	1ms
chomp	main	13ms	1ms
chomp	make_data	5ms	0ms
chomp	make_list	56ms	5ms
chomp	make_play	23ms	2ms
chomp	melt_data	9ms	1ms
chomp	next_data	6ms	1ms
chomp	valid_data	6ms	0ms
chomp	where	4ms	0ms
fannkuch	fannkuch_586	174ms	8ms
fannkuch	main	3ms	0ms
fib	fib	3ms	0ms
fib	main	3ms	0ms
lists	buildlist	4ms	0ms
lists	checklist	60ms	2ms
lists	main	20ms	2ms
lists	reverse_inplace	2ms	0ms
lists	reverselist	10ms	1ms
nsievebits	main	5ms	0ms
nsievebits	nsieve_664	138ms	6ms
nsievebits	test_671	15ms	1ms
nsieve	main	26ms	2ms
nsieve	nsieve_664	107ms	4ms
qsort	cmpint	2ms	0ms
qsort	main	54ms	3ms
qsort	quicksort	65ms	3ms
sha1	do_bench_711	6ms	0ms
sha1	do_test_701	6ms	0ms

sha1	main	5ms	0ms
sha1	SHA1_add_data	16ms	1ms
sha1	SHA1_copy_and_swap_668	12ms	1ms
sha1	SHA1_finish	9ms	1ms
sha1	SHA1_init	5ms	0ms
sha1	SHA1_transform_675	126ms	6ms
sha3	keccak	137ms	9ms
sha3	keccakf	3508ms	191ms
sha3	main	17ms	1ms

Most of the functions are analyzed in a few milliseconds, but some of them take a few seconds. The functions that take the longest to analyze (up to 3 seconds), are functions with loops, and the longest one is a function with two nested loops. The most contributing factor is therefore not the number of instructions of a function, but rather the level of loops, as each loop content is analyzed multiple times. Since the analysis of each function is independent from one another, we could implement a multithreaded analysis, which would reduce the time taken to analyze a complete program.

Compared to more standard SFI verifiers, we rely on abstract interpretation and analyze more finely the instructions of the binary. Other SFI implementations are simple linear passes and have therefore better results: typically a program is analyzed in a few milliseconds. This is comparable to our results when a function is purely linear.

Our implementation uses a finite domain for values, so it does not implement a widening. Implementing a widening, such as immediately reaching top for values that changed between two passes on the same instruction, would help with analysis speed, but would reduce the power of the analyzer, and we would not be able to detect and accept as much optimizations as the current implementation does. It could be interesting in practice to run the analyzer with a widening on every function, and if it failed to prove correctness of a function, run the more costly analysis on it.

PART III

Extending SFI to Thread-level Isolation

CHAPTER 6

WEAK MEMORY MODELS

6.1 An Introduction to Concurrency

Up until now, we have only seen an analyzer for a single-threaded application and untrusted module. However, in practice, applications are usually multithreaded. If we stopped there, it would have meant that multithreaded applications with untrusted modules might be insecure. In this part, we will first see how to model multithreaded programs and introduce a new multithreaded concrete semantics for REIL. Then, we will use the same methodology as before to show property preservation on more and more abstract semantics. We will finally see that this methodology allows us to use the same intermediate semantics as with the single-threaded implementation, which allows us to use the same abstract semantics, and the same analyzer.

6.1.1 Interleaving Semantics

A first idea when we want to introduce multithreading in a semantics is to reuse the same semantics as before, but instead of a single state, have as many states as there are threads in the program, for thread-local values (registers, program counter, . . .) and a separate state for global values (memory, . . .). Then, each execution step of the program is the execution of one of these threads, non-deterministically. This models the fact that each thread executes independently of the others: any thread can advance one step at any time, but they still have global effects on other threads through a global memory.

The following program is a C source code for a very simple concurrent program example. In every iteration of the main loop, it zeroes out four variables, x, y, $r1$ and $r2$, creates two threads that each load the value of x and y in $r1$ and $r2$ before modifying the other (x or y). Then, the loop waits for the threads to finish and prints the value contained in $r1$ and $r2$.

Listing 6.1 – Example multithreaded program

```c
#include <pthread.h>
#include <stdio.h>

int x = 0, y = 0;
int r1 = 0, r2 = 0;

/* this function is run by the first thread */
void *inc_x(void *args) {
  x = 1;
  r1 = y;
  return NULL;
}

/* this function is run by the second thread */
void *inc_y(void *args) {
  y = 1;
  r2 = x;
  return NULL;
}

int main() {
  r1 = 1;
  while(r1 != 1 || r2 != 1) {
    pthread_t inc_x_thread, inc_y_thread;
    /* set the initial values of x and y */
    r1 = r2 = x = y = 0;

    pthread_create(&inc_x_thread, NULL, inc_x, NULL);
    pthread_create(&inc_y_thread, NULL, inc_y, NULL);

    /* wait for the first thread to finish */
    if(pthread_join(inc_x_thread, NULL)) {
      fprintf(stderr, "Error␣joining␣thread\n");
      return 2;
    }
    /* wait for the second thread to finish */
    if(pthread_join(inc_y_thread, NULL)) {
      fprintf(stderr, "Error␣joining␣thread\n");
      return 2;
    }
    printf("r1:␣%d,␣r2:␣%d\n", r1, r2);
  }
  return 0;
}
```

If we copy this code in a file, *test_wmm.c* and compile it with `gcc -O0 test_wmm.c -lpthread -O test_wmm`, this will create a *test_wmm* executable. Before running it, let us ask ourself: does this program even terminates?

To answer that question, we model the behavior of the program in the following table. The first line shows the initial state: x and y are shared variables, initially set to 0. Then the program executes. It is composed of two threads, with two instructions each. An instruction is labeled with a letter and they are all C-style assignments here. The last line is a question on the result.

x = y = 0	
(a) x = 1	(c) y = 1
(b) r1 = y	(d) r2 = x
Can r1 = r2 = 0?	

Now, what are the possible interleavings? There are six possibilities:

— a, b, c, d: first, x is set to 1, then $r1$ is set to the value of y, 0, then y to 1, then $r2$ to the value of x which has just been modified, so 1. In the end, $r1$ = 0 and $r2$ = 1.

— a, c, b, d: first, x and y are set to 1, then $r1$ and $r2$ are set to these values. In the end, $r1$ = $r2$ = 1.

— a, c, d, b: first, x and y are set to 1, then $r1$ and $r2$ are set to these values. In the end, $r1$ = $r2$ = 1.

— c, d, a, b: first, y is set to 1, then $r2$ is set to the value of x, 0, then x to 1, then $r1$ to the value of y which has just been modified, so 1. In the end, $r1$ = 1 and $r2$ = 0.

— c, a, d, b: first, y and x are set to 1, then $r1$ and $r2$ are set to these values. In the end, $r1$ = $r2$ = 1.

— c, a, d, b: first, y and x are set to 1, then $r1$ and $r2$ are set to these values. In the end, $r1$ = $r2$ = 1.

We have just listed all the possible interleavings, and none of them can result in $r1 = r2 = 0$, which is the condition for the loop to stop. So we can confidently conclude that the program we just compiled cannot terminate.

Let's run the program anyway: `./test_wmm`.

As we expected, the program prints a lot of lines with variations of results that we previously predicted. But after some time, it stops, with no error message and with this mysterious last line: `r1: 0, r2: 0`. What happened? We have just encountered a behavior that cannot be explained by our interleaving semantics. Some processors, like the *x86_64* one that was used in this experiment allow more behaviors than the interleaving semantics. To account for this excess of behaviors, we can use so-called weak memory models (or wmm for short, hence the name of the file before).

The weak memory behavior is not so rare. Here is the result of running `for i in $(seq 1 100); do ./test_wmm | wc -l; done > test_wmm.res` followed by a quick analysis using R with `R -q -e "x <- read.csv('test_wmm.res', header = F); summary(x); sd(x[ , 1])"`. It took a minute and a half to collect the data:

Min.	2
1st Qu.	4300
Median	15912
Mean	29279
3rd Qu.	35922
Max.	209758

This means that it took on average 29,000 tries for the weak memory behavior to happen and 16,000 times was the median. In average, it also took less than a second to exhibit the behavior.

6.1.2 Weak Memory Models

There is an explanation to this behavior: the processor has its own memory cache, and memory writes do not always propagate in time to another processor. In our example, it could be the case that the first thread is executed, then the second, but the write to x is only propagated after setting the value of $r2$, and that is how the result can be 0 in both variables.

Also note that this kind of behavior can be introduced in the compilation process: if the compiler decides that a function will execute faster, it can reorder unrelated instructions as it sees fit. It is not the case here, as we have disabled all optimizations. A decompilation of the program also shows that the order of execution in both threads is the same in assembly and in C.

Instead of using a somewhat ad-hoc explanation to the observed behavior, it is best to have a mathematical model that can explain it. Programming languages in weak memory models can also have their own semantics, and many different semantics exist, because there are many different memory models. We can also compare memory models, and we say that a memory model is weaker (resp. stronger) than another memory model if it allows more (resp. less) behaviors than the other. Some memory models are not comparable as they might both allow behaviors the other prohibits.

Processors use different memory models depending on their architecture, vendor, family and series. Vendors are usually very defensive in the kind of information they disclose about their products, and the exact memory model is not officially known for most of them. Instead of trying to build one memory model for each physical processor series, researchers have focused on finding the strongest memory model that allows any behavior of an architecture or processor family.

Litmus tests have been designed to check for certain categories of weak behaviors. A litmus test is a minimal test that checks whether a certain behavior happens on a processor or not. If the litmus test is positive, a certain weak behavior is allowed, otherwise it is forbidden, which teaches us some of the underlying constraints.

An example of a litmus test is the *store buffering* test. It is the test we have presented in the introduction to concurrency. The test checks whether the model uses buffered writes that would allow the behavior we observed. The litmus test was successful on the x86 processor it was run on.

Another interesting litmus test is known as load buffering. A load buffering happens when a thread registers a load action, but only processes it later. It could happen if read actions are buffered, or reordered after writes.

x = y = 0	
(a) r1 = y (b) x = 1	(c) r2 = x (d) y = 1
Can r1 = r2 = 1?	

This time, the litmus test is negative on our x86 processor.

The next sections will present a few common memory models and tools used to evaluate them.

Total Store Ordering

The *total store ordering* (or TSO) memory model is a model where each processor has a local buffer for writes. It can be modeled as a small-steps semantics, where local states contain a write buffer. At each step, a thread can either write the last value of its buffer to the global memory, or execute one instruction. In that last case, it may need to read from memory: it reads the most recent write in its buffer to the same address and if there is no value at that address in its buffer, from the global memory. If it needs to write to memory, it adds the address and value in its buffer.

A simplistic version of that model is shown below. Sevcik et al. have implemented a complete implementation of x86-TSO in CompCert [34]. We will not explain it in details, but the following semantics is a very simplified version that highlights the differences between TSO and interleaving semantics. In this semantics, ρ is an environment: it maps registers to values. ι is the instruction pointer for a thread and β is the thread-local buffer. $\langle \iota, \beta, \rho \rangle$ is a thread-local state, and the global state is composed of a list of local threads and a global memory, which maps addresses to values. On each step, a thread can execute or write its last value of its buffer to memory:

$$\frac{m \vdash s_i \rightarrow s_i'}{\langle \langle s_1, \dots, s_i, \dots, s_n \rangle, m \rangle \rightarrow \langle \langle s_1, \dots, s_i', \dots, s_n \rangle, m \rangle}$$

$$\frac{}{\langle \langle s_1, \dots, \langle \iota, \beta :: (a, v), \rho \rangle, \dots, s_n \rangle, m \rangle \rightarrow \langle \langle s_1, \dots, \langle \iota, \beta, \rho \rangle, \dots, s_n \rangle, m[a \leftarrow v] \rangle}$$

Then, the semantics for a thread-local step is the same as for a single threaded semantics, except for memory reads and writes, which need to take into account β and m. Here is the memory read inference system, where $\beta, m \vdash_a v$ means the value v is read at address a, either from the local buffer β or from the main memory m:

$$\frac{}{\emptyset, m \vdash_a m[a]} \qquad \frac{}{(a, v) :: \beta, m \vdash_a v} \qquad \frac{\beta, m \vdash_a v \quad a \neq a'}{(a', v') :: \beta, m \vdash_a v}$$

And the memory write simply adds an address-value tuple at the beginning of the buffer. Here is how the store buffering litmus test could be satisfied with this semantics.

As we have seen before, the first thread is executed fully, but the write is buffered, and the second thread is executed fully before the write propagates to the global memory:

$$\langle(\langle(a), \beta, \rho\rangle, \langle(c), \beta, \rho\rangle), m\rangle \rightarrow$$

The first thread executes `(a) x = 1` and needs to write to memory location x. Since writes are buffered, the write happens in the local buffer of the first thread:

$$\langle(\langle(b), (x, 1) :: \beta, \rho\rangle, \langle(c), \beta, \rho\rangle), m\rangle \rightarrow$$

Then, the second thread executes `(b) y = 1` and needs to write to memory location y. Since writes are buffered, the write happens in the local buffer of the second thread:

$$\langle(\langle(b), (x, 1) :: \beta, \rho\rangle, \langle(d), (y, 1) :: \beta, \rho\rangle), m\rangle \rightarrow$$

Then, the first thread resumes and executes its last instruction, `(c) r1 = y`, which reads from memory location y. Since y is not in its local buffer, it reads its value from the global memory to which the previous write did not propagate yet. So it reads the value 0:

$$\langle(\langle(end), (x, 1) :: \beta, \rho[r1 \leftarrow 0]\rangle, \langle(d), (y, 1) :: \beta, \rho\rangle), m\rangle \rightarrow$$

The write from the second thread now propagates to the global memory. Its buffer is emptied and memory is updated with the new value:

$$\langle(\langle(end), (x, 1) :: \beta, \rho[r1 \leftarrow 0]\rangle, \langle(d), \beta, \rho\rangle), m[y \leftarrow 1]\rangle \rightarrow$$

Finally, the second thread executes its last instruction, `(d) r2 = x`, which reads the value at memory location x. Since it is not in its local buffer, it reads the value from memory, where the write from the first thread did not propagate yet. So it reads the value 0:

$$\langle(\langle(end), (x, 1) :: \beta, \rho[r1 \leftarrow 0]\rangle, \langle(end), \beta, \rho[r2 \leftarrow 0]\rangle), m[y \leftarrow 1]\rangle \rightarrow \ldots$$

6.1.3 Axiomatic Weak Memory Models

Not all weak memory model have a small-steps semantics, as can be seen in the survey from Zhang et al. [44]. Instead, most memory models have an axiomatic semantics. In these semantics, an execution is a set of events (reads, writes, fences) and relations between them. There are two parts: first the semantics defines the set of possible executions, a set of events sets. Then, the second part is a set of predicates on executions, that accept or reject executions. The semantics is then defined as the set of possible executions that are accepted by the predicates.

In their paper, Aglave and Cousot [2] show how to build the set of acceptable executions. They use events and states together in a small-steps semantics where memory is not modeled, and so memory reads can result in any value. This semantics is given by the set of language-specific constraints, and it gives us the set of possible executions. Then, they add more constraints to restrict the set of executions to actual executions. Following their approach, we first show how to build the set of possible executions of a program.

From a single-threaded small-steps semantics, one can introduce events at every step. So a memory read will yield a read event, a memory write will yield a write event, etc. We also need to strip the semantics from its memory, and allow reads to read any value, non deterministically. In section 6.2, we will see how we build this set of possible executions for our assembly-like language. This small-steps semantics relates states by execution order, but we are actually more interested in event relations. From the execution order, we are able to derive a program-order (po), an order on events of the same thread:

$$s \xrightarrow{e} s' \xrightarrow{e'} s'' \Rightarrow e \xrightarrow{po} e'$$

The second part of the semantics adds constraints on execution graphs by introducing other relations between events. In addition to program-order, we add a read-from (rf) relation which relates a write to a read. Read-from constraints include that the read and the write must be from and to the same location, and of the same value. Additionally, every read must be satisfied, and can only be satisfied by one write. This last condition allows us to talk about rf^{-1}.

Sequential Consistency

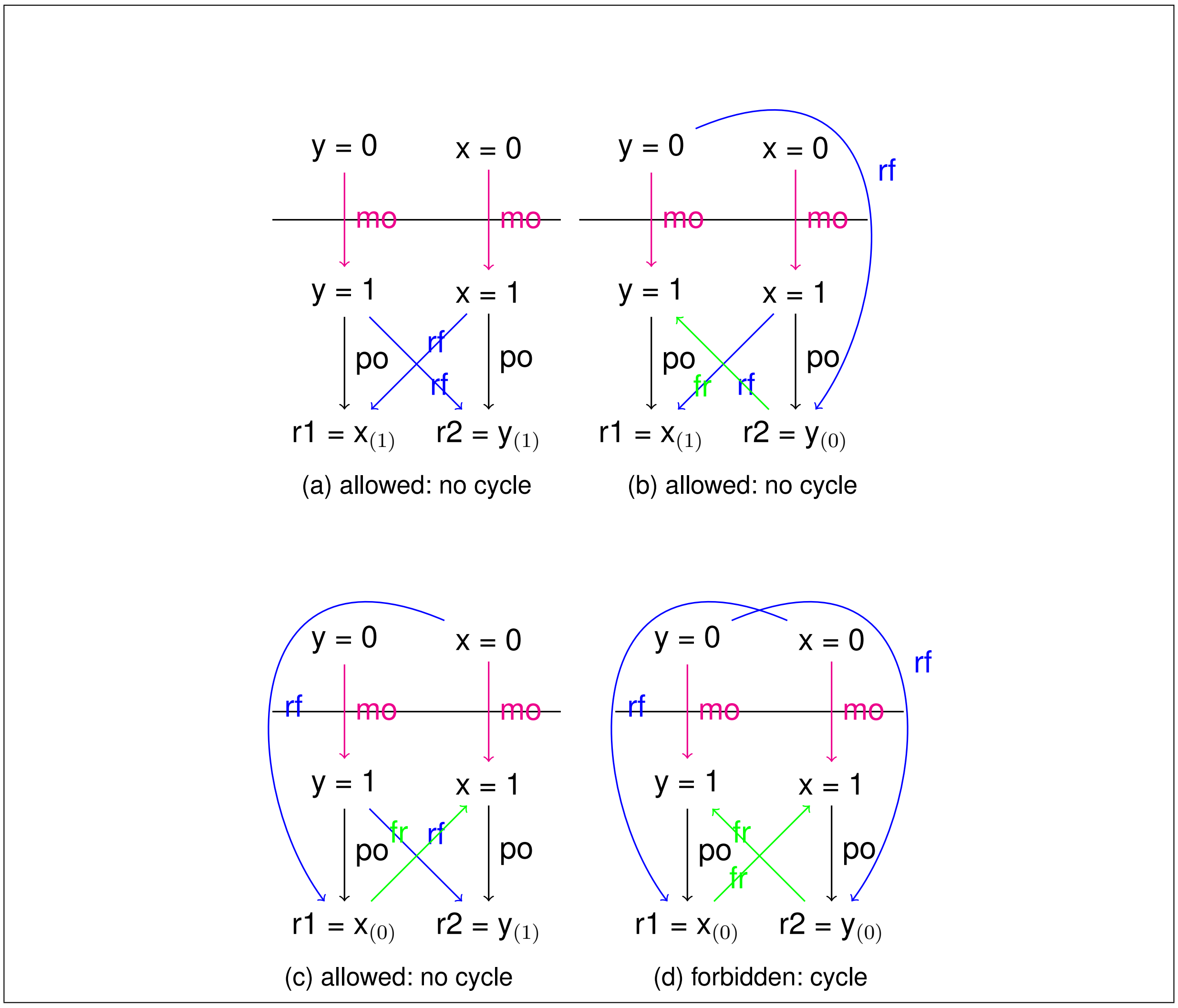
Sequential consistency can be defined as an axiomatic semantics. In addition to rf and po, it introduces two relations. The modification order, mo, is a total order on writes to the same location. The from-read relation, fr, is defined as $rf^{-1}; mo$.

Relation	Short name	Description
Program Order	po	Relates two events yielded by two consecutive instructions in the same thread
Read from	rf	Relates a write to a read, where the read reads the value from the write
Memory Order	mo	Relates two writes at the same location in a sort of "history" of the values held at that location.
From Read	fr	Relates a read to a write via $rf^{-1}; mo$. The write needs to happen after the read, otherwise the read would not have been able to read from the first write.

The sequential consistency memory model only allows executions where the po, rf, mo and fr relations together form an acyclic graph. Here are a four example outcomes for the store buffering litmus test, and reasons why they are accepted or rejected by the consistent memory model. In the graphs in Fig. 6.1, each event is represented by the instruction that generates it, and we also drew the values read by read events in small letters. Each type of relation is drawn with a different color and a label.

Total Store Ordering

The sequential consistency model is equivalent to the interleaving semantics, which makes it a strong memory model. Owens et al. [31] proposed a more complete semantics for x86-TSO. We are going to highlight a few important points here, but we are also going to simplify their model. They are not using fr, and a different definition for the modification order. In this model, the modification order is a total order on memory writes (independent of their location), and a partial order on memory reads. Additionally, when a read precedes a read or a write, and when a write precedes a write in program order, the two events are also related by memory order. Read-from relations also constrain the memory order: for a read-from to be valid, the write must be related



(a) allowed: no cycle

(b) allowed: no cycle

(c) allowed: no cycle

(d) forbidden: cycle

Figure 6.1 – Store buffering under sequential consistency

to the read in memory order and their must not be any write to the same location between the two events. Acceptable executions are executions where there is a memory order and read-from that satisfy these definitions.

The graph in Fig. 6.2 shows why the store-buffering litmus test is accepted by the x86-TSO model.

Figure 6.3 shows why the load-buffering litmus test is not accepted by the x86-TSO model. Each step is shown below. First, we select the only read-from relation that makes sense for that execution graph (a). Then, we need a total order on writes, that respects the po-order (b). There are different possibilities, but they are all equivalent to this one. Then, we need to have a modification order from reads to writes when they

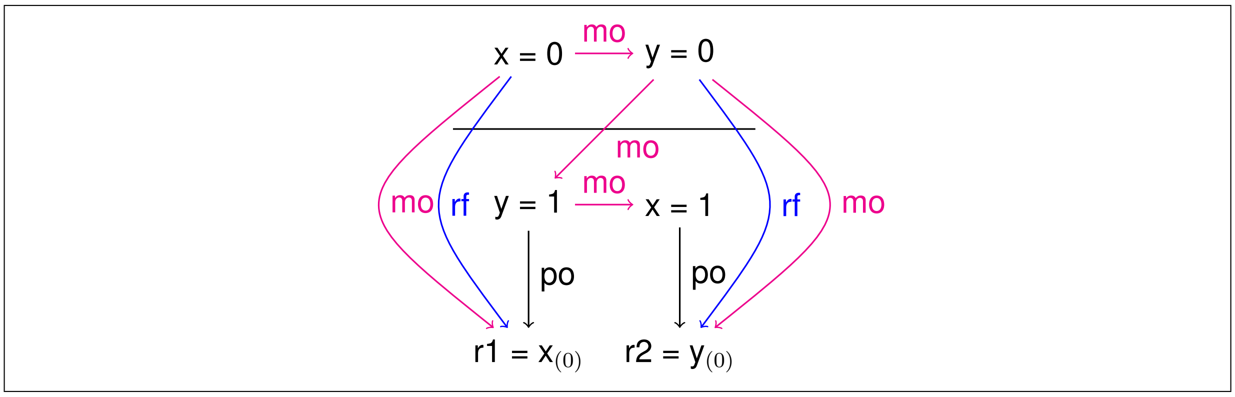


Figure 6.2 – Store buffering under TSO

follow program order (c). Finally, the write in rf needs to be before the read in mo, and must not contain another write to the same value (d). Whatever we choose here, we always create a cycle in mo, which therefore cannot be a partial order.

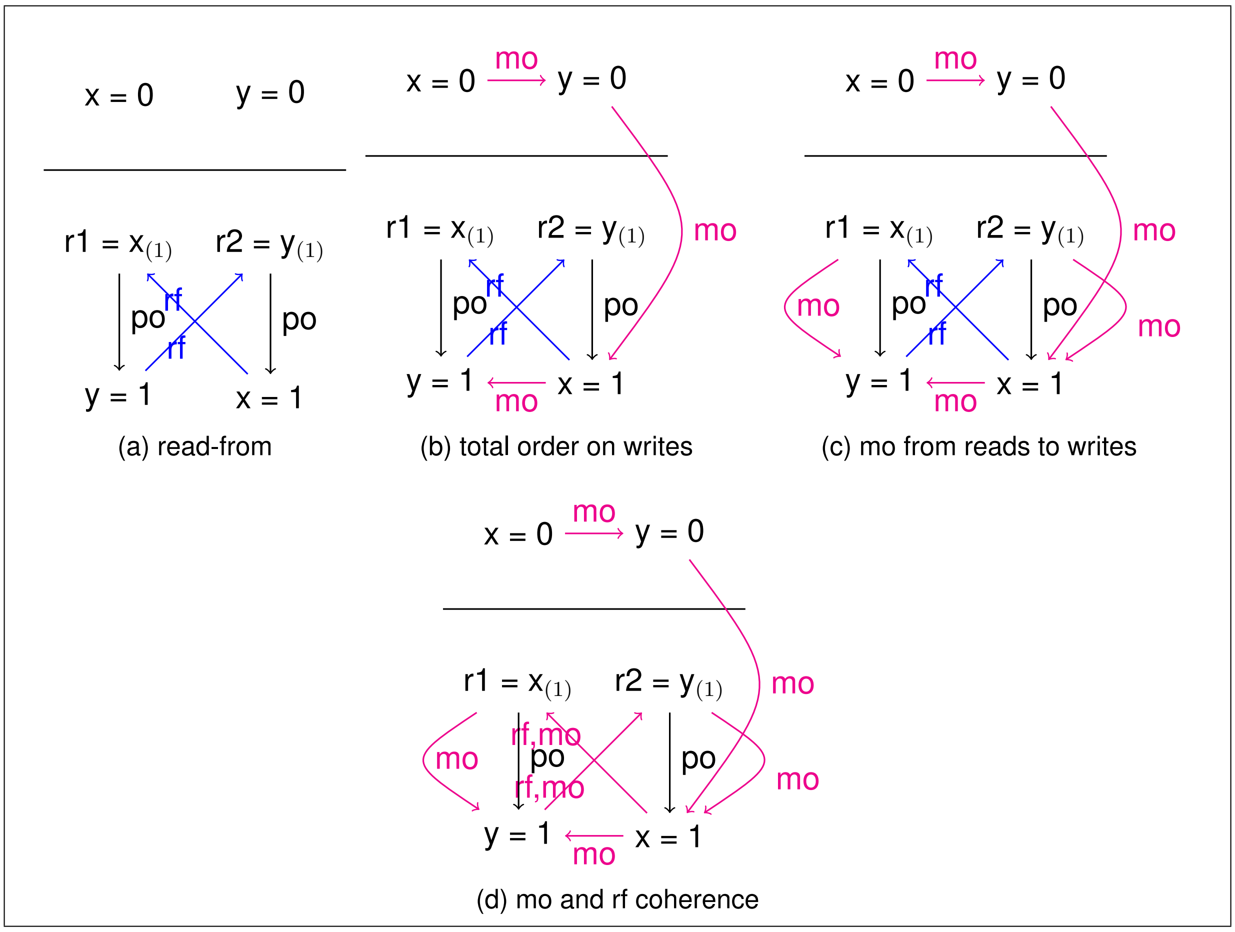


Figure 6.3 – Load buffering under TSO

6.2 REIL in a Weak Memory Model

We are now interested in defining a semantics for our small assembly-like language. Intuitively, we want to have the weakest possible semantics in order to over-approximate executions that happen on actual hardware. Here, we mostly follow Alglave and Cousot [2] to define our semantics.

6.2.1 Event

As we have seen before, our axiomatic semantics is based on a small-steps semantics that is composed of states and events. Intuitively, an event is yielded by the execution of an instruction. An event is therefore composed of:

— A type: $ty \in \{R, W, L, B, T\}$, where R means Read, W means Write, L means Local computation, B means Branch and T means Trusted. Read events are only yielded by memory reads, not register reads, and write events are only yielded by memory writes. A branch event is yielded by a conditional jump, a call and a return. A local event is yielded by every other instruction, except the halt instruction which does not yield any event. The Trusted event is special, because it corresponds to the execution of code in the trusted library. More precisely, it represents a set of Write events that happen because of the code in the trusted library.

— A unique event identifier, composed of tid, the id of the thread whose execution yielded the event, and uid, a unique id in the thread execution. The same uid can appear for events of other threads, but is unique inside a thread.

— An address, ι, the address of the instruction that yielded the event.

— A stack information, stk, that corresponds to that same information in the state that yielded the event (see later in the state description).

The two last items are not necessary, but will greatly help when defining the security property later.

Additionally, events contain a structure with useful information about the events that happened. A read and a write will contain the memory address that is accessed (note that no instruction can read and write to memory at the same time) and the value that is read or written. A branch event will contain the address of the following instruction, either the next instruction if the program does not jump, or the jump target if the program jumps. Finally, the local event will contain the result of the local computation. Our notation for events will be:

- $\langle R, tid, uid, \iota, stk, \langle addr, val\rangle\rangle$: the address $addr$ at which the value val is read.
- $\langle W, tid, uid, \iota, stk, \langle addr, val\rangle\rangle$: the address $addr$ to which the value val is written.
- $\langle B, tid, uid, \iota, stk, \langle typ, addr\rangle\rangle$: the address $addr$ to which the program jumps and the type typ (jump, call or ret) of instruction that yielded the event.
- $\langle L, tid, uid, \iota, stk, \langle val\rangle\rangle$: the value val that is computed.
- Trusted events do not have more data.

We also have a special event, ϵ_{start} that corresponds to the beginning of a thread, and to no instruction.

The Trusted event corresponds to the execution of code in the trusted library. As we made assumptions about the code in the trusted library, but did not actually show what it was, when we were defining the monothreaded semantics, we are not going to define precisely what happens in the trusted library. Intuitively, the trusted library can do a lot of things; that is why the trusted event represents more than one event and can represent any kind of read, writes, branches and local computations. We are not going to model all of them.

In fact, we are only going to model the trust event as a set of write events. Although the trusted code can read from memory, a memory read only adds more constraints to the system. Not representing them does not remove any accessible state, because trusted events can have almost arbitrary effects on the next state, and will help us prove the security of future semantics. Local and branches have no effect on other threads or on the execution of the thread, because trusted events already have almost arbitrary effects, so we do not represent them either.

We still want to restrict the power of the trusted event: it cannot write to every memory address, in particular it is forbidden to write to other threads' stack or to its own stack, above the base pointer (it can still write to its stack frame and below of course). This restriction is part of the assumption on the trusted library. Similarly to the assumption we made on the trusted library in monothreaded semantics, we also assume that it correctly handles its stack: when called it either returns to the callee, or calls a function in the module.

6.2.2 State

We now need to define states. In our axiomatic semantics, states are similar to the standard semantics of our language, with three major differences: each state is labeled with a unique id, composed of tid, a thread id, and uid, a unique id inside the thread. As with events, the unique id does not need to be unique between threads, only inside a thread. The second difference is that we do not represent memory. The last difference is the addition of stack information. This stack information is only used to describe the security property later on. This information will not interfere with the behavior of the semantics beyond the stack information itself.

A state $\langle \iota, tid, uid, \rho, stk \rangle\rangle$ is therefore composed of:

— An address ι of the next instruction to be evaluated.

— A thread id tid of the thread that is in that state.

— A unique identifier uid in the thread execution.

— An environment ρ mapping local registers to their value.

— A call stack, stk. It is a list of triples containing the return address, the initial environment and the caller's stack frame address.

The call stack contains triples in the form $\langle ret, \rho_0, bp_0 \rangle$.

6.2.3 Execution Traces

From states and events, we can now introduce *computations*. A computation ς is a set of anarchic traces, where each anarchic trace σ_t corresponds to the execution of thread t. An anarchic trace is composed of events and states, in this way:

$$\sigma_t \in \left(\xrightarrow{event} state \right)^*$$

A specific thread, numbered 0 is called the *prelude*. It does not correspond to an execution of an actual program thread, but to the initial memory. This prelude ensures that every read is satisfied by a write, either from the program, or from the initial memory.

An execution is also described by a read-from relation, *rf*. In the end, we have an execution $\xi = \langle \varsigma, rf \rangle$.

6.2.4 Well-formedness Conditions

The second part of the semantics is well-formedness conditions and memory model-related conditions. We specify here the conditions for an execution to be well-formed with regards to program P. We write $[\![P]\!]$ for the set of well-formed executions of P that also respect the memory model conditions. There are three kinds of conditions for an execution ξ to be well-formed.

Computation conditions

A computation $\varsigma = \tau_0 \times \prod_{t=0}^{n} \tau_t$ must respect the following conditions:

Definition 21 (start)**.** *Each trace τ must start with a unique (in that thread) ϵ_{start} event:*

$$\forall i \in [0,n], \overline{\tau_i}^0 = \epsilon_{start} \wedge \forall j \neq 0, \overline{\tau_i}^j \neq \epsilon_{start}$$

Definition 22 (identifiers)**.** *The unique identifiers of events must be unique in each thread:*

$$\forall i \in [0,n], \forall j, tid(\overline{\tau_i}^j) = i \wedge \forall j', uid(\overline{\tau_i}^j) = uid(\overline{\tau_i}^{j'}) \implies j = j'$$

Definition 23 (initialization)**.** *Each data location is initialized once in the prelude:*

$$\begin{array}{l} \forall addr \in [d_0; d_0 + d_s] \bigcup Td \bigcup (\bigcup_{i=1}^{p} St_i), \\ \wedge \begin{cases} \langle W, 0, \_, \_, \_, \langle addr, init(addr)\rangle\rangle \in \tau_0 \\ (\langle W, 0, id, \_, \_, \langle addr, \_\rangle\rangle \in \tau_0 \wedge \langle W, 0, id', \_, \_, \langle addr, \_\rangle\rangle \in \tau_0) \implies id = id' \end{cases} \end{array}$$

Read-from conditions

The communications in rf must satisfy the following conditions:

Definition 24 (satisfiability)**.** *Each read event must have a related read-from relation in rf:*

$$\begin{array}{l} \forall \langle R, tid, uid, \iota, stk, \langle addr, val\rangle\rangle \in \tau_{tid}, \\ \exists w, \wedge \begin{cases} \langle w, \langle R, tid, uid, \iota, stk, \langle addr, val\rangle\rangle\rangle \in rf, \\ type(w) \in \{W, T\} \end{cases} \end{array}$$

Definition 25 (uniqueness)**.** *Each read is satisfied by at most one write in rf:*

$$\forall w, w', r, \quad \langle w, r\rangle \in rf \wedge \langle w', r\rangle \implies w = w'$$

Definition 26 (match)**.** *The address and value read and written in rf must match:*

$$\forall \langle r, w\rangle \in rf, type(w) \neq T \implies addr(r) = addr(w) \wedge val(r) = val(w)$$

Definition 27 (inception)**.** *Every event in rf must exist in the execution:*

$$\forall \langle r, w\rangle \in rf, \exists ij, r \in \tau_i \wedge w \in \tau_j$$

Definition 28 (trustworthiness)**.** *If the write occurs in the trusted library, it is not in another thread's stack nor in its own stack, above the base pointer:*

$$\forall (t, r) \in rf, type(t) = T \implies addr(r) \in stack_i \implies tid(t) = i \wedge addr(r) < bp(t)$$

Here, $bp(t)$ is the last base pointer in the stk field of t.

Language-dependent conditions

Definition 29 (start)**.** *Each thread starts in its starting state, in the trusted library:*

$$\forall i \in [1, n], \underline{\tau_{i}}_0 = \langle \iota_0^i, i, 0, \rho_0^i, nil\rangle \wedge \iota_0^i \in Trusted$$

Definition 30 (next)**.** *Each thread is either stopped or can continue to a next state, by emitting an event. $\forall i \in [1, n], \forall j, \langle \iota, i, j, \rho, stk\rangle \xrightarrow{\overline{\tau_i}^j} \langle \iota', i, j+1, \rho', stk'\rangle$, with $\overline{\tau_i}^j$, ι', ρ' and stk' defined by the code in P and the communications in rf:*

- *Trust instruction ($\iota \in Trusted$):*
  $\overline{\tau_i}^j = \langle T, i, j, \iota, stk\rangle$*. The next state is either:*
  - *$\iota' \in \mathcal{F}$, ρ' is arbitrary, $addr \in Trusted$, $stk' = \langle addr, \rho', bp'\rangle :: stk$, such that bp' is smaller than the last bp in stk (if any), and inside the thread stack frame.*
  - *When $stk = \langle ret, \rho_i, bp\rangle :: stk_2$, and stk_2 is not empty, $stk' = stk_2$, $\rho' \sim \rho_i$ and $\iota' = addr$.*
- *Register instruction ($instr(\iota) = \lfloor r = e\rfloor$):*
  $\overline{\tau_i}^j = \langle L, i, j, \iota, stk, \langle [\![e]\!]_\rho\rangle\rangle$, $\rho' = \rho[r \mapsto [\![e]\!]_\rho]$, $stk' = stk$ *and* $\iota' = \iota^+$
- *Write instruction ($instr(\iota) = \lfloor [e_1] = e_2\rfloor$):*
  $\overline{\tau_{ij}} = \langle W, i, j, \iota, stk, \langle [\![e_1]\!]_\rho, [\![e_2]\!]_\rho\rangle\rangle$ $\rho' = \rho$, $stk' = stk$ *and* $\iota' = \iota^+$

— *Read instruction* ($instr(\iota) = \lfloor r = [e] \rfloor$)*:*
  $\overline{\tau_{ij}} = \langle R, i, j, \iota, stk, \langle [\![e]\!]_\rho, val \rangle \rangle$, $\rho' = \rho[r \mapsto val]$, $stk' = stk$ *and* $\iota' = \iota^+$

— *Branch instruction* ($instr(\iota) = \lfloor call\ e \rfloor$, $instr(\iota) = \lfloor ret\ e \rfloor$ *or* $instr(\iota) = \lfloor jmpif\ c\ e \rfloor$)*:*

  — *A call or a ret:* $\overline{\tau_{ij}} = \langle B, i, j, \iota, stk, \langle call \vee ret, \iota' \rangle \rangle$, $\iota' = [\![e]\!]_\rho$, $\rho' = \rho$. *In case of a call, we have:* $stk' = (\langle \iota^+, \rho, [\![ESP]\!]_\rho \rangle) :: stk$

    In case of a ret, we have: $stk' = nil$ *if* $stk = nil$ *or* $stk' = stk_{rest}$ *if* $stk = stk_{first} :: stk_{rest}$

  — *On the true branch of jmpif:* $\overline{\tau_{ij}} = \langle B, i, j, \iota, stk, \langle jump, \iota' \rangle \rangle$ *with* $[\![c]\!]_\rho \neq 0$ *and* $\iota' = [\![e]\!]_\rho$.

  — *On the false branch of jmpif:* $\overline{\tau_{ij}} = \langle B, i, j, \iota, stk, \langle jump, \iota^+ \rangle \rangle$ *with* $[\![c]\!]_\rho = 0$ *and* $\iota' = \iota^+$.

6.2.5 Memory Model

Finally, we define here the memory model we will use. Intuitively, we want our memory model to allow as many behaviors as possible, so we have an over-approximation of the actual behaviors of the program on real hardware. The first two conditions apply to communications (*rf*). They are a weaker version of the standard *Uniprocessor correctness condition*:

— No communication happen between a write and a read in the same thread if the read is before the write in program order.

— No communication happen between a write and a read in the same thread if there is a write event to the same location between the two events in program order.

— No communication happen between an initial write and a read if there is a write event to the same location between the initial event and the read in program order.

The following two assumptions add a *dependency* relation *dep* between some events. This relation covers both the *address dependencies* and *data dependencies* and is defined as:

— *address dependency*: There are edges from read and register events where registers were last assigned to read, write or branch events where these registers are used to compute the address of the event.

— *data dependency*: There are edges from read and register events where registers were last assigned to write, register or branch events where these registers are used to compute the value of the event.

Additionally, we define the *control* relation *ctrl* between some events. This relation is defined as:

$$ctrl = \left\{ e \xrightarrow{ctrl} w | \exists b, type(b) \in \{B, T\} \wedge r \xrightarrow{dep} b \xrightarrow{po^*} w \right\}$$

The preserved program order, *ppo* is defined as *dep* ∪ *ctrl*.

With these additional relations, we can write our memory model condition:

NO-THIN-AIR: acyclic(*rf* ∪ *ppo*)

Note that in [3], the assumption is acyclic(rfe ∪ *ppo*). Compared to our definition, their version of *ppo* also contains *po-loc*, which is exactly *rf* − *rfe*, so we have the same assumption. In this paper, there is an additional assumption that we do not need to make, SC-PER-LOC, so our model is weaker than their's.

This assumption is sufficiently weak to allow some existing architectures, such as x86, arm or power. The alpha architecture is weaker than our assumptions; however we could extend our analyzer to verify that barriers are put in place to ensure the presence of “artificial” dependencies.

CHAPTER 7

ANALYZING MULTITHREADED MODULES

We now have all the basics for defining an analysis of multithreaded modules. The goal of the analysis is to create an over-approximation of the possible executions. In the previous chapter, we defined a concrete semantics for multithreaded modules under weak memory models.

In the previous part of this thesis, we had been following the methodology defined in Chapter 2.2. However, weak memory models feature out of order execution and two weak memory models do not always feature the same kind of order. This is an issue when we want (later) to go from a sequential (abstract and interleaving) semantics to a weaker (concrete) memory model: a step in the weak memory model cannot always correspond to a step in the stronger memory model as it could leave many "holes" behind.

To solve this issue, we change our methodology. In this chapter, we are going to define a few axiomatic semantics: the anarchic semantics, that is similar to the concrete semantics, but where the data is abstracted away, in a very similar fashion to what we did between the defensive and the dataless semantics in the monothreaded semantics. Then, we will define a sequential semantics, which is stronger than the previous semantics, since its execution must follow po-order. This semantics is then compared to a multithreaded, interleaving variant of the dataless semantics. At each step, we will prove that the security of a program under a more abstract semantics implies the security of that program under the more concrete semantics. Then, the final theorem will summarize our work: if a program is secure under the abstract, monothreaded semantics, then it is secure under the concrete, multithreaded, axiomatic semantics. The table below shows the semantics we use, their specificities and what the link with the other semantics is:

	Concrete semantics [6.2]	The axiomatic REIL semantics
$\star$	Anarchic semantics [7.2]	The same semantics, but the sandbox and initial content of the stack are not modeled
$\star$	Sequential semantics [7.3]	An axiomatic version of the interleaving dataless semantics
$\star$	Small-step Dataless semantics [7.4]	A small-step monothreaded semantics, where the sandbox is abstracted. Compared to the following semantics, it uses small-steps for calls
$\star$	Dataless semantics [4.2]	A monothreaded semantics, where the sandbox is abstracted, and calls are modeled with a transitive closure
α	Intra-procedural semantics [4.3]	The same semantics where function calls are abstracted
α	Abstract semantics [4.4]	The fully abstract semantics where we do our analysis

$\star$: Specific lemma — α: abstract interpretation

Abstract interpretation using our methodology is possible only between the abstract semantics and the dataless semantics. For the remaining semantics, we prove a specific lemma to show that, when the program is secure under an abstract semantics, it is secure under the semantics above it in the previous table.

7.1 Security Property

In the first part of this thesis, we defined the security property as a progress property of a defensive semantics. However, a defensive multithreading semantics would not make sense: first of all, even if a thread is blocked, another might still go on. Then, the semantics is non deterministic, so conditions should take this into account. So, it would be possible to have a defensive semantics, but it would be a lot more complex than the standard property. Therefore, we define the security property as a property on events of an execution. When every legal execution of a program is secure, we say that the program itself is secure. A secure execution verifies the following properties on events:

Definition 31 (Security Property). *A read event $\langle R, i, j, \iota, stk, \langle addr, val\rangle\rangle$ verifies either:*

- *the address of the event, $addr$, is in the sandbox, or*
- *the address of the event, $addr$, is in the thread's stack*

A write event $\langle W, i, j, \iota, stk, \langle addr, val\rangle\rangle$ verifies either:

- *the address of the event, $addr$, is in the sandbox, or*
- *the address of the event, $addr$, is in the current stack frame*

A branch event $\langle B, i, j, \iota, stk, \langle addr, val\rangle\rangle$ verifies either:

- *the event type is a jump and the address ι is in the same context (trusted or untrusted) as the jump address, or*
- *the event type is a call and the address $addr$ of the jump is a callable function, in the trusted or untrusted module.*
- *the event type is a ret and the address $addr$ of the jump is the last return address in stk (so, stk is not empty), and if stk has only one element, that address is in the trusted module.*

Other events do not need to verify any condition.

7.2 Anarchic Semantics

We define a new semantics, called the *anarchic semantics* that is very similar to our first semantics, but we model memory access from the sandbox and initial memory as being non deterministic, i.e. not modeled with *rf*. More precisely, we relax the read-from condition *satisfiability*, which becomes:

Definition 32 (Satisfiability).

$$\forall \langle R, tid, uid, \iota, stk, \langle addr, val\rangle\rangle \in \tau_{tid},$$
$$addr \notin [d_0; d_0 + d_s] \implies \exists w, \wedge \begin{cases} \langle w, \langle R, tid, uid, \iota, stk, \langle addr, val\rangle\rangle\rangle \in rf \\ typ(w) \in \{W, T\} \end{cases}$$

We also add the condition *noinitmem*:

Definition 33 (NoInitMem). $\forall (w, r) \in rf, addr(w) \in Stack(tid) \implies tid(w) \neq 0.$

This condition does not remove any accessible state from the semantics, because an initial read of a value can be replaced by a read from the initial state (which yields a trusted event). The reason why we add this condition is to allow an easier proof when we will want to show the security of this semantics from the security of the sequential semantics we are going to define later.

Now we prove that a program secure under the anarchic semantics is secure under the concrete semantics.

Lemma 16. $\forall P, Secure(\llbracket P \rrbracket_{ana}) \implies Secure(\llbracket P \rrbracket)$

Proof. The new *satisfiability*, which replaces the same-named condition in the concrete semantics, is more relaxed: it allows for more behaviors, while the additional *noinitmem* does not restrict any behavior. If the program is secure under the anarchic semantics, every correct execution of the program is secure under that semantics. Since every correct execution of the program under the concrete semantics is a correct execution of the program under the anarchic semantics, the program is also secure under the concrete semantics. □

Before we go on with a more abstract semantics, we first show a lemma on this semantics. This lemma is going to be very useful to show abstraction properties on the next semantics. Intuitively, the lemma says that there is a total order such that, if two events are ordered by *rf* or *ppo*, they are also in the same order in *to* and if an event is after a branch or a trust event in the program ($b \leq e$), it is also after the branch or trust event in *to*.

Lemma 17.

$$\begin{aligned} &\forall P, \forall \langle \sigma_0 \times \prod_{t=0}^{n} \sigma_t, rf \rangle \in \llbracket P \rrbracket_{ana}, \exists \textit{to}, \textit{a total order on } (\bigcup_{t=0}^{n} \sigma_t), \\ &\quad \forall e \forall e', e \leq_{\textit{rf} \cup \textit{ppo}} e' \implies e \leq_{\textit{to}} e' \wedge \\ &\quad \forall e \forall b, type(b) \in \{B, T\} \implies b \leq_{\textit{po}} e \implies b \leq_{\textit{to}} e \end{aligned}$$

Proof. Let P be a program and ς a valid computation of this program under the anarchic semantics.

We construct a family (to_i) of total orders on subsets (ς_i) of the computation ς, such that $\mid \varsigma_i \mid = i$ and $\forall i, \varsigma_i \subset \varsigma_{i+1}$.

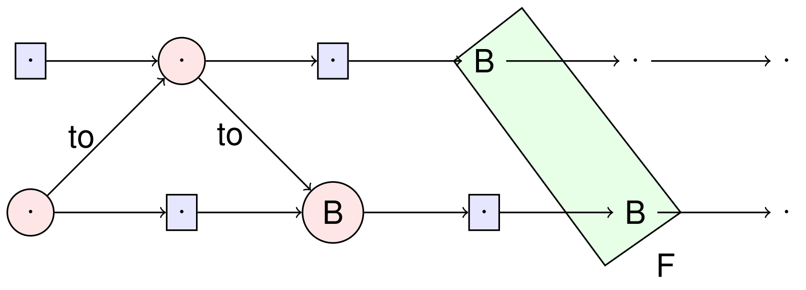
First, we can only take $\varsigma_0 = \emptyset$ and to_0 is a total order on an empty set.

Then, we suppose by induction that we have to_n, a total order on $\varsigma_n \subseteq \varsigma, \mid \varsigma_n \mid = n$.

If $\mid \varsigma \mid = n$, we have our total order: *to* $= to_n$.

Otherwise, by induction hypothesis, we suppose: $\forall e \forall e' \in \varsigma_n, e \leq_{\textit{rf} \cup \textit{ppo}} e' \implies e \leq_{to_n} e'$ and $\forall e \in \varsigma_n, \forall e' \in \varsigma, type(e') = B \implies e' \leq_{po} e \implies e' \in \varsigma_n \land e' \leq_{to_n} e$.

We now prove that we can select an new element of ς that respects the property we want: it is either the first branch or trust event in a thread that is not yet in ς_i, or an event before that first branch or trust event. For this, we create two new sets: The frontier F which consists in branches or trust events that satisfy the previous property and EF, the set of events that satisfy that property. Graphically, we want to prove that one of the events in $\boxed{\text{EF}}$ or $\boxed{\text{F}}$ can be selected as the next event in the order we are constructing:



B: branch or trust event — ·: arbitrary non-branch non-trust event

$\boxed{e}$: EF e : ς_i

We define F:

$$F = \left\{ e \;\middle|\; \begin{array}{l} type(e) = \{B, T\} \land \\ e \in \varsigma \setminus \varsigma_n \land \\ \forall e', e' \leq_{po} e \implies type(e') \in \{B, T\} \implies e' \in \varsigma_n \end{array} \right\}$$

and EF:

$$EF = \left\{ e \;\middle|\; \begin{array}{l} e \in \varsigma \setminus \varsigma_n \land \\ (\exists e' \in F, e \leq_{po} e') \lor (\forall e' \in F, tid(e') \neq tid(e)) \end{array} \right\}$$

We need to prove one of the events in EF can be the additional event in ς_{i+1}. For this, we examine each node in EF and list the nodes it depends on. We will see that, if a node e in EF depends on a node by *rf* or *ppo* that is outside of ς_i, there is another node e' in EF such that is either a good choice for the new element in ς_{i+1} or that is such that $e' \leq_{\textit{rf}\cup\textit{ppo}} e$. Since EF is finite, if all nodes depended on another node of EF by *rf* and *ppo*, we would be able to find a cycle in *rf* ∪ *ppo* which is supposed to be acyclic. Therefore, either we have found a good choice, or at least a node depends only on nodes from ς_i and it is a good choice for the new element in ς_{i+1}.

So, we now prove that:

$$\forall e' \in EF, \forall e \notin \varsigma_i,$$
$$(e, e') \in \textit{rf} \cup \textit{ppo} \implies \exists e'' \in EF, \left| \begin{array}{l} e'' \leq_{\textit{rf}\cup\textit{ppo}} e' \\ \quad\vee \\ \forall e''', e''' \leq_{\textit{rf}\cup\textit{ppo}} e'', e''' \in \varsigma_i \end{array} \right.$$

Let's take such a pair of events and examine them by case analysis on the type of relation.

If the relation is *ppo*, e is po-before e' and is therefore inside EF, so we can take $e'' = e$. If the relation is *rf*, we have different cases to examine. Either e is in the same thread as e', in which case it must be po-before it (according to the memory model), so we can also take $e'' = e$. Either e is in another thread than e'. In that case, it could already be in EF, in which case we can again take $e'' = e$, or it is po-after the branch or trust event in F for that thread. The write event e might be the target of control relations from EF or not. If there is e'' such that $(e'', e) \in$ *ctrl*, we take e''. Otherwise, since there is no control event in EF, the branch or trust event b in F for that thread only has dependencies on events from ς_i or no dependency at all (otherwise, there would be a control relation from an event before it). Since a branch can only be targeted by a dependency relation, and a trust event is not targeted by any relation, that event only depends on events from ς_i by *rf* and ppo, so it is a good choice for the next event in ς_{i+1} and we take $e'' = b$.

We construct $\varsigma_{n+1} = \varsigma_n \cup \{e''\}$ and we add e'' as the biggest element of to_{n+1}. This new order respects *rf* ∪ *ppo*.

Additionally, we have $\forall e' \in \varsigma, type(e') = B \implies e' \leq_{po} e'' \implies e' \leq_{to_{n+1}} e''$, because ς_n contains every branch or trust event that is *po*-before any event in EF. □

7.3 Sequential Semantics

We now define another semantics, called the *sequential semantics* (sequential in the sense that it is sequentially consistent, even though it's still multithreaded) that is similar to the previous one, but the communication assumptions, previously NO-THIN-AIR is now that $acyclic(\mathit{rf} \cup \mathit{po} \cup \mathit{mo} \cup \mathit{fr})$, where *mo* is a *modification order* defined as a total order on write events to the same address ($\mathit{mo} = \bigcup_{i \in \mathbb{B}_*} \mathit{mo}_i$) and *fr* (from reads) is defined as the relation between two events where the first is targeted by a *rf* relation from a write, and the second is targetted by a *mo* relation from that write, unless the two events are the same: $\mathit{fr} = \mathit{rf}^{-1}; \mathit{mo} \setminus \mathit{id}$. Note that we do not use *dep* nor *ctrl* since they are included in *po*. The security property is still the same as before.

7.3.1 Intuition

The previous proof of security was easy, because the abstraction had a superset of the behaviors of the concrete semantics. In the case of the sequential semantics, some behaviors of its concrete semantics (the anarchic semantics) are not possible. Let's take this short program for instance:

[x] = [y] = 0	
(a) r1 = [y] (b) [x] = 1	(c) r2 = [x] (d) [y] = 1
Can r1 = r2 = 1?	

In the anarchic semantics, the result $r1 = r2 = 1$ is possible. However, in the sequential semantics, the NO-THIN-AIR condition forbids $rf \cup po$ to be circular, which would be the case if (a) read from (d) and (b) from (c). We see here that we do not have a superset of the behaviors of the anarchic semantics.

We can still provide a proof of security though: instead of proving the behaviors under the sequential semantics are a superset of the behaviors of the program under the anarchic semantics, we will use the fact that it is the case when the program is secure under the sequential semantics.

This program can only be secure under the sequential semantics when x and y are both inside the sandbox. Otherwise, say if x is outside of the sandbox, the execution where the first thread executes completely before the second thread is not secure in the sequential semantics: either the write to x is insecure, or it is a write to the current stack frame, but the subsequent read of x is insecure. The same applies to y when we examine the execution that starts from the second thread.

In the case where x and y are both in the sandbox, both reads from the sandbox do not need to come from a write and can read any value, including 1. The behavior that seemed to be impossible at first is allowed by the new *satisfiability* rule we introduced in the anarchic semantics when the execution is secure.

7.3.2 Proofs

As before, we will prove that, when the program is secure under the sequential semantics, it is secure under the anarchic semantics.

Theorem 7. $\forall P, secure(\llbracket P \rrbracket_{seq}) \implies secure(\llbracket P \rrbracket_{ana})$.

Proof. First, we notice that $secure(\llbracket P \rrbracket_{seq}) \implies$ there is no inter-thread communications.

We can prove that if an execution of the program under the anarchic semantics has an inter-thread communication, there is an execution of the program under the sequential semantics that also has an inter-thread communication (see Lemma 22). By contraposition, we can deduce that there is also no inter-thread communication in any anarchic execution of the program.

When an execution in the anarchic semantics does not have any inter-thread communication, it is also a valid execution under the sequential semantics (see Lemma 21). Therefore, since anarchic executions of the program have no inter-thread communication, they are all executions of the program under the sequential semantics. By hypothesis, they are all secure.

Finally, since the security property is the same on both semantics, we can conclude with $secure(\llbracket P \rrbracket_{ana})$. □

The proof of this theorem was rather small. In fact, it relies on a few intermediate lemmas we will now present. The first intermediate lemma takes a state s' from the same thread as an event e that wrote a value v to a register r. When s' is executed after e (after the initial write of v), but before any other event that writes to the same register r (before v is overwritten), the environment of that state associates v to r.

Lemma 18.

$$\forall P, \forall \langle \sigma_0 \times \prod_{t=0}^{n} \sigma_t, rf\rangle \in [\![P]\!]_{seq}, \forall t, \forall (\xrightarrow{e} s) \in \sigma_t, \forall (\xrightarrow{e'} s') \in \sigma_t,$$
$$\wedge \left(\begin{array}{l} instr(\iota(e)) = \lfloor r = expr \rfloor \\ e \leq_{po} e' \\ \forall (\xrightarrow{e''} s'') \in \sigma_t, \wedge \left(\begin{array}{l} e \leq_{po} e'' <_{po} e' \\ (instr(\iota(e'')) = \lfloor r' = expr' \rfloor \implies r \neq r') \\ (instr(\iota(e'')) = \lfloor r' = [expr'] \rfloor \implies r \neq r') \end{array} \right) \end{array} \right) \implies [\![r]\!]_{\rho(s')} = val(e)$$

Proof. Suppose first that we have $\langle \iota, tid, uid, \rho, stk\rangle \xrightarrow{\langle L, tid, uid', \iota, stk, \langle val \rangle\rangle} \langle \iota', tid', uid', \rho', stk'\rangle \in \sigma_{tid}$. In that case, by the language-dependent well-formedness conditions, we have $\rho' = \rho[r \leftarrow val]$, so $[\![r]\!]_{\rho'} = val$.

Then, suppose that we have $\xrightarrow{\langle L, tid, uid, \iota, stk, \langle val \rangle\rangle} s \xrightarrow{e} ... \xrightarrow{e'} \langle \iota', tid, uid', \rho', stk'\rangle \xrightarrow{e''} \langle \iota'', tid, uid'', \rho'', stk''\rangle \in \sigma_{tid}$, $[\![r]\!]_{\rho'} = val$, and
$instr(\iota') = \lfloor r' = expr \rfloor \implies r \neq r' \wedge instr(\iota') = \lfloor r' = [expr] \rfloor \implies r \neq r'$.

By case analysis on $instr(\iota')$, and using the language-dependent well-formedness conditions, we can show that $[\![r]\!]_{\rho'} = [\![r]\!]_{\rho''}$, unless the instruction modifies the register r, which we have supposed is not the case. □

The previous lemma can be used to prove the following lemma. When there is a dependency relation from e to e', it means that e is a local computation or a memory read that sets register r, and e' uses that register. The following lemma says that the register contains the value that was written to it by e in the state where e' is executed.

Lemma 19.

$$\forall P, \forall \varsigma \in [\![P]\!]_{seq}, (\xrightarrow{\langle L, tid, uid, \iota, stk, \langle val \rangle\rangle} s) \xrightarrow{\textsf{dep}} (\xrightarrow{e'} s'),$$
$$(\xrightarrow{\langle L, tid, uid, \iota, stk, \langle val \rangle\rangle} s) \xrightarrow{\textsf{po}^*} (\xrightarrow{e} \langle \iota', tid', uid', \rho', stk'\rangle) \xrightarrow{\textsf{po}} (\xrightarrow{e'} s') \implies$$
$$instr(\iota) = \lfloor r = expr \rfloor \implies [\![r]\!]_{\rho'} = val$$

Proof. This is a corollary of the previous lemma: if two events are related by a dependency, there cannot be an event that modifies r in between, because of the definition of the dependency relation. Conditions for the lemma are verified to $[\![r]\!]_{\rho'} = val$. □

The following lemma is the same, but when e is a read from memory instead of a local computation.

Lemma 20. $\forall P, \forall \varsigma \in [\![P]\!]_{seq}$,
$(\xrightarrow{\langle R,tid,uid,\iota,stk,\langle addr,val\rangle\rangle} s) \xrightarrow{dep} (\xrightarrow{e'} s') \implies$
$(\xrightarrow{\langle R,tid,uid,\iota,stk,\langle addr,val\rangle\rangle} s) \xrightarrow{po} (\xrightarrow{e} \langle \iota', tid', uid', \rho', stk' \rangle) \xrightarrow{po} (\xrightarrow{e'} s') \implies$
$instr(\iota) = \lfloor r = [expr] \rfloor \implies [\![r]\!]_{\rho'} = val$

Proof. The proof is similar to the proof of Lemma 19. □

With these intermediate lemmas, we can now show that a program that is secure under the sequential semantics is also secure under the anarchic semantics. To do that, we have to show the following two main lemmas.

First, this lemma says that if an anarchic execution has no inter-thread communications, it is also a valid sequential execution:

Lemma 21. $\forall (\varsigma, rf) \in [\![P]\!]_{ana}, (\forall (e, e') \in rf, tid(e) = tid(e')) \implies \varsigma \in [\![P]\!]_{seq}$.

Proof. Take such an execution in the anarchic semantics. Since there is no inter-thread communication, we have $\forall (e_1, e_2) \in rf, tid(e_1) = tid(e_2)$.

To exhibit a sequential execution, we need to find a modification order mo and prove that $po \cup rf \cup mo \cup fr$ is acyclic. Remember that $mo = \bigcup mo_l$, where mo_l is a total order on writes to memory location l. Here, we take mo_l to be the lexicographic order of write events to memory location l in the trace, ordered by thread id first, then by their unique id in the thread. In this way, we can say that $\forall (e_1, e_2), e_1 \leq_{mo} e_2 \implies tid(e_1) \leq tid(e_2)$, and mo is compatible with po. Now, mo, po and rf are all three a subset of the lexicographic order on the thread id and unique id.

Then, by definition we have $fr = rf^{-1}; mo \setminus id$. For any two events related by fr, there is a third event such that: $e_2 \leq_{fr} e_3 \implies \exists e_1, e_1 \leq_{rf} e_2 \land e_1 \leq_{mo} e_3$. In order to prove that this relation is also a subset of the lexicographic, we have three cases to consider, because rf and mo both are compatible with the lexicographic ordering of events, and because rf only happens inside the same thread by hypothesis:

e_1 and e_3 are in the same thread, and so is e_2 because rf is intra-thread

$$\begin{tikzcd}
e_1 \arrow[d, "mo"] \arrow[dd, bend right, "rf"'] \\
e_3 \\
e_2 \arrow[u, "fr"]
\end{tikzcd}$$

If e_2 is po-after e_3, but in that case, rf is not well formed, since e_3 is a write to the location read by e_2. The rf relation canot cross it.

$$\begin{tikzcd}
e_1 \arrow[d, "rf"'] \arrow[dd, bend left, "mo"] \\
e_2 \arrow[d, "fr"] \\
e_3
\end{tikzcd}$$

But if e_2 is po-before e_3, then fr respects the lexicographic order of thread id and unique id.

e_1 and e_3 are in separate threads.

$$\begin{tikzcd}
e_1 \arrow[r, "mo"] \arrow[d, "rf"'] & e_3 \\
e_2 \arrow[ur, "fr"] &
\end{tikzcd}$$

Since mo respects the lexicographic order, we have $tid(e_2) < tid(e_3)$, so fr respects the lexicographic order.

In every case, fr respects the lexicographic order.

Since all four relations are a subset of the lexicographic ordering of events, the union is also a subset of that order. Since the lexicographic order is acyclic, so is $po \cup rf \cup mo \cup fr$.

Since the well-formedness conditions are respected (because it is a valid anarchic execution) and the memory model is respected (because of the acyclicity condition), that execution is also an execution of the program in the sequential semantics. □

The second main lemma we use in the proof of the theorem is the following. It states that, when there is an inter-thread communication in the anarchic semantics, there is an execution of the program in the sequential semantics that also has a communication.

Lemma 22. $\forall(\varsigma, rf) \in [\![P]\!]_{ana}, \forall(e_1, e_2) \in rf,$
$tid(e_1) \neq tid(e_2) \implies \exists(\varsigma', rf') \in [\![P]\!]_{seq}, \exists(e'_1, e'_2) \in rf', tid(e'_1) \neq tid(e'_2).$

Proof. In order to prove that lemma, we are actually going to exhibit a specific execution of the program in the sequential semantics, that has the same structure as a subset of the anarchic execution. The proof relies on the specific total order *to* on events of the anarchic execution of which we proved the existence in Lemma 17.

We call r_0 the first read in the total order that reads from another thread. We call the thread of r_0 the first thread, we call w the write event from which it reads and we call the thread of w_0 the second thread. We consider the following total order *to'*: first execute the entirety of the first thread in program order, up to but not including r_0, then the second thread up to and including w_0, then r_0. *to'* is defined on a subset of events from the execution, and we call A the set of such events that are *after* r_0 in *to*.

We will show that there is a sequential execution that follows order *to'*, with the same events for those not in A and similar, but not necessarily equal events for those in A. We first have to define what it means to be "similar".

Definition 34 (ς-OK)**.** *We say that a pair of event and state in the sequential semantics is ς-OK if it has a corresponding pair in ς (an anarchic execution) that agrees on some of their content, that is:*

$$\begin{cases} \varsigma\text{-}OK(\xrightarrow{\langle R,tid,uid,\iota,stk,\langle addr,val\rangle\rangle} \langle addr,tid,uid,\rho,stk\rangle) \equiv \\ \quad \exists(\xrightarrow{\langle R,tid,uid,\iota,stk,\langle\_,\_\rangle\rangle} \langle addr,tid,uid,\_,\_\rangle) \in \varsigma \\ \varsigma\text{-}OK(\xrightarrow{\langle W,tid,uid,\iota,stk,\langle addr,val\rangle\rangle} \langle addr,tid,uid,\rho,stk\rangle) \equiv \\ \quad \exists(\xrightarrow{\langle W,tid,uid,\iota,stk,\langle\_,\_\rangle\rangle} \langle addr,tid,uid,\_,\_\rangle) \in \varsigma \\ \varsigma\text{-}OK(\xrightarrow{\langle B,tid,uid,\iota,stk,\langle typ,addr\rangle\rangle} \langle addr,tid,uid,\rho,stk\rangle) \equiv \\ \quad \exists(\xrightarrow{\langle B,tid,uid,\iota,stk,\langle\_,\_\rangle\rangle} \langle addr,tid,uid,\_,\_\rangle) \in \varsigma \\ \varsigma\text{-}OK(\xrightarrow{\langle L,tid,uid,\iota,stk,\langle val\rangle\rangle} \langle addr,tid,uid,\rho,stk\rangle) \equiv \\ \quad \exists(\xrightarrow{\langle L,tid,uid,\iota,stk,\langle\_\rangle\rangle} \langle addr,tid,uid,\_,\_\rangle) \in \varsigma \\ \varsigma\text{-}OK(\xrightarrow{\langle T,tid,uid,\iota,stk\rangle} \langle addr,tid,uid,\rho,stk\rangle \equiv \\ \quad \exists(\xrightarrow{\langle T,tid,uid,\iota,stk\rangle} \langle addr,tid,uid,\_,\_\rangle) \in \varsigma \end{cases}$$

First, we notice that the first event in *to'* is ϵ_{start} and that is the same event as in the sequential semantics. Suppose now we have a sequential execution Σ that agrees on the first i events in *to'*, i.e. for any of the first i events, either it is in A, and there is an event in the sequential execution that is ς-OK, or is not in A and the event is also present in the sequential execution. We consider the $(i+1)^{th}$ event in *to'*, e. First, we prove that ς-OK(e).

If e is the start event of the second thread, it is the same in both semantics, so in particular, it is ς-OK. Otherwise, the previous event is either ς-OK, a branch or a trust event. In these two last cases, because of the way *to* is constructed, the event is necessarily to-before r_0 or w_0 depending on its thread, so it is also present in the sequential execution we consider. If it was a branch, then the computed address and value is necessarily the same in both semantics. If it was a trust event, one of the possible next event has the same instruction pointer as the event in the anarchic semantics.

In all these cases, because of the language-dependent conditions, the next event in the sequential semantics has the same thread id, unique id, instruction pointer, stack (if the previous event was a ret or a call that changes the stack, remember that the event is present in full) and type as e. This is enough to prove that all events are ς-OK, except for branches: we still need to prove they jump to the same address that the branch in the anarchic semantics.

Because branches only have dependencies, and these dependencies are in to, they are also present in full in the sequential execution we consider. Therefore, the values that are used to compute the branch target and condition are the same in both semantics, which results in the same following instruction pointer.

If e is in A, this is enough, but if it is not in A, we need to prove that the event in the sequential execution is the same as e. We prove this by case analysis on e.

— If it is a read, by the same reasoning as with the branch event previously, all its dependencies have the same value, so the computed address of the read is the same in both semantics. If the address is in the sandbox, the value being read is arbitrary, so at least a sequential execution has an event with the same value. If the address is outside of the sandbox, the read must come from the local thread, unless it is r_0. If it is from the local thread, it is also the same for the sequential semantics: this is the last write to that address in the order we consider. For r_0, the last write is that of w_0, which is not in A, so it is present in the sequential execution. In both cases, the value being read is the same, since the write event is present in both semantics.

— If it is a write, because of the same reasoning, all its dependencies have the same value, so the computed address and value are the same in both semantics.

— If it is a local computation, for the same reasons, the value being computed is the same.

— If it is a branch, the value and address are both the same in both semantics.

— If it is a trusted event, there is nothing more than ς-OK to prove the equality between e and the event in the sequential semantics.

In the end, we have proved that we have a (partial) sequential execution whose event are all ς-OK, and correspond to events either in A or to the same event in ς. In particular, r_0 and w_0 are both present in the Σ and they form an inter-thread communication.

□

7.4 Bridging the gap with the Single-threaded Semantics

The goal of this last section is to show that the static analysis we defined before is still valid and sound for multithreaded semantics, even in the context of a weak memory model. To do this, we are going to show that the dataless semantics (see section 4.2) is an abstraction of the sequential semantics. To do that, we first define a small-step variant of the dataless semantics (remember that the function call was defined as a closure on the execution of the function). This small-step semantics is the last semantics in the path between the abstract and the concrete semantics.

We will prove the equivalence between the small-step semantics and the dataless semantics, and the equivalence of the security property between the two variants. Finally, using an abstraction, we will prove that a program that is secure under the small-step semantics is also secure under the sequential semantics.

7.4.1 Small Step Dataless Semantics

First, we define a small-step variant of the dataless semantics. It works similarly to the dataless semantics, but instead of using a context and states, it regroups both as parts of its states, with no execution context. A state in this new semantics is composed of a stack (represented as a list) whose elements are a dataless inter-procedural context and an intra-procedural dataless state, and an intra-procedural state. The stacks will be useful for the proof, and for the security property of the return instruction.

$$State = Context \times (Context^{\downarrow} \times State^{\downarrow})^{*} \times State^{\downarrow}$$

In the dataless semantics, we used small steps except on the call rules. We define a fully small-step variant of that semantics that has the same rules, except that call rules are split into two distinct rules: a call and a ret.

Except for the two call rules, we have:

$$\Gamma \vdash s \rightarrow^{\downarrow} s' \implies (\Gamma, \Pi, s) \triangleright (\Gamma, \Pi, s')$$

The small-step call rule is similar to the `CallAcc` rule, but we push the new context and previous state to the stack:

$$\textsc{Call}\;\frac{\begin{array}{c} instr(\iota) = \lfloor \mathbf{call}\ e \rfloor \quad [\![e]\!]_\rho = f \quad f \in \mathcal{F} \cup \mathcal{T} \\ \rho(\mathbf{esp}) = bp - o \quad \mid \phi_1 \mid = o \quad o < f_s \quad isret(\iota^+, \rho, \phi_1) \quad s = \langle \rho, \delta, \phi_1 \cdot \phi_2, \iota \rangle \end{array}}{\langle \langle cs, bp, \rho_i \rangle, \Pi, s \rangle \triangleright \langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho \rangle, (\langle cs, bp, \rho_i \rangle, s) :: \Pi, \langle \rho, \delta, \phi_2, f \rangle \rangle}$$

The small-step return rule pops one item from the stack. Security is ensured by conditions $isret$ and $\sim$, which ensure that the ret corresponds to its call. These conditions are the conditions related to the return instruction in the call rule of the dataless semantics.

$$\textsc{Ret}\;\frac{\rho_i \sim \rho' \quad isret(ret, \phi_1, \rho_i) \quad instr(\iota) = \lfloor \mathbf{ret}\ e' \rfloor \quad [\![e']\!]_{\rho'} = ret}{\langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, (\Gamma, \sigma) :: \Pi, \langle \rho', \delta, \phi_2', \iota \rangle \rangle \triangleright \langle \Gamma, \Pi, \langle \rho', \delta, \phi_1 \cdot \phi_2', ret \rangle \rangle}$$

Finally, some of these calls can be calls to the trusted library, and some returns can be returns to the trusted library. In both cases, the following state has its address in the trusted library, for which there is no instruction, since we do not model the content of the trusted library. We are in the same situation as with the multithreaded semantics: we have states in the trusted library, but we abstract it away by only specifying the limits on what it is allowed to do.

$$\textsc{TRet}\;\frac{\iota \in T \quad \rho_i \sim \rho' \quad isret(ret, \phi_1, \rho_i) \quad \mid \phi_2 \mid = \mid \phi_2' \mid}{\langle \langle cs :: \langle bp, \phi_1 \rangle, bp - o, \rho_i \rangle, (\Gamma, \sigma) :: \Pi, \langle \rho', \delta, \phi_2, \iota \rangle \rangle \triangleright \langle \Gamma, \Pi, \langle \rho', \delta, \phi_1 \cdot \phi_2', ret \rangle \rangle}$$

$$\textsc{TCall}\;\frac{\begin{array}{c} \iota \in T \quad addr \in \mathcal{F} \quad isret(\iota^+, \phi_1, \rho) \\ \sigma = \langle \rho, \delta, \phi_1 \cdot \phi_2, \iota \rangle \quad \mid \phi_1 \mid = o \quad o \leq f_s \end{array}}{\langle \langle cs, bp, \rho_i \rangle, \Pi, \sigma \rangle \triangleright \langle \langle cs :: \langle s_0, \phi_1 \rangle, bp - o, \rho \rangle, (\langle cs, bp, \rho_i \rangle, \sigma) :: \Pi, \langle \rho, \delta, \phi_2, addr \rangle \rangle}$$

The security property on this semantics is the same as the security property of the dataless semantics: a program is secure if any reachable state has a next state.

An initial state of this small-steps semantics is in the trusted library, with an arbitrary stack content, an empty state stack and a context stack with only one element: an empty call stack, a base pointer set to the top of the stack and an arbitrary environment, that is the same as the environment in the state:

$$Init = \{\langle nil, s_0, \rho\rangle, nil, \langle \rho, \phi, \iota\rangle || \ \phi \models s_s\}$$

7.4.2 Small-Step Security

The strategy to prove the security of the program under the small-step semantics is to first define an abstraction from small-step states to dataless states. Then, we show that the set of abstracted states is included in the set of reachable dataless states, and finish by showing that the security of the abstracted state implies the security of the small-step state.

The abstraction is defined as:

$$\alpha(\Gamma, \Pi, \sigma) = \langle \Gamma, \sigma\rangle$$

We also define an abstraction for an element of the stack in this way:

$$\alpha(\Gamma, s) = \langle \Gamma, s\rangle$$

We will use the same symbol and name for both abstractions. As their types are different, they cannot be confused.

Now, we prove that if the abstraction is secure (i.e. has a next state), so is the state we abstracted:

Lemma 23. $Secure(\alpha(\Gamma, \Pi, \sigma)) \implies Secure(\Gamma, \Pi, \sigma)$

Proof. If the abstraction is secure, it has a following state: $\Gamma, \sigma \Rightarrow \Gamma', \sigma'$.

There are three cases: either this is because the `RetAcc` rule was taken, in which case the new `Ret` rule can apply as its conditions are the same.

If the `CallAcc` rule was taken, similarly, the new `call` rule can be taken by the small-step state.

If another rule was taken, it may be `CallTrust` rule, in which case all the conditions for the new `call` rule apply, the `Call` rule, in which case all the conditions fer the new `call` rule also apply, or any other rule, in which case the conditions are the same in both semantics, so they also apply.

In every case, a rule of the small-steps semantics apply, so the small-step state has a next state and is therefore secure. □

Now, we prove that for any reachable small-step state, its abstraction is either in the trusted library or is reachable in the dataless semantics, and we prove the same for the abstraction of every element of its stack.

The following predicates state exactly that. If the stack is empty, then a small-steps state is reachable through the dataless semantics if its abstraction is reachable from the abstraction of an initial state.

$$\frac{(\Gamma, \emptyset, \sigma) \in \mathit{Init} \quad \Gamma \vdash \sigma \to^* \sigma'}{\mathit{BReachable}(\Gamma, \emptyset, \sigma)}$$

If the stack is not empty, then a small-steps state is reachable through the dataless semantics if its abstraction is reachable in the dataless semantics from the state that came from the saved state in the stack, and if the state in the stack is itself reachable through the dataless semantics.

$$\frac{\begin{array}{c}\mathit{BReachable}(\langle cs, bp, \rho_i\rangle, \Pi, \langle \rho, \phi_1 \cdot \phi_2, \iota\rangle) \quad \Gamma' = \langle cs :: (bp, \phi_1), bp - o, \rho\rangle \\ f \in \mathcal{F} \quad \mathit{isret}(\iota^+, \phi_1, \rho) \\ \mid \phi_1 \mid = o \quad o \leq f_s \quad \Gamma' \vdash f \to^* s'\end{array}}{\mathit{BReachable}(\Gamma', (\langle cs, bp, \rho_i\rangle, \langle \rho, \phi_1 \cdot \phi_2, \iota\rangle) :: \Pi, s')}$$

And we also want to accept programs that go back and forth to the trusted library, so a state in the trusted library is also said to be reachable through the dataless semantics, because it is not modeled there:

$$\frac{\iota \in T \quad \mathit{BReachable}(\Gamma, \Pi, s)}{\mathit{BReachable}(\Gamma', (\Gamma, s) :: \Pi, \langle \rho, \phi, \iota\rangle)}$$

Then we prove that if a state is reachable in the small-steps semantics, it is also reachable through the dataless semantics:

Lemma 24. $\forall (\Gamma, \Pi, \sigma) \in \mathit{Reachable}, \mathit{BReachable}(\Gamma, \Pi, \sigma)$.

Proof. We reason by induction of the reachability. First of all, the initial state is reachable through the dataless semantics, because it is in the trusted library.

In the inductive case, we have a small step from a state s that is reachable through the dataless semantics, to a state s'. We proceed by a case analysis on the type of transition between s and s'.

If it is neither a ret nor a call, then the stack and context do not change and both s and s' are in the untrusted module. Therefore, we know that $\Gamma \vdash f \rightarrow^* s$ and $\Gamma \vdash s \rightarrow^* s'$. The actual rule depends on the size of the stack, but in both cases (null and non null), every requirement is met, as well as $\Gamma \vdash f \rightarrow^* s'$, so s' is reachable through the dataless semantics.

If it is a call, the following state s' is either in the trusted library, in which case it is reachable through the dataless semantics (because the recursive case is that s is reachable through the dataless semantics). Otherwise, the call verifies every condition for being reachable through the dataless semantics: we have $\Gamma' \vdash f \rightarrow^* f$ (it suffices to make no step) and the recursive case is that s is reachable through the dataless semantics. In every case, s' is reachable through the dataless semantics.

Finally, if it is a ret, s has one element in the stack than s', and from that, we can already derive the recursive case (if any). We also have $\Gamma \vdash \sigma \rightarrow^* \sigma'$ where σ' is the state that calls f, the first state of the function we return from. We also deduce that $\Gamma' \vdash f \rightarrow^* s$, which is enough, with the rest of the reachability conditions, to have the conditions for the big-step call from the dataless semantics. So we have $\Gamma \vdash \sigma \rightarrow^* \sigma' \rightarrow s'$, which allows us to conclude that s' is reachable through the dataless semantics. □

Finally, if a state is reachable through the dataless semantics, its abstraction is also reachable in the dataless semantics, or it is in the trusted library.

Lemma 25. $\forall s, BReachable(s) \rightarrow Reachable^{\downarrow}(\alpha(s)) \vee s \in T.$

Proof. We can prove this by noting that there are three cases, one per rule. In the two first rules, we have $\Gamma \vdash f \rightarrow^* s$ where f is an initial state in the dataless semantics, so s is reachable. In the last case, s is in the trusted library. □

Lemma 26. *If a program is secure under the dataless semantics, it is secure under the small-step variant of the same semantics.*

Proof. Every reachable dataless state is secure. By the previous lemma, we know that any abstraction of a reachable small-steps state is a reachable dataless state, so any abstraction of a reachable small-steps state is secure. By the lemma before that, we can conclude that, since any abstraction is secure, any reachable small-steps state is also secure. □

7.4.3 Sequential Security

We now have to prove that the sequential semantics (see section 7.3) is linked to the small-steps semantics we defined above, in such a way that if the program is secure under the small-steps semantics, it is also secure under the sequential semantics. To prove this, we again use an abstraction, α, that takes a sequential trace and returns a set of small-steps states. The abstraction is defined recursively. The abstraction content is mostly derived from the last sequential state and the abstraction of the trace that ends just before the last event. In particular, the abstracted state always has the same environment and instruction pointer as the last sequential state.

The abstraction of the initial pair of events and states is:

$$\alpha(\xrightarrow{\epsilon_{start}} \langle \iota, i, j, \rho, nil \rangle) = \left\{ \left(\begin{array}{c} \langle nil, s_0^i, \rho \rangle \\ nil \\ \langle \iota, \rho, \phi \rangle \end{array} \right) \middle| \; \phi \models s_s \right\}$$

Where s_0^i is the address at the top of the stack for thread i. We can note that this is a set of initial events for the small-steps semantics.

In general, if we have an abstraction of some trace, we can construct an abstraction for a trace that has one more pair of events and states. If the last event of the trace is a read event, we have:

$$\alpha(\sigma \xrightarrow{\langle R, i, j, \iota, stk, \langle addr, val \rangle \rangle} \langle \iota', i, j', \rho', stk \rangle) = \left\{ \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \langle \iota', \rho', \phi \rangle \end{array} \right) \middle| \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \langle \iota, \rho, \phi \rangle \end{array} \right) \in \alpha(\sigma) \right\}$$

Intuitively, reading a value in memory does not change the memory, so we keep ϕ as it was before the read. We did not change the context either, so it is kept the same as before. The same goes for local computations:

$$\alpha(\sigma \xrightarrow{\langle L, i, j, \iota, stk, \langle val \rangle \rangle} \langle \iota', i, j', \rho', stk \rangle) = \left\{ \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \langle \iota', \rho', \phi \rangle \end{array} \right) \middle| \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \langle \iota, \rho, \phi \rangle \end{array} \right) \in \alpha(\sigma) \right\}$$

Write events are more interesting: a write might be inside the current frame, in which case ϕ must be updated with the new value, or it is outside of the frame, in which case nothing is updated, because the memory outside of the frame is not represented. Note that it is not completely "correct" for insecure modules, but we will show later that it is correct for modules that are secure under the dataless semantics.

$$\alpha(\sigma \xrightarrow{\langle W,i,j,\iota,stk,\langle addr,val\rangle\rangle} \langle \iota', i, j', \rho', stk\rangle) =$$

$$\left\{ \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \left\langle \iota', \rho', \left| \begin{array}{rl} \phi[addr \leftarrow val] & \text{if } 0 \leq bp - addr, \\ & bp - addr \leq \mid \phi \mid - \mathsf{GZ}_\perp - \mid val \mid \\ & \text{and } addr \leq s_0 - \mathsf{GZ}_\top \\ \phi & \text{otherwise} \end{array} \right. \right\rangle \end{array} \right) \middle| \begin{pmatrix} \Gamma, \\ \Pi, \\ \langle \iota, \rho, \phi \rangle \end{pmatrix} \in \alpha(\sigma) \right\}$$

Branch events are also more interesting: there are three types of branches. The first is a conditional jump, and nothing is surprising here:

$$\alpha(\sigma \xrightarrow{\langle B,i,j,\iota,stk,\langle jump,addr\rangle\rangle} \langle \iota', i, j', \rho', stk\rangle) = \left\{ \begin{pmatrix} \Gamma, \\ \Pi, \\ \langle \iota', \rho', \phi \rangle \end{pmatrix} \middle| \begin{pmatrix} \Gamma, \\ \Pi, \\ \langle \iota, \rho, \phi \rangle \end{pmatrix} \in \alpha(\sigma) \right\}$$

The second type of branch is a call, and we need to update our memory representation here. First, we need to split the stack in two at the new base pointer position, just as we do with the dataless semantics. Then, we push a new inter-procedural context, that corresponds to the context used for computing the closure of the function call in the dataless semantics:

$$\alpha(\sigma \xrightarrow{\langle B,i,j,\iota,stk,\langle call,addr\rangle\rangle} \langle addr, i, j', \rho, stk\rangle) =$$

$$\left\{ \begin{pmatrix} \langle cs :: (bp', \phi_1), \rho, bp \rangle, \\ (\langle cs, \rho', bp' \rangle, \langle \iota, \rho, \phi_1 \cdot \phi_2 \rangle) :: \Pi, \\ \langle addr, \rho, \phi_2 \rangle \end{pmatrix} \middle| \begin{pmatrix} \langle cs, \rho', bp' \rangle, \\ \Pi, \\ \langle \iota, \rho, \phi_1 \cdot \phi_2 \rangle \end{pmatrix} \in \alpha(\sigma) \right\}$$

The last type of branch is a function return, and we need to update our memory representation here too. First, we need to merge the last stack from back into ϕ in the state. Then, we pop the last inter-procedural context from the list of contexts.

$$\alpha(\sigma \xrightarrow{\langle B,i,j,\iota,\langle ret,\rho_i,bp\rangle::stk,\langle ret,addr\rangle\rangle} \langle addr,i,j',\rho,stk\rangle) = \left\{ \left(\begin{array}{c} \Gamma, \\ \Pi, \\ \langle addr,\rho,\phi_1\cdot\phi_2\rangle \end{array} \right) \middle| \left(\begin{array}{c} \langle cs :: (bp,\phi_1),\rho',bp'\rangle, \\ (\Gamma,s) :: \Pi, \\ \langle \iota,\rho,\phi_2\rangle \end{array} \right) \in \alpha(\sigma) \right\}$$

Finally, the last type of event is the trust event, that corresponds to running any code in the trusted library. This event has two cases that correspond to one of two ways to exit the trusted library and enter the module again: a call to a function, or a return to after the last function call to the trusted library.

If the transition corresponds to a function call, we have the same kind of abstraction as with the call event, but the trusted library can have modified its own stack frame (ϕ_1 and ϕ_2):

$$\alpha(\sigma \xrightarrow{\langle T,i,j,\iota,stk\rangle} \langle addr,i,j',\rho,stk\rangle) = \left\{ \left(\begin{array}{c} \langle cs :: (bp',\phi_1'),\rho,bp\rangle, \\ (\langle cs,\rho',bp'\rangle,\langle \iota,\rho,\phi_1\cdot\phi_2\rangle) :: \Pi, \\ \langle addr,\rho,\phi_2'\rangle \end{array} \right) \middle| \begin{array}{l} \left(\begin{array}{c} \langle cs,\rho',bp'\rangle, \\ \Pi, \\ \langle \iota,\rho,\phi_1\cdot\phi_2\rangle \end{array} \right) \in \alpha(\sigma) \\ \wedge \quad \mid \phi_1 \mid = \mid \phi_1' \mid \\ \quad\;\; \mid \phi_2 \mid = \mid \phi_2' \mid \end{array} \right\}$$

If the transition corresponds to a function return, we have the same kind of abstraction as with the ret event, but the trusted library can have modified its own stack frame (ϕ_2 only):

$$\alpha(\sigma \xrightarrow{\langle T,i,j,\iota,\langle ret,\rho_i,bp\rangle::stk\rangle} \langle addr,i,j',\rho,stk\rangle) = \left\{ \left(\begin{array}{c} \Gamma, \\ \Pi \\ \langle ret,\rho'',\phi_1\cdot\phi_2'\rangle \end{array} \right) \middle| \begin{array}{l} \left(\begin{array}{c} (ret,\rho',bp'), \\ (\Gamma,s) :: \Pi, \\ \langle \iota,\rho,\phi_2\rangle \end{array} \right) \in \alpha(\sigma) \\ \wedge \quad \mid \phi_2 \mid = \mid \phi_2' \mid \\ \quad\;\; \rho' \sim \rho'' \end{array} \right\}$$

We can note that, for insecure executions, it might not always be possible to find an abstraction. However, the following lemma states that, when the module is secure under the dataless semantics, we can always find an abstraction which is in the set of reachable dataless states, and that will help us show at the same time that sequential executions are then secure too.

Lemma 27. *If the small-steps semantics is secure, the abstraction of any valid sequential execution is included in the reachable set of the small-steps semantics and is not empty, and the sequential execution is actually secure:*

$$Secure(\llbracket P \rrbracket) \implies \forall s \in \alpha(\sigma), \sigma \in \llbracket P \rrbracket_{seq} \implies Reachable(s) \wedge Secure(\sigma)$$

Proof. We prove this by induction on the sequential execution.

First of all, the abstraction of the initial sequential execution is a list of initial small-steps states, which are reachable by definition. The initial sequential execution is only composed of initial start events, so it is also secure.

Then, if we take a sequential execution on which there is a total order, we consider the sequential execution composed of the same event and states except for the last one in that total order. By induction hypothesis, we suppose this execution has at least one abstraction that is reachable in the small-steps semantics. Every small-steps state in the abstraction is also part of the abstraction of the corresponding thread, except for the thread that has one more event. Since by hypothesis these are reachable, have at least one abstraction and are secure, we only need to focus on the thread with the new event.

This thread can be written as $\sigma \xrightarrow{e} s$, and we know that $\alpha(\sigma)$ is not empty and contains only reachable small-steps states. By case analysis on e, we can deduce some information about $\alpha(\sigma)$ and the transition from any state in it to some s' (which always exists because we suppose the abstraction is secure).

For branches and local computations, the language-specific conditions impose that there is only one possible next step, and that is also the case for the small-steps semantics. When the preconditions are respected in both semantics, the next state is computed in the same way, so the set of next states of abstractions of σ is the abstraction of $\sigma \xrightarrow{e'} s'$, which is therefore not empty and reachable.

For a read event, we note that the preconditions impose that it happens either in the stack or in the sandbox. If it is in the sandbox, then the next state of any abstraction is the same state, with one register set to an arbitrary value, which is the same as what happens to s', so the abstraction of $\sigma \xrightarrow{e'} s'$ is not empty and contains the next states of the abstraction of σ. They are therefore reachable. If the read is from the stack, the sequential execution contains an rf relation to that read. It cannot come from an event in another thread, because they are secure: they cannot write to another thread's stack. So the read comes from the last trust event or write event to that address, whichever is the latest. If it comes from a write, the abstraction contains the value that was written in its ϕ or previous states, which is the value read by the sequential semantics. If it comes from a trust event, at least one abstraction contains the value read in e'. In both cases, every abstraction of $\sigma \xrightarrow{e'} s'$ is one step after an abstraction of σ. The abstraction of $\sigma \xrightarrow{e'} s'$ is not empty and contains only reachable small-steps states.

For a write, the conditions impose that the address is either in the sandbox or the current stack frame. In both cases, there can only be one next state for a small-steps state, and every abstraction is updated in the same way. This way is compatible with the small-steps semantics, so the abstraction of $\sigma \xrightarrow{e'} s'$ is composed of the next states of the states in the abstraction of σ.

For a trust event, the new abstraction contains any state where the current frame contains an arbitrary value, which is also the case for the trust transition in the small-steps semantics. For a return as well as for a call-style trust event, the abstraction of $\sigma \xrightarrow{e'} s'$ is composed of the next states of the abstraction of σ.

In every cases, because of the preconditions, the new event e' is secure. □

From this lemma, it is easy to derive this corollary:

Lemma 28. $Secure(\llbracket P \rrbracket) \implies Secure(\llbracket P \rrbracket_{seq})$

Proof. Since the small-steps semantics is secure, any execution in the sequential semantics has an abstraction and is secure, so the program is secure under the sequential semantics. □

7.4.4 Final Theorem

With all this work, we are now able to show that the same analyzer we used to show the security of a single-threaded module can be used to show the security of a multi-threaded module under a weak memory model.

Theorem 8. $Secure(\llbracket P \rrbracket^\sharp) \implies Secure(\llbracket P \rrbracket)$

Proof. By following the methodology and results of Chapter 2.2, and by using again the results of Chapter 4, we can see that the abstract semantics is an abstraction of the dataless semantics. Therefore, abstract interpretation is able to determine the security of the program under the dataless semantics.

Then, by following the results of this chapter, we can successively use the security of the dataless semantics to prove the security of the program under the small-steps semantics, then under the sequential semantics, the anarchic semantics and finally, under the concrete semantics. $\square$

In the end, the analyzer we designed for our language with a simple execution model is generic enough to handle the same language with a much more complex execution model, with a weak memory model and multiple threads. Although this reflects the fact that the analyzer is very imprecise, it also means that no infrastructure change needs to happen between the two execution models. This lack of precision is mostly caused by the fact that we completely abstract the sandbox away, but that is also the key reason why the analysis works on a multithreaded model.

Instead of designing a complex analysis that would have to take into account every thread, we have a much simpler analysis that works on every thread at once, function by function, independently of whether or not they are going to be run. Going even further, in Chapter 5, we have used a tool based on a C compiler to produce modules that can be analyzed. This tool was not meant to produce modules that would be run in parallel. With this result, and since our analyzer should, in theory, validate any binary produced by that compiler, we can conclude that the tool is indeed able to produce modules that can be run in a parallel program.

CONCLUSION

Summary

In this thesis we have proposed a verification process to ensure cooperation of untrusted modules with the rest of a host program. It is based on *Software Fault Isolation*, a technique to isolate untrusted modules in a sandbox (i.e. a memory region outside of which the module is not allowed to read or write). SFI provides an answer to the question: how to isolate a program from its guest, untrusted, modules, even when they are untrusted and there are no hardware isolation mechanism available. Its answer however is limited to a syntactic property. Although it allows for a fast and simple linear verifier, its properties are somewhat arbitrary. In fact, this work started by a semantic analysis of an arbitrary module, and what kind of behavior would be acceptable, and what kind would not be, in order to design a more generic property for SFI.

Formalization

We have developed a *defensive semantics* that formalizes the verifications done by SFI. In most cases, we have followed the spirit of SFI: we implemented a sandbox in this semantics and we developed restrictions on what code could be called by the module. The defensive semantics defines a relaxed isolation property allowing for a more flexible analysis than standard SFI: contrary to SFI, our technique allows for sharing the stack between the trusted host and untrusted module. We propose a new approach to verifying that calling conventions are respected and that stack overflows and underflows cannot happen at run-time.

Intuitively, the security property states that reads and writes are only possible in the sandbox and in the stack. The dynamic nature of our security property meant that we had to implement a static analyzer that would do more than a simple syntax check. Most papers on SFI do not talk about the implementation of their verifier extensively, if at all. Contrary to them, this thesis is focused on the implementation of a verifier and a proof of its correctness, under different architectural models.

We used a simple theory based on abstract interpretation to prove the soundness of our analysis. By abstracting a concrete semantics step by step until reaching the target abstract semantics, we modularize and simplify the proof effort, where each intermediate semantics exhibits a single kind of abstraction. Every abstraction is not sound under the abstract interpretation framework, but the result is still correct, because they are still sound for programs that are accepted by the verifier. More precisely, for each pair of abstract and concrete semantics, we have proved that when the program is secure under the more abstract semantics, the abstraction is sound, and the program is secure under the more concrete semantics.

An abstract domain inspired by value-set analysis is used to verify properties on calling conventions as well as on sandboxing. An actual implementation of a verifier was then derived from the abstract semantics we developed.

Implementations

Our first implementation is based on a simple monothreaded semantics that is very close to a standard semantics. Its main differences with a more standard semantics are its defensive nature (most rules come with additional checks) and a call rule implemented as a transitive closure of the function call, when the rest of the semantics is implemented as a small-steps semantics. This ensures the semantics is already quasi intra-procedural. We showed how to abstract this initial semantics step by step, by abstracting away the memory sandbox, then abstracting the stack as a small window around the base pointer, and finally abstracting the values manipulated by the semantics. A preliminary version of this work was published in *Static Analysis Symposium* [7].

Our second implementation is based on a more complex multithreaded semantics. This time, it is an axiomatic semantics that works under a weak-memory model. We presented a model that is weak enough to have at least the same behaviors as some commonly used architectures (`arm`, `x86`, ...). Using a similar approach, we abstracted the semantics further and further, each step exhibiting a specific abstraction: first we abstracted the data sandbox away, then we abstracted the weak behaviors under a sequential semantics and finally, we abstracted the semantics to an interleaving, small-steps semantics and demonstrated how this interleaving semantics was abstracted by a previously defined semantics. By reusing previous results, we could demonstrate that the analyzer we implemented was able to work even when considering multithreaded programs under a weak-memory model. This shows how robust the analyzer, and SFI in general, is to changing the memory model.

Future Work and Perspectives

A first extension is extending to multiple untrusted, separate modules. Our current proofs were done under the assumption that there was a trusted library and a single untrusted module. What if there were more than one untrusted module? Although no work was done on that direction yet, it seems that from the point of view of an untrusted module, there is no difference between the trusted library and other untrusted module: we should not allow direct access to the memory of other modules, or direct call of functions in other untrusted modules. This tends to indicate that our current semantics are appropriate for modeling the behavior of a module in a process where other modules are running.

Another interesting extension would be an implementation of a verifier in a proof assistant such as Coq, with a complete proof of correctness. First, because our implementation uses a slightly different language from what our proofs are using, and second, because it would further reduce the Trusted Computing Base to that of Coq or another proof assistant.

Some engineering work could also be done on the implementation to improve it using some standard techniques, such as refining on guards. Currently, we do not refine on guards and that is detrimental for detecting some sound optimizations. We could also implement a double analysis, where the first pass implements a very imprecise widening, but is quicker, and if it failed, a second pass that implements a more precise analysis. Finally, more work needs to be done on detecting and validating some forms of indirect jumps: switch cases in C are typically implemented with a table of jump addresses for instance. With a more precise analysis in general, we would be able to validate more kinds of optimizations.

Finally, we could implement a verifier for more standard SFI implementations. Remember that in NaCl for instance, the module's stack is separate from the trusted library's stack. The module code is also divided into constant-length bundles and the generator ensures that control flow always points to the beginning of a bundle, either by nop-padding them, or by sandboxing jumps. If, for instance, the lifter transformed any jump, call and ret from an NaCl module to simple conditional jumps (ignoring the nature of calls and returns), and bundle boundaries were marked as potential function starts, our analyzer could probably work and validate these modules.

Control flow would be harder to analyze than in more standard binaries: almost any bundle can jump to any other bundle. However, if we decide there is no function call nor return involved at the intermediate language level, and the stack is not touched (an NaCl module has its own private stack in the sandbox, separate from the actual process stack), the analyzer will simply analyze every bundle and be very imprecise at the beginning of each bundle (their state would be the union of the states at the end of almost every bundle).

There might be some difficulties in validating calls to the trusted library: they still need to be marked as calls and correctly set their return address, but they are not allowed to write to the process stack. The architecture-dependent $isret$ may need to be changed for that purpose.

In the long term, it could be interesting to analyze the content of the sandbox more precisely: it could allow for a better analysis and for a better understanding of the module's stack under more standard implementations. However, we have seen that abstracting the sandbox away was central to the proofs of correctness in the multithreading case. It would be challenging to find what kind of analysis would be precise enough on the sandbox content to be useful, while still allowing for a multithreaded execution.

BIBLIOGRAPHY

Sources primaires

[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] Jade Alglave and Patrick Cousot, « Ogre and Pythia: An Invariance Proof Method for Weak Consistency Models », *in*: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, Paris, France: ACM, 2017, pp. 3–18, ISBN: 978-1-4503-4660-3, DOI: `10.1145/3009837.3009883`, URL: `http://doi.acm.org/10.1145/3009837.3009883`.

[3] Jade Alglave, Luc Maranget, and Michael Tautschnig, « Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory », *in*: *ACM SIGPLAN Notices* 49 (Aug. 2013), DOI: `10.1145/2627752`.

[4] 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`.

[5] Gogul Balakrishnan and Thomas W. Reps, « Analyzing Memory Accesses in x86 Executables », *in*: *Compiler Construction*, vol. 2985, LNCS, Springer, 2004, pp. 5–23.

[6] Frédéric Besson, Sandrine Blazy, and Pierre Wilke, « A Concrete Memory Model for CompCert », English, *in*: *Interactive Theorem Proving*, ed. by Christian Urban and Xingyuan Zhang, vol. 9236, Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 67–83, ISBN: 978-3-319-22101-4, DOI: `10.1007/978-3-319-22102-1_5`, URL: `http://dx.doi.org/10.1007/978-3-319-22102-1_5`.

[8] 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.

[9] Philippe Biondi et al., « BinCAT: purrfecting binary static analysis », *in*: *Symp. sur la sécurité des technologies de l'information et des communications*, 2017.

[10] David Brumley et al., « BAP: A Binary Analysis Platform », *in*: *Computer Aided Verification*, vol. 6806, LNCS, Springer, 2011, pp. 463–469.

[11] Erik Buchanan et al., « When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC », *in*: *Proceedings of CCS 2008*, ed. by Paul Syverson and Somesh Jha, ACM Press, Oct. 2008, pp. 27–38.

[12] 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.

[13] TIS Committee, *Executable and Linking Format specification 1.2*, May 1995, URL: http://refspecs.linuxbase.org/elf/elf.pdf.

[14] P. Cousot et al., « Varieties of Static Analyzers: A Comparison with ASTRÉE, invited paper », *in*: *Proc. First IEEE & IFIP International Symp. on Theoretical Aspects of Software Engineering, TASE'07*, ed. by He Jifeng and J. Sanders, Shanghai, China: IEEE Computer Society Press, June 2007, pp. 3–17.

[15] Patrick Cousot and Radhia Cousot, « Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints », *in*: *Proc. of the 4th Symp. on Principles of Programming Languages*, ACM, 1977, pp. 238–252, DOI: 10.1145/512950.512973, URL: http://doi.acm.org/10.1145/512950.512973.

[16] Christopher Domas, *Breaking the x86 ISA*, tech. rep., July 2017, URL: https://github.com/xoreaxeaxeax/sandsifter/raw/master/references/domas_breaking_the_x86_isa_wp.pdf.

[17] Christopher Domas, *Sandsifter*, URL: https://github.com/xoreaxeaxeax/sandsifter.

[18] Thomas Dullien and Sebastian Porst, « REIL: A platform-independent intermediate representation of disassembled code for static code analysis », *in*: *CanSecWest'09*, 2009.

[19] 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`.

[20] 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`.

[21] Andreas Haas et al., « Bringing the Web Up to Speed with WebAssembly », *in*: *Proc. of the 38th Conf. on Programming Language Design and Implementation*, ACM, 2017, pp. 185–200, ISBN: 978-1-4503-4988-8, DOI: `10.1145/3062341.3062363`, URL: `http://doi.acm.org/10.1145/3062341.3062363`.

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

[23] Jacques-Henri Jourdan et al., « A Formally-Verified C Static Analyzer », *in*: *Proc. of the 42Nd Symp. on Principles of Programming Languages*, ACM, 2015, pp. 247–259, ISBN: 978-1-4503-3300-9, DOI: `10.1145/2676726.2676966`, URL: `http://doi.acm.org/10.1145/2676726.2676966`.

[24] Johannes Kinder, « Static Analysis of x86 Executables », PhD thesis, Technische Universität Darmstadt, Nov. 2010, URL: `http://tubiblio.ulb.tu-darmstadt.de/47134/`.

[25] 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`.

[26] 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.

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

[28] Bogdan Mihaila, « Adaptable Static Analysis of Executables for proving the Absence of Vulnerabilities », PhD thesis, Technische Universität München, 2015.

[29] Antoine Miné, « Abstract Domains for Bit-Level Machine Integer and Floating-point Operations », *in*: *Proc. of the Workshops on Automated Theory eXploration and on Invariant Generation*, vol. 17, EPiC Series in Computing, EasyChair, 2013, pp. 55–70, URL: http://www.easychair.org/publications/?page=1928840229.

[30] 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.

[31] Scott Owens, Susmit Sarkar, and Peter Sewell, « A Better x86 Memory Model: x86-TSO », *in*: Aug. 2009, pp. 391–407, DOI: 10.1007/978-3-642-03359-9_27.

[32] *REIL Specification*, Zynamics, URL: https://www.zynamics.com/binnavi/manual/html/reil_language.htm.

[33] 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, ISBN: 888-7-6666-5555-4, URL: http://dl.acm.org/citation.cfm?id=1929820.1929822.

[34] Jaroslav Ševčík et al., « CompCertTSO: A Verified Compiler for Relaxed-Memory Concurrency », *in*: *Journal of the ACM (JACM)* 60 (June 2013), DOI: 10.1145/2487241.2487248.

[35] Hovav Shacham, « The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) », *in*: *Proceedings of CCS 2007*, ed. by Sabrina De Capitani di Vimercati and Paul Syverson, ACM Press, Oct. 2007, pp. 552–61.

[36] Yan Shoshitaishvili et al., « SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis », *in*: *IEEE Symp. on Security and Privacy*, 2016.

[37] 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.

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

[39] 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.

[40] The Coq Development Team, *Coq*, version 8.7, Oct. 15, 2017, URL: https://coq.inria.fr.

[41] 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.

[42] 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.

[43] 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.

[44] S. Zhang, M. Vijayaraghavan, and Arvind, « Weak Memory Models: Balancing Definitional Simplicity and Implementation Flexibility », *in*: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2017, pp. 288–302, DOI: 10.1109/PACT.2017.29.

[45] 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.

Travaux publiés durant la thèse

[7] 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.

Titre : Vérification d'isolation de faute logicielle

Mot clés : Vérification, SFI, Sémantique, Interprétation abstraite, Modèles mémoire faibles

Résumé : Nous sommes habitués à utiliser des ordinateurs sur lesquels coopèrent des programmes d'origines diverses. Chacun de ces programmes a besoin d'accéder à de la mémoire vive pour fonctionner correctement, mais il ne faudrait pas qu'un programme accède ou modifie la mémoire d'un autre programme. Si cela ce produisait, les programmes ne pourraient plus faire confiance à la mémoire et pourraient se comporter de manière erratique. Les programmeurs n'ont pourtant pas besoin de se mettre d'accord à l'avance sur les zones mémoire qu'ils pourront ou non utiliser. Le matériel s'occupe d'allouer des zones de mémoire distinctes pour chaque programme. Tout cela est transparent pour le programmeur. Un programme malveillant ne pourrait d'ailleurs pas non plus accéder ou modifier la mémoire d'un autre programme pour l'attaquer directement. Mais il existe une catégorie de programmes qui ne bénéficient pas de cette protection : les modules qui étendent les fonctionnalités d'autres programmes, comme un module complémentaire de navigateur. Cette thèse repose sur une technique d'isolation de faute logicielle, et non matérielle et en propose deux sémantiques, l'une parallèle et pas l'autre, ainsi qu'un analyseur statique basé sur l'interprétation abstraite. Elle présente aussi une preuve de correction de l'analyseur.

Title: Verifying Software Fault Isolation

Keywords: Verification, SFI, Semantics, Abstract Interpretation, Weak Memory Models

Abstract: We are used to use computers on which programs from diverse origins are installed and running at the same time. Each of these programs need to access memory for proper operation, but none of them should access or modify the memory of another. If this happened, programs would not be able to trust their memory and could start behaving erratically. Still, programmers do not need to coordinate and agree in advance on what parts of the memory they are allowed to use or not. Hardware takes care of allocating distinct memory zones for each program. This is completely transparent to the programmer. A malware cannot access or modify the memory of another program to attack it directly either. However, there exists a category of programs that do not benefit from this protection: modules that extend the features of other programs, such as plugins in a web browser. This thesis is based on a software (and not hardware) fault isolation technique, and proposes two semantics for it, single-threaded and multi-threaded, as well as a static analyzer based on abstract interpretation. We also present a proof of correctness for the analyzer.