Cap' ou pas cap' ?

Preuve de programmes pour une machine à capacités en présence de code inconnu

Aïna Linn Georges[1], Armaël Guéneau[1], Thomas Van Strydonck[2], Amin Timany[1], Alix Trieu[1], Sander Huyghebaert[3], Dominique Devriese[3], and Lars Birkedal[1]

[1] Aarhus University, Danemark [2] KU Leuven, Belgique [3] Vrije Universiteit Brussel, Belgique

Résumé

Une machine à capacités est un type de microprocesseur permettant une séparation des permissions précise grâce à l'utilisation de *capacités*, mots machine porteurs d'une certaine autorité. Dans cet article, nous présentons une méthode permettant de vérifier la correction fonctionnelle de programmes exécutés par la machine alors même que ceux-ci appellent ou sont appelés par du code inconnu (et potentiellement malveillant). Le bon fonctionnement de tels programmes repose sur leur utilisation judicieuse des capacités. Du point de vue logique, notre approche permet donc de tirer parti des garanties fournies par la machine pour raisonner formellement sur des programmes. Les éléments clefs de cette approche sont la définition d'une logique de programmes puis d'une relation logique dont on démontre qu'elle fournit une spécification pour du code inconnu, le tout étant formalisé en Coq.

La méthodologie en question sous-tend le travail précédent des auteurs lié à la formalisation d'une convention d'appel sûre en présence d'un nouveau type de capacités [GGVS+21], mais n'est pas détaillée dans l'article en question. L'article présent se veut être une introduction pédagogique à cette méthodologie, dans un cadre plus simple (sans nouvelles capacités exotiques), et sur un exemple minimal.

1 Introduction

Une machine à capacités ("*capability machine*") est un certain type de microprocesseur fournissant, en sus des fonctionnalités usuelles, des mécanismes de compartimentalisation mémoire et de séparation des privilèges, à grain fin, sous la formes de capacités matérielles. Ce type d'architecture matérielle est étudié depuis les années 1960 [DVH66, Lev84], et en particulier plus récemment au sein du projet CHERI [WNW+19]. La machine à capacités considérée dans cet article se veut être un modèle simplifié d'une machine de la famille CHERI [1].

Une capacité est une valeur associée à une certaine autorité, permettant par exemple d'accéder à une zone de la mémoire ou d'interagir avec un autre composant du système. Dans une machine à capacités, une capacité est représentable par un mot machine, pouvant être stocké dans les registres ou la mémoire, et dont la machine garantit l'intégrité (il est impossible de contrefaire des capacités). Indépendamment de la représentation en machine, on modèle ici une capacité comme étant un 4-uplet (p, b, e, a), avec p une permission et b, e et a des adresses mémoire, $[b, e[$ correspondant à l'intervalle d'autorité de la capacité, et a étant dans l'intervalle.

On peut distinguer différents types de capacités ; on s'intéresse ici aux deux types les plus communs à CHERI. Les *capacités mémoire* donnent l'autorité d'accéder à la plage de mémoire $[b, e[$, avec la permission p (par exemple RW ou RX). Celles-ci sont utilisables comme un pointeur

1. Une différence notable concerne notre utilisation de capacités "enter" au lieu de paires de capacités "scellées".

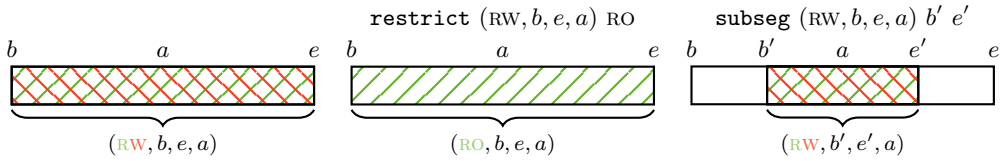

FIGURE 1 – Illustration du comportement de `restrict` et `subseg`.

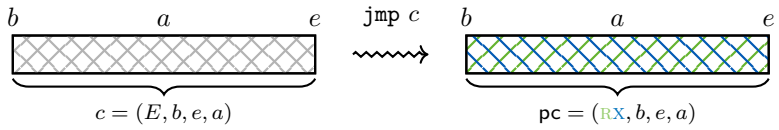

FIGURE 2 – Illustration du comportement de `jmp`.

dont l'inclusion entre les bornes et la permission sont directement vérifiés par le matériel. Par ailleurs, étant donné une capacité mémoire, il est possible d'en dériver une nouvelle capacité avec une autorité plus restreinte, en restreignant la permission (avec l'instruction `restrict`) ou en restreignant la plage d'autorité (avec l'instruction `subseg`) comme illustré en Figure 1.

Les *capacités objet* fournissent un mécanisme similaire aux clôtures des langages de haut niveau. Une capacité objet (associée à la permission "enter" ou E) détient l'autorité permettant d'invoquer un certain composant, sans toutefois donner accès aux capacités privées dont celui-ci a besoin pour fonctionner. Invoquer la capacité (via l'instruction `jmp`) exécute le composant en question et change la permission de la capacité de E à RX (Figure 2), donnant par là accès au code et capacités dans sa plage d'autorité, qui étaient auparavant inaccessibles. Dans CHERI, les capacités objet prennent la forme de paires de capacités code et données, qui sont ensuite "scellées" ensemble [WNW+16] ; la formulation que l'on considère ici provient du M-Machine [CKD94] et est légèrement plus simple mais similaire conceptuellement.

Les capacités permettent donc d'interagir de manière sûre avec du code auquel on ne fait pas confiance, en restreignant les capacités auxquelles celui-ci a accès. Étant donné un système dont les composants ne se font pas tous mutuellement confiance, certains pouvant inclure du code inconnu ou malicieux, les capacités fournissent un moyen de garantir malgré tout que le système obéit à certaines propriétés de sécurité – en initialisant avec soin les composants auxquels on fait confiance, la manière dont ils interagissent avec ceux auxquels on ne fait pas confiance, et en tirant parti des vérifications (dues au capacités) effectuées par la machine à tout instant.

La question est alors : quelles propriétés formelles peut on effectivement faire respecter grâce à l'utilisation des capacités ? Et comment peut on démontrer rigoureusement qu'elles le sont, autrement dit, comment tirer parti des propriétés de la machine à capacités pour raisonner formellement sur l'interaction d'un programme connu avec du code inconnu ?

La réponse proposée ici est la suivante. Tout programme étant appelé par – ou appelant – du code inconnu peut protéger l'accès à certaines capacités et régions mémoires, en utilisant notamment des capacités objet. On dit alors que ces données protégées constituent son "état privé", sur lequel il peut librement établir et maintenir certaines propriétés, à condition d'avoir correctement restreint l'accès du code inconnu aux données privées. Pour toute propriété de l'état privé qui nous intéresse, il suffit ensuite de vérifier : 1) que cette propriété est un *invariant* du code connu (elle est vraie initialement et préservée par son exécution) et 2) que le code connu satisfait dans son ensemble une spécification de "bonne encapsulation". Alors, par propriété de la machine à capacités, on obtient que cet invariant est préservé lors de l'exécution du système

entier, quel que soit le code inconnu interagissant avec le programme connu qui a été vérifié.

Plus précisément, les éléments clefs de cette méthodologie sont les suivants :

— Nous définissons une logique de programme permettant de formellement vérifier la correction de programmes s'exécutant sur notre machine à capacités. Celle-ci est définie à l'aide d'Iris [JKJ+18], une logique de séparation nous fournissant de puissants principes de raisonnement dont notamment la notion d'invariant logique (Section 3).

— Nous définissons, à l'aide de la logique de programme, la spécification de ce que sont une capacité et un programme "sans risque" : une capacité (ou un programme, respectivement) est "sans risque" si elle ne peut pas être utilisée pour invalider un invariant établi précédemment dans la logique. Une capacité sans risque peut donc être partagée librement avec du code inconnu. Cette définition peut être vue comme une relation logique unaire caractérisant notre notion de "sûreté des capacités" (Section 4).

— Nous démontrons (et c'est notre théorème principal) que pour un programme arbitraire, si celui ci n'a accès qu'à des valeurs "sans risque", alors l'exécution du programme lui même est "sans risque". Ceci est une propriété globale de la machine, exprimant que celle-ci "fonctionne bien" : il n'est pas possible pour un programme d'outrepasser l'autorité reçue initialement, quelque soit la séquence d'instructions qu'il exécute (Section 4).

— La dernière pièce du puzzle est le théorème reliant les invariants établis dans la logique de programme à la sémantique opérationnelle de la machine (Section 3). Étant donné un scénario concret (typiquement, un système mélangeant du code connu et vérifié avec du code inconnu et arbitraire), ceci nous permet d'obtenir *in fine* un théorème élémentaire décrivant son exécution en terme de la sémantique opérationnelle de la machine.

L'objectif de cet article est enfin d'illustrer cette méthodologie en la déployant sur un exemple simple. On introduit l'exemple en Section 2, et on détaille sa preuve en Section 5, après avoir introduit les principes de raisonnement nécessaires. Les résultats et exemples présentés ici ont été intégralement formalisés en Coq, et sont disponibles en ligne : `https://github.com/logsem/cerise`.

2 Motivation

On considère dans cet article un scénario simple, dans lequel on vérifie la correction d'un composant connu interagissant avec un composant "adversaire" composé de code inconnu et auquel on ne fait pas confiance.

Commençons par raisonner dans le cadre d'un langage de haut niveau avec références et fonctions de première classe (on utilise ici une syntaxe OCaml, mais le même exemple s'appliquerait aussi en Javascript entre autres).

$$\text{let } x = \text{ref } 0 \text{ in } (\lambda n.\ \text{if } n \geq 0 \text{ then } x := !x + n)$$

Que dire du programme ci-dessus ? Celui-ci alloue une nouvelle référence x initialisée à 0, et crée une clôture permettant d'augmenter la valeur de la référence en y ajoutant un entier n passé en argument, à condition que celui-ci soit positif. Cette clôture est renvoyée au contexte environnant le programme. On s'attend alors que, quelque soit l'utilisation qui est faite de cette clôture par le contexte environnant (même mal intentionné), la valeur de x sera toujours positive. Et en effet, c'est une propriété qu'il est possible de formuler et prouver formellement [SGD17].

Essentiellement, cette propriété est vraie car une implémentation raisonnable d'un langage de haut niveau (par exemple, OCaml ou Javascript) ne permet pas d'inspecter l'environnement d'une clôture. Donc, avoir accès à la clôture produite par le programme ci-dessus ne donne pas

d'accès (direct) à x. Il est seulement possible d'appeler la clôture en lui donnant un argument : celle-ci "encapsule" donc correctement l'état local du programme (la référence x).

La même propriété ("x contient un entier positif à tout instant") est-elle vraie si, après avoir construit la clôture, notre programme passe la main à un contexte adversaire implémenté, non pas en OCaml, mais en assembleur ? La réponse est non : celui-ci peut simplement forger un pointeur vers x, et y écrire un entier négatif, invalidant ainsi la propriété.

Sur une machine à capacités, on peut toutefois préserver cette propriété ! À condition qu'un programme tel que ci-dessus fasse une utilisation judicieuse des capacités objet (une fois compilé ou traduit pour la machine à capacités), alors celui-ci peut passer la main à un contexte arbitraire, écrit en assembleur, sans que celui-ci puisse modifier sa référence privée.

Dans la suite de cet article, on détaille comment la propriété "x est toujours positif" peut être vérifiée formellement, pour une version du programme ci-dessus implémentée en langage machine, et interagissant avec un composant inconnu, également en langage machine. L'implémentation précise de notre programme vérifié apparaît en Figure 4, et sa vérification sera détaillée en Section 5. Dans les sections qui suivent, on définit d'abord les deux principes de raisonnement clefs nécessaires à la vérification : une logique de programme pour notre machine, permettant de vérifier la correction de code connu, et une relation logique et son théorème fondamental, qui donnent une "spécification universelle" pour du code inconnu.

3 Logique de programme

On définit une logique de programme permettant de raisonner de façon modulaire sur les programmes exécutés par la machine. Il s'agit en particulier d'une logique de séparation, définie comme une instanciation d'Iris [JKJ+18] avec la sémantique opérationnelle de notre machine à capacités. Pour des raisons de concision, la définition formelle de la sémantique opérationnelle n'est pas détaillée ici – elle est verbeuse et peu surprenante.

Un aspect important de la sémantique opérationnelle concerne la gestion des erreurs : lorsqu'une vérification liée aux capacités échoue (par exemple lorsqu'un programme essaye d'utiliser une capacité hors de sa plage d'autorité), la sémantique n'est pas "bloquée" ; à la place, la machine évolue explicitement vers un état "échec". Dans la logique de programme, on établira des spécifications "modulo échec", qui autorisent le programme à échouer en cours de route. En effet, d'un point de vue sécurité, faire échouer la machine est en fait sûr ! La propriété que l'on cherche à garantir est que des invariants du code connu sont préservés lors de l'exécution de code inconnu. Dans cette situation, que la machine échoue n'est à la fois pas un problème (les invariants sont bien préservés si la machine est dans l'état d'échec), et également une possibilité qu'on ne peut exclure (on ne peut empêcher le code inconnu d'échouer un test de capacités).

La logique de programme inclut les connecteurs standard de logique de séparation, dont la conjonction séparante ($*$) et la baguette magique ($-\!\!*$, à lire comme une implication). L'assertion $\mathsf{a} \mapsto w$ exprime la possession d'une case mémoire d'adresse a contenant le mot machine w, et l'assertion $r \Mapsto w$ exprime la possession d'un registre r contenant w. Un mot machine w est soit un entier, soit une capacité. On note également $\vec{a} \mapsto \vec{l}$ la possession de plusieurs cellules mémoires d'adresses $\vec{a}$ et contenant $\vec{l}$. Le compteur de programme (registre contenant la capacité pointant vers le code actuellement exécuté) est noté pc.

Un élément clef, hérité d'Iris, est la notion d'invariant logique. L'assertion $\boxed{P}$ (duplicable et persistante) exprime que l'assertion P est satisfaite, et continuera de l'être à chaque étape future de l'exécution. Les règles de preuve associées sont standards et héritées d'Iris.

On distingue trois formes différentes de spécifications de programmes :

$\{w; P\} \rightsquigarrow \bullet$	exécution complète
$\{w_0; P\} \rightsquigarrow \{w_1; Q\}$	fragment de code
$\langle w_0; P\rangle \to \langle w_1; Q\rangle$	unique instruction

Dans chaque cas, w, w_0 ou w_1 dénote la valeur du compteur de programme (pc), et P ou Q est une assertion décrivant l'état de la machine. Typiquement, le compteur de programme contient une capacité avec permission RX, pointant vers une zone de mémoire contenant des entiers correspondant au code du programme, et qui sont décodés en des instructions lors de l'exécution (à noter, tenter de décoder une capacité échoue toujours).

On a $\{w; P\} \rightsquigarrow \bullet$ si, partant d'un état machine satisfaisant P et avec pc égal à w, alors la machine peut s'exécuter jusqu'à s'arrêter (possiblement dans l'état "échec"), ou boucler sans terminer. On a $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$ si, partant d'un état satisfaisant P et avec pc égal w_0, alors on peut arriver à un état satisfaisant Q avec pc égal à w_1. On a $\langle w_0; P\rangle \to \langle w_1; Q\rangle$ lorsque ceci résulte de l'exécution d'une unique instruction. (On a alors typiquement $w_1 = w_0 + 1$, sauf dans le cas des instructions `jmp` et `jnz`). Ces trois spécifications requièrent de plus (ceci est implicite au fonctionnement d'Iris mais crucial) *que les invariants de la logique soient préservés à chaque étape de l'exécution.*

Prouver une spécification de la forme $\{w_0; P\} \rightsquigarrow \{w_1; Q\}$ revient à utiliser en séquence une série de règles de la forme $\langle w_0; R\rangle \to \langle w_1; S\rangle$, une pour chaque instruction du bloc de code considéré. Plus généralement, ces trois notions de spécification de programmes se composent des façons auxquelles on peut s'attendre ; par exemple, on a les propriétés suivantes :

SeqFrag
$$\frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}}$$

SeqFull
$$\frac{\{w_0; P\} \rightsquigarrow \{w_1; Q\} \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet}$$

StepFull
$$\frac{\langle w_0; P\rangle \to \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \bullet}{\{w_0; P\} \rightsquigarrow \bullet}$$

StepFrag
$$\frac{\langle w_0; P\rangle \to \langle w_1; Q\rangle \qquad \{w_1; Q\} \rightsquigarrow \{w_2; R\}}{\{w_0; P\} \rightsquigarrow \{w_2; R\}}$$

La dernière pièce du puzzle est le théorème d'adéquation reliant une spécification établie dans la logique de programme à la sémantique opérationnelle de la machine. Son énoncé (ci-dessous) est ici légèrement informel faute d'avoir défini la sémantique de la machine. On le lit de la manière suivante : "si une spécification de la forme $\{w; P\} \rightsquigarrow \bullet$ est établie dans la logique de programme, sous des invariants $\boxed{I_0}$, ..., $\boxed{I_n}$, et pour un état initial de la machine $(regs_0, mem_0)$ qui satisfait P et les invariants, alors ces invariants sont préservés pour tout état ultérieur lors de l'exécution de la machine".

Adéquation
$$\frac{\boxed{I_0}, \ldots, \boxed{I_n} \vdash \{w; P\} \rightsquigarrow \bullet \qquad (regs_0, mem_0) \vDash I_0 * \ldots * I_n * \mathsf{pc} \Rrightarrow w * P \qquad (regs_0, mem_0) \longrightarrow^* (regs, mem)}{(regs, mem) \vDash I_0 * \ldots * I_n}$$

Pour le lecteur familier avec Iris, un point plus technique mais intéressant est que nos trois notions de spécification sont en fait définies à partir de la notion plus primitive de *plus faible précondition* (wp), fournie par Iris, et qui est définie directement en fonction de la sémantique de la machine. Dans un langage de haut niveau, wp est typiquement paramétré par une expression du langage. Dans le cadre de notre machine à capacités, cette notion d'expression n'existe pas :

les programmes sont des données ordinaires en mémoire. À la place, notre notion de wp est paramétrée par des "modes d'exécution", dont SingleStep, correspondant à l'exécution d'une unique instruction, et RepeatSingleStep, correspondant à une exécution complète de la machine.

Les définitions (ci-dessous) montrent que les spécifications pour une unique instruction ($\langle w_0; P\rangle \to \langle w_1; Q\rangle$) et pour une exécution complète ($\{w; P\} \rightsquigarrow \bullet$) correspondent alors à ces deux modes d'exécution de wp. Finalement, la spécification d'un fragment de code ($\{w_0; P\} \rightsquigarrow \{w_1; Q\}$) est définie en style "passage de continuation", d'après la spécification pour une exécution complète.

$$\begin{array}{lcl} \langle w_0; P\rangle \to \langle w_1; Q\rangle & \triangleq & \mathsf{pc} \mapsto w_0 * P \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{SingleStep}\ \{\mathsf{pc} \mapsto w_1 * Q\} \\ \{w; P\} \rightsquigarrow \bullet & \triangleq & \mathsf{pc} \mapsto w * P \mathrel{-\!\!*} \mathsf{wp}\ \mathsf{Repeat}\ \mathsf{SingleStep}\ \{\mathsf{True}\} \\ \{w_0; P\} \rightsquigarrow \{w_1; Q\} & \triangleq & \{w_0; P * \{w_1; Q\} \rightsquigarrow \bullet\} \rightsquigarrow \bullet \end{array}$$

4 Relation logique et théorème fondamental

Notre logique de programmes permet non seulement d'établir la correction fonctionnelle de code connu, mais est également utile pour définir ce qui est notre principe de raisonnement clef pour raisonner à propos de code inconnu.

On donne ainsi une définition de ce qui rend une valeur (un mot machine) "sûre à partager avec du code inconnu". Intuitivement, une valeur est sûre à partager avec un adversaire si elle se conforme à un contrat de sûreté des capacités : elle donne accès exactement à la mémoire sur laquelle elle possède une autorité (défini par son intervalle d'autorité et sa permission), et elle ne peut pas être utilisée pour augmenter cette autorité ou invalider un invariant de la logique.

La définition formelle est donnée en Figure 3. On définit simultanément la notion de "valeur sûre à partager" ($\mathcal{V}$) et "valeur sûre à exécuter" ($\mathcal{E}$).

— *Une valeur sûre à partager* ne donne accès transitivement qu'a des valeurs sûres à partager, ou du code sûr à exécuter (dans le cas d'une clôture).

— *Une valeur sûre à exécuter*, étant donné des valeurs sûres dans les registres, s'exécute sur la machine tout en préservant les invariants Iris (par définition de $\{\cdot;\cdot\} \rightsquigarrow \bullet$).

À proprement parler, cette définition est circulaire. Il est possible de l'énoncer en Iris car il s'agit d'une logique step-indexée, d'où l'utilisation de la modalité $\triangleright$ dans la définition, que le lecteur peut essentiellement ignorer ici.

Une capacité RW- donne accès en lecture et écriture à son intervalle : on ne peut que définir son contenu comme étant lui même sûr à partager. En revanche, une capacité avec une permission RO/RX ne peut pas être utilisée par du code inconnu pour changer les mots dans son intervalle ; dans ce cas, ceux-ci peuvent obéir à tout invariant (P) plus restrictif que $\mathcal{V}$.

Une capacité objet E est sûre à partager si le code qu'elle encapsule est sûr à exécuter. Celle-ci peut également être exécutée à tout instant : cette contrainte est représentée par la modalité $\square$, qui dans Iris s'interprète comme restreignant la définition de $\mathcal{V}(\text{E}, -)$ à être "toujours vraie", et ne pouvant donc être prouvée qu'en fonction d'invariants logiques.

Cette définition de sûreté est-elle triviale ? Autrement dit, la définition de sûreté donnée en Figure 3 ne serait-elle pas toujours vraie ? La réponse est non ! Toutefois, il n'est pas complètement évident de s'en convaincre.

En première approche, la définition de $\mathcal{E}(w)$ n'est pas triviale car elle nécessite de prouver que, partant de w, une exécution complète de la machine *préserve les invariants de la logique*. Cette contrainte n'est pas visible dans la définition, car implicite à la définition de la logique de programmes et au fonctionnement d'Iris, mais elle est cruciale. Par ailleurs, la définition de

$$\boxed{\mathcal{E}(w)} \triangleq \forall reg,\ \left\{w; \divideontimes_{(r,v)\in reg, r\neq \mathsf{pc}}\ r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$$

$$\boxed{\mathcal{V}(w)} \begin{cases} \mathcal{V}(z) & \triangleq \mathsf{True} \\ \mathcal{V}(\text{E}, b, e, a) & \triangleq \triangleright\, \square\, \mathcal{E}(\text{RX}, b, e, a) \\ \mathcal{V}(\text{RO}/\text{RX}, b, e, -) & \triangleq \divideontimes_{a\in[b,e[} \exists P,\ \boxed{\exists w,\ a \mapsto w * P(w)} * \triangleright\, \square\, \forall w,\ P(w) \mathrel{-\!\!*} \mathcal{V}(w) \\ \mathcal{V}(\text{RW}/\text{RWX}, b, e, -) & \triangleq \divideontimes_{a\in[b,e[} \boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)} \end{cases}$$

FIGURE 3 – Relation logique définissant la notion de "valeur sûre à partager".

$\mathcal{V}(w)$ n'est pas non plus triviale, car, par exemple dans le cas d'une capacité RW, elle impose que la permission $a \mapsto -$ pour chaque cellule mémoire a soit gouvernée par un invariant spécifique ($\boxed{\exists w,\ a \mapsto w * \mathcal{V}(w)}$). Or une permission "$a \mapsto -$" n'est pas duplicable. Toute cellule mémoire qui est donc gouvernée par un invariant plus spécifique ne peut donc pas être associée à une capacité sûre au sens de $\mathcal{V}$: on ne peut avoir qu'un seul exemplaire de $a \mapsto -$, qui ne peut donc pas faire partie de deux invariants différents.

Quel est donc un exemple de capacité qui n'est *pas* sûre ? Considérons une case mémoire d'adresse x initialisée à 0. Supposons alors qu'on alloue l'invariant Iris suivant : $\boxed{x \mapsto 0}$. Celui-ci exprime que x contiendra l'entier 0 pour le reste de l'exécution. Alors, intuitivement, une capacité $(\text{RW}, x, x+1, x)$ n'est pas sûre à partager avec un adversaire ! En effet, celui-ci pourrait l'utiliser pour écrire une valeur arbitraire à l'adresse x, invalidant l'invariant. Formellement, on ne peut prouver $\mathcal{V}(\text{RW}, x, x+1, x)$, car il n'est pas possible de créer l'invariant $\boxed{\exists w,\ x \mapsto w * \mathcal{V}(w)}$: la ressource pour la case mémoire x fait déjà partie de l'invariant $\boxed{x \mapsto 0}$. De même, on ne peut prouver $\mathcal{E}$ pour tout fragment de code qui écrit une valeur différente de 0 à l'adresse x, car on ne peut justifier dans la preuve la préservation de l'invariant lié à x.

Théorème fondamental. Le théorème fondamental (Théorème 1) est le résultat principal de ce travail : c'est un théorème non trivial, dont la preuve exige d'examiner tous les cas possibles de la sémantique de chaque instruction de la machine. Celui-ci établit que, selon notre définition, tout code "sûr à partager" est en fait également "sûr à exécuter". Ce théorème nous donne donc une spécification pour le comportement de code arbitraire. Tant qu'elle ne donne accès qu'à des mots mémoire sûrs (en particulier, des instructions arbitraires correspondent à des entiers et sont donc toujours sûres), alors une capacité est sûre à exécuter.

Théorème 1 (TFRL). *Soient* $p \in \text{Perm}, b, e, a \in \text{Addr}$. *Si* $\mathcal{V}(p, b, e, a)$, *alors* $\mathcal{E}(p, b, e, a)$.

Une autre interprétation du théorème fondamental est qu'il exprime que la machine "fonctionne bien". Exécuter les instructions ne peut créer plus d'autorité que ce qui était déjà disponible initialement ; le cas contraire serait un bogue de conception de la machine à capacités.

Le lecteur attentif aura remarqué que notre modèle ne distingue pas les capacités -X et celles sans X. Ceci est une conséquence directe du théorème fondamental ! Notre modèle exprime "l'autorité" qu'une portion de code a sur la mémoire. D'après le Théorème 1, être capable d'exécuter une portion de code ne donne pas d'autorité supplémentaire par rapport à seulement savoir le lire.

Pour récapituler, notre relation logique caractérise l'interface entre du code vérifié qui veut préserver des invariants sur un état interne ; et du code "externe" arbitraire dont on a suffisamment restreint les capacités. Le théorème fondamental nous donne une propriété de sûreté pour du code inconnu, et nous permet de vérifier du code connu qui appelle du code adversaire et potentiellement malicieux.

```
;; r1 : capacité vers une zone mémoire où
;;  écrire le code d'activation de la clôture
;; r3 : capacité vers l'adresse x
g: move r2 pc
   lea r2 23 ;; offset vers f
   subseg r2 f f_end ;; restreint au code de f
   ;; crtcls (emplacement) (code) (data)
   crtcls r1 r2 r3
   ;; r1 = clôture (capacité E), r2, r3 = 0
   jmp r0
g_end:

a: move r1 pc
   lea r1 7
   load renv r1
   lea r1 -1
   load r1 r1
   jmp r1
   data 0 ;; sera : capacité de code
   data 0 ;; sera : capacité de données
a_end:
```

```
;; renv : capacité vers l'adresse x
;; r2 : argument passé par l'appelant
;;     (censé être un entier)
f: move r1 pc ;; / r1: addresse de la fin
   lea r1 7   ;; \     du programme
   lt r3 r2 0 ;; a-t-on r2 ≥ 0?
   jnz r1 r3 ;; si non : quitter
   load r3 renv  ;; / si oui : l'ajouter
   add r3 r3 r2  ;; | à la mémoire privée
   store renv r3 ;; \ ...
   move renv 0 ;; / nettoyer les capacités
   move r1 0   ;; \ vers l'état privé
   jmp r0
f_end:
```

FIGURE 4 – Implémentation de notre composant vérifié.
g : code de création de la clôture ; f : corps de la clôture ; a : code d'activation de la clôture. Pour simplifier, on suppose que le code pour g et f se suivent en mémoire, i.e. g_end = f.

Il est important de noter que la distinction entre code connu et code adversaire est purement logique : elle n'existe pas à l'exécution. On peut avoir deux composants vérifiés séparément et qui ne se font mutuellement pas confiance : dans ce cas, du point de vue de la preuve de chaque composant, l'autre composant sera alors considéré comme le code adversaire.

5 Raisonner en présence de code inconnu : un exemple

(Le code et preuve présentés dans cette section correspondent aux fichiers `adder.v` et `adder_adequacy.v` dans le dossier `theories/examples/` de la formalisation Coq.)

Revenons sur l'exemple introduit en Section 2. Pour se simplifier légèrement la tâche et se passer d'une routine auxiliaire d'allocation mémoire, on suppose que notre composant vérifié est donné accès (exclusif) à une cellule mémoire x, et construit alors une clôture qui encapsule l'accès à x. À haut niveau, l'implémentation du composant est donc équivalente à :

$$(\lambda n.\ \text{if } n \geq 0 \text{ then } x := !x + n)$$

Alors, on veut vérifier la propriété suivante : *quelle que soit l'implémentation du composant adversaire, si x contient initialement un entier positif, à tout moment de l'exécution, la valeur de x reste positive.* En effet, l'adversaire n'a pas d'accès direct à x mais seulement à la clôture ci-dessus : si celle-ci est correctement implémentée à l'aide de capacités objet, alors l'adversaire ne peut modifier x que via des appels à la clôture, qui ne peut qu'augmenter la valeur de x.

Le code que l'on vérifie effectivement est une séquence d'instructions machine. Il apparaît en Figure 4, en syntaxe pseudo-assembleur. Le code spécifique au composant est composé de deux routines : g, exécutée initialement et créant la clôture, et f implémentant la clôture elle-même.

La Figure 5 illustre l'agencement de la mémoire avant et après l'exécution de g. Le code de g reçoit dans le registre r_2 une capacité vers x, crée la clôture encapsulant cette capacité,

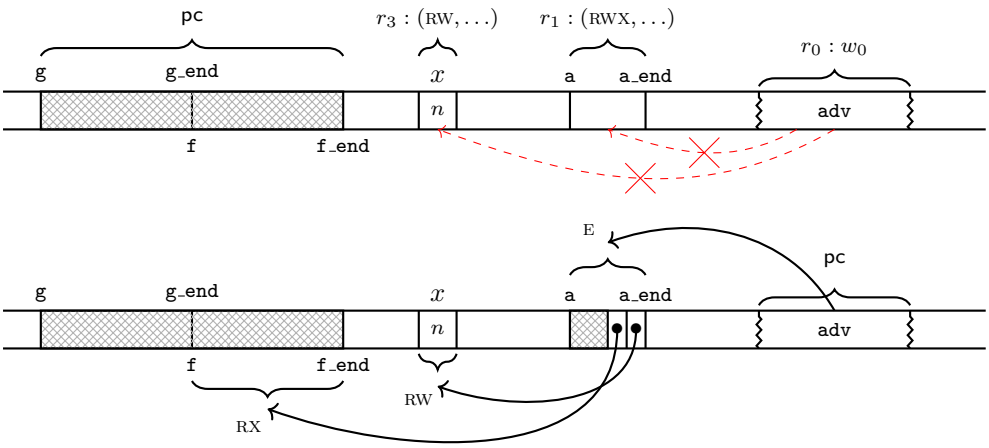

FIGURE 5 – Agencement de la mémoire : 1) initialement, et 2) après l'exécution de g.

et passe le contrôle à l'adversaire en sautant à la capacité dans la registre r_0, sur laquelle on ne sait rien. Pour créer la clôture, g utilise la macro-instruction `crtcls` (un alias pour une séquence d'instructions qu'on ne détaille pas ici). Celle-ci écrit en mémoire le code d'activation de la clôture, la capacité vers x, et la capacité vers le code de f, et crée une capacité objet (avec permission E) encapsulant le tout. Le code d'activation (qui est exécuté à chaque invocation de la clôture et passe le contrôle à f) est reproduit Figure 4 (en a). Il n'est pas spécifique à l'exemple considéré ici, et correspond seulement à la manière dont on implémente ici des clôtures.

Lorsque la clôture est invoquée par l'adversaire, le code d'activation (a) est exécuté. Celui-ci se contente de copier dans les registres les capacités pour x et f (stockées à la suite du code). Il invoque ensuite f, qui utilise la capacité vers x pour modifier sa valeur (le cas échéant), et prend ensuite soin de nettoyer les registres des capacités temporaires avant de rendre le contrôle à l'adversaire (par convention, r_0 contient le pointeur de retour).

L'étape suivante est d'énoncer et prouver une spécification pour chaque routine (Figure 6). On ne détaille pas les preuves : il s'agit ici d'un exercice standard de preuves de programmes. Les spécifications sont vérifiées en parcourant le code de f, g et a, et en utilisant successivement la spécification de chaque instruction machine rencontrée. Notons l'utilisation de plusieurs invariants : un pour le code de chaque programme (qui doit être en mémoire et accessible), et, plus important, un invariant contraignant la valeur stockée à l'adresse x à être un entier positif.

Il reste alors la partie la plus intéressante de la preuve : combiner les spécifications des routines individuelles avec la spécification du code inconnu (donnée par le Théorème 1), afin d'obtenir une spécification pour une exécution complète du système et finalement appliquer le théorème d'adéquation, obtenant ainsi que x contient bien un entier positif à tout instant, par préservation de l'invariant idoine. Les étapes principales du raisonnement sont comme suit.

Le but est de montrer la spécification suivante pour une exécution complète de la machine. Cette spécification devant être établie sous l'invariant logique $\boxed{\exists n, x \mapsto n \wedge n \geq 0}$, par le théorème d'adéquation, cela implique alors directement la propriété sur x voulue.

$$\boxed{[\mathsf{g}, \mathsf{g\_end}[\mapsto g_{instrs}]}, \boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}]}, \boxed{\exists n, x \mapsto n \wedge n \geq 0}$$

$$\vdash \left\{ \begin{array}{l} (\text{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g}); \\ \\ \end{array} \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\text{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto - * \\ r_3 \Mapsto (\text{RW}, x, x+1, x) * [\mathsf{a}, \mathsf{a\_end}[\mapsto [-] * \\ \mathcal{V}(w_0) * \ast_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0 .. r_3\}} r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

$$\boxed{[\mathsf{g}, \mathsf{g\_end}[\mapsto g_{instrs}}$$

$$\vdash \quad \left\{ (\mathrm{RX}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g});\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{RWX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto - * \\ r_3 \Mapsto (\mathrm{RW}, x, x+1, x) * [\mathsf{a}..\mathsf{a\_end}] \mapsto [-] \end{array} \right\} \rightsquigarrow$$
$$\left\{ w_0;\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (\mathrm{E}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}..\mathsf{a\_end}] \mapsto (a_{instrs} +\!\!+ [(\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] +\!\!+ [(\mathrm{RW}, x, x+1, x)]) \end{array} \right\}$$

$$\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}}, \boxed{\exists n,\, x \mapsto n \wedge n \geq 0}$$

$$\vdash \quad \left\{ (\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f});\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto - * r_2 \Mapsto k * r_3 \Mapsto - * \\ r_{\mathrm{env}} \Mapsto (\mathrm{RW}, x, x+1, x) \end{array} \right\} \rightsquigarrow$$
$$\left\{ w_0;\ \exists k'\, n',\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto 0 * r_2 \Mapsto k' * r_3 \Mapsto n' * \\ r_{\mathrm{env}} \Mapsto 0 \end{array} \right\}$$

$$\boxed{[\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ c_{code} +\!\!+ c_{data})}$$

$$\vdash \quad \left\{ (\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ r_1 \Mapsto - * r_{\mathrm{env}} \Mapsto - \right\} \rightsquigarrow \left\{ c_{code};\ r_1 \Mapsto c_{code} * r_{\mathrm{env}} \Mapsto c_{data} \right\}$$

FIGURE 6 – Spécifications pour les routines g, f et a.
On note g_{instrs}, f_{instrs} et a_{instrs} les listes de leurs instructions encodées en mots machine.

Cette spécification requiert les ressources nécessaires à l'exécution de g (cf. la précondition de g en Figure 6), mais aussi que w_0 (le pointeur vers le code inconnu) et les valeurs contenues dans le reste des registres soient des mots machines sûrs. Par exemple, on peut préalablement vérifier que w_0 est une capacité vers une zone de mémoire ne contenant que du code et des données (donc pas d'autres capacités), et initialiser le reste des registres à zéro : d'après la définition de $\mathcal{V}$, un entier est toujours sûr, de même qu'une capacité pointant sur une région contenant des entiers. En fait, par composition avec la spécification de g (Figure 6), il suffit de raisonner sur la continuation de g. Il suffit donc de montrer :

$$\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}}, \boxed{\exists n, x \mapsto n \wedge n \geq 0}$$

$$\vdash \quad \left\{ w_0;\ \begin{array}{l} r_0 \Mapsto w_0 * r_1 \Mapsto (E, \mathsf{a}, \mathsf{a\_end}, \mathsf{a}) * r_2 \Mapsto 0 * r_3 \Mapsto 0 * \\ [\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ [(\mathrm{RX}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f})] +\!\!+ [(\mathrm{RW}, x, x+1, x)]) * \\ \mathcal{V}(w_0) * \text{✻}_{(r,v) \in reg, r \notin \{\mathsf{pc}, r_0..r_3\}}\, r \Mapsto v * \mathcal{V}(v) \end{array} \right\} \rightsquigarrow \bullet$$

Or, puisque w_0 est sûr (on a $\mathcal{V}(w_0)$), d'après le Théorème 1, il est sûr à exécuter (donc on a $\mathcal{E}(w_0)$). En dépliant la définition de $\mathcal{E}$, on obtient le but voulu (la spécification ci-dessus), à condition de pouvoir prouver que la valeur de chaque registre est elle-même sûre. La preuve est immédiate pour tous les registres sauf r_1, qui pointe sur notre clôture : puisque l'on partage celle-ci avec le code inconnu, il nous incombe de prouver qu'elle est sûre.

Il reste donc à prouver $\mathcal{V}(\mathrm{E}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$. On prend soin de placer les ressources correspondant au code d'activation de la clôture (entre a et a_end) dans un nouvel invariant. Alors, sans rentrer dans les détails liés aux modalités $\Box$ et $\triangleright$, il suffit d'établir $\mathcal{E}(\mathrm{RX}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$.

En composant les spécifications de a et f, on peut vérifier que celles-ci ne supposent rien à propos des valeurs contenues initialement dans les registres (dont on sait seulement qu'elles sont sûres) ; et que de même, les valeurs contenues dans les registres après l'exécution de f sont

sûres (soit ce sont des entiers, soit elles n'ont pas été modifiées). On a donc :

$$\begin{array}{l}\boxed{[\mathsf{f}, \mathsf{f\_end}[\mapsto f_{instrs}]}, \boxed{\exists n,\, x \mapsto n \wedge n \geq 0}, \\ \boxed{[\mathsf{a}, \mathsf{a\_end}[\mapsto (a_{instrs} +\!\!+ (\textsc{rx}, \mathsf{f}, \mathsf{f\_end}, \mathsf{f}) +\!\!+ (\textsc{rw}, x, x+1, x))}\end{array}$$
$$\vdash \quad \begin{array}{l}\{(\textsc{rx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a});\ \ r_0 \Mapsto w_0 * \mathcal{V}(w_0) * \mathop{\Large\ast}_{(r,v)\in reg, r\neq \mathsf{pc}, r_0} r \Mapsto v * \mathcal{V}(v)\ \} \rightsquigarrow \\ \{w_0;\ \ \mathop{\Large\ast}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\ \}\end{array}$$

Pour établir $\mathcal{E}(\textsc{rx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$, il suffit alors de raisonner sur la continuation de f (f retournant à l'adversaire en exécutant le pointeur de retour w_0) ; c'est à dire, il suffit d'établir : $\left\{w_0; \mathop{\Large\ast}_{(r,v)\in reg, r\neq \mathsf{pc}} r \Mapsto v * \mathcal{V}(v)\right\} \rightsquigarrow \bullet$. Or, w_0 est supposé sûr (par définition de $\mathcal{E}$) : on a $\mathcal{V}(w_0)$. Le Théorème 1 donne alors $\mathcal{E}(w_0)$, ce qui est exactement ce qu'il restait à prouver. Qed.

Théorème final. Via le théorème d'adéquation, on obtient alors le théorème suivant pour l'exécution de la machine dans notre scénario, exprimé en fonction de la sémantique opérationnelle :

Théorème 2 (Exécution correcte de la machine)**.** *En partant d'un état initial de la machine* (reg, mem) *où :*

- *mem a été initialisée avec le code de* g *et* f*, et du code inconnu pour l'adversaire (entre les adresses* adv *et* $\mathsf{adv\_end}$*) (Figure 5) ;*
- $reg(\mathsf{pc}) = (\textsc{rx}, \mathsf{g}, \mathsf{f\_end}, \mathsf{g})$, $reg(r_0) = (\textsc{rwx}, \mathsf{adv}, \mathsf{adv\_end}, \mathsf{adv})$, $reg(r_1) = (\textsc{rwx}, \mathsf{a}, \mathsf{a\_end}, \mathsf{a})$, $reg(r_3) = (\textsc{rw}, x, x+1, x)$*, et* $reg(r) \in \mathbb{Z}$ *dans les autres cas ;*
- $mem(x)$ *est un entier positif.*

Alors, pour tous reg', mem'*, si* $(reg, mem) \longrightarrow^* (reg', mem')$ *alors* $mem'(x)$ *est un entier positif.*

En d'autres termes, pour une machine correctement initialisée, alors l'invariant établi dans la logique à propos de x est vrai à chaque étape de l'exécution : à tout instant, la valeur stockée en mémoire à l'adresse x est un entier positif.

Ce théorème est-il satisfaisant ? On peut anticiper deux commentaires possibles à propos du théorème ci-dessus, qui suggéreraient que celui-ci n'est pas aussi général que l'on pourrait vouloir, et auxquelles on répond ci-dessous.

Le théorème fait-il des hypothèses trop spécifiques à propos de l'état initial de la machine ? Le Théorème 2 ne détaille en effet pas la manière dont la mémoire est initialisée avec le code pour g et f, ou le code adversaire. De même, on peut se demander d'où viennent les capacités que l'on suppose ici être présentes dans les registres pc, r_0, r_1 et r_3. Quel est l'état initial d'une machine à capacités, immédiatement après sa mise sous tension ?

Les détails précis dépendent de l'implémentation matérielle ; mais invariablement, une machine à capacités doit initialement fournir une "capacité omnipotente" donnant autorité sur l'ensemble de la mémoire (ici, ce serait $(\textsc{rwx}, 0, \mathsf{addr\_max}, 0)$). C'est alors le rôle du code de démarrage de la machine que de restreindre et diviser cette capacité et de la distribuer aux différents composants du programme. Le Théorème 2 suppose que l'on se place immédiatement après l'exécution du code de démarrage de la machine, afin de s'abstraire des détails d'implémentation liés à l'initialisation de la machine. Étant donné une implémentation concrète du code de démarrage, ce serait un exercice standard de vérification de programmes que de vérifier sa correction et le connecter au théorème établi ici.

Ne serait-il pas plus général de commencer par l'exécution du code inconnu plutôt que du code connu ? Pour les raisons détaillées précédemment, il est nécessaire de faire confiance au code

exécuté au démarrage de la machine. Si celui-ci est considéré inconnu (ou un adversaire), alors il est impossible de garantir quoi que ce soit, puisque celui-ci a accès à une capacité omnipotente. Il est donc nécessaire d'exécuter une partie du code d'initialisation au démarrage de la machine (ici, la création des clôtures), avant de passer le contrôle à l'adversaire. Le scénario détaillé ici requiert du code de démarrage de la machine que celui-ci ait déjà préparé un certain nombre de régions mémoires pour son fonctionnement (la cellule à l'adresse x et la région pour le code d'activation). L'exemple du compteur présenté ensuite en Section 6 a des prérequis différents pour le code de démarrage : l'implémentation du compteur (code d'initialisation et clôtures) alloue elle même la mémoire nécessaire à la demande, mais nécessite que le système inclue une routine supplémentaire (`malloc`) implémentant un allocateur mémoire : le travail du code de démarrage est alors de fournir au compteur un pointeur vers cette routine.

On peut donc imaginer divers scénarios distribuant différemment les responsabilités entre code exécuté initialement ou invoqué via le code inconnu, mais il est nécessaire dans tous les cas d'exécuter une section de code d'initialisation connu (et vérifié) au démarrage de la machine.

6 Études de cas

En plus de l'exemple détaillé dans la section précédente, nous avons implémenté et vérifié les exemples suivants. Pour simplifier, on se contente ici de présenter une version haut niveau de leur implémentation, la formalisation Coq contenant les détails. Dans chaque cas, on vérifie que les assertions n'échouent pas. Plus précisément, chaque exemple est muni d'une routine auxiliaire "assert", maintenant une cellule mémoire interne, initialisée à 0 et mise à 1 en cas d'appel à assert avec une condition fausse. On prouve alors que cette cellule contient 0 à tout moment de l'exécution.

Compteur pouvant être incrémenté, lu et réinitialisé On vérifie un compteur comportant une référence privée (la valeur actuelle du compteur), et exposant trois clôtures, pour respectivement incrémenter le compteur, lire sa valeur, et la réinitialiser à zéro.

$$\begin{array}{l}\text{let } x = \text{alloc } 0 \text{ in} \\ \quad (\lambda().\ x := !x + 1),\ (\lambda().\ \text{assert } (!x \geq 0);\ !x),\ (\lambda().\ x := 0)\end{array}$$

Les trois points d'entrée sont vérifiés indépendamment, et utilisent le même invariant à propos de x que dans l'exemple précédent. Contrairement à l'exemple précédent, la mémoire pour x est ici allouée dynamiquement, en faisant appel à une routine auxiliaire "alloc". Le théorème final requiert donc seulement que la routine d'allocation soit initialisée en mémoire, et ne demande pas au code de démarrage de réserver de la mémoire pour le fonctionnement du compteur. Comme précédemment, les clôtures sont passées à un contexte adversaire composé d'instructions arbitraire, et on peut montrer que l'assertion n'échoue pas. Pour plus de généralité, on prouve également que le point d'entrée de la routine "alloc" est sûr, et on donne accès à "alloc" à l'adversaire.

Partage d'une capacité read-only (RO) On vérifie un exemple illustrant l'utilisation d'une capacité RO (en lecture seule).

$$
\begin{array}{l}
\text{let } x = \text{alloc } 1 \text{ in} \\
\text{let } y = \text{restrict } x \text{ RO in} \\
\text{unknown\_code}(y); \\
\text{assert } (!x = 1); \\
\text{halt}()
\end{array}
$$

Dans cet exemple, "unknown_code" est une fonction inconnue, et "restrict x RO" restreint la capacité x (qui a la permission RWX car renvoyée par alloc) à une permission RO. Ici, l'adversaire correspond à la fonction unknown_code. Selon le modèle, on peut raisonner sur l'exécution de foo tant que l'on sait que unknown_code est une capacité valide (concrètement : elle n'a pas d'accès direct à x). Dans ce cas, on sait (soit par la définition de validité d'une capacité E, soit par le théorème fondamental) que unknown_code est sûr à exécuter, tant que les valeurs partagées avec celui-ci sont valides. On doit donc montrer que la capacité y est valide. D'après la définition de validité (Figure 3), on doit donc montrer :

$$\exists P, \boxed{\exists w,\, y \mapsto w * P(w)} * \triangleright \Box \forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)$$

Autrement dit, on peut décrire la mémoire pointée par y en choisissant un prédicat P *au moins aussi restrictif* que $\mathcal{V}$. Intuitivement, la valeur stockée doit être valide, car elle peut être lue par l'adversaire, mais celui-ci ne peut la modifier, donc il est possible de garantir un invariant plus fort. Pour montrer que l'assertion n'échoue pas, on choisit ici $P(w) \triangleq w = 1$. Ce prédicat satisfait la condition $\triangleright \Box \forall w,\, P(w) \mathrel{-\!\!*} \mathcal{V}(w)$ (1 est toujours un mot valide), et nous permet de montrer que x pointe vers l'entier 1 après l'appel au code inconnu.

7 Travaux connexes

Cet article présente une version simplifiée de la méthodologie précédemment mise en place par les auteurs pour raisonner à l'aide d'Iris à propos d'une convention d'appel sûre [GGVS+21]. L'utilisation seule des capacités objets ne permet en effet pas d'implémenter au niveau assembleur des appels de fonction (vers un adversaire) qui soient fidèles à la notion d'appel de fonction d'un langage de haut niveau. Si les capacités objets permettent d'implémenter une forme d'encapsulation d'état local, elles ne garantissent pas que l'ordre des appels et retours de fonction soit bien parenthésé. Notamment, elles n'empêchent pas un adversaire d'invoquer plusieurs fois un pointeur de retour fourni par un appelant (chose impossible dans un langage de haut niveau sans opérateurs de contrôle).

Skorstengaard et al. [SDB19] montrent qu'il est possible d'implémenter une convention d'appel fidèle en utilisant un type additionnel de capacités "locales". L'article en question suit une méthodologie similaire à celle décrite ici, définissant une relation logique caractérisant une certaine notion de sûreté. Les preuves ne sont toutefois pas mécanisées et les détails de la relation logique sont relativement difficile à suivre.

L'article ultérieur de Georges et al. [GGVS+21] introduit d'une part un nouveau type de capacités ("non-initialisées") pour améliorer l'efficacité de la convention d'appel de Skorstengaard et al., et utilise Iris pour formuler une définition de sûreté comme une relation logique et mécaniser les preuves correspondantes. L'utilisation d'Iris permet la relation logique d'être exprimée de façon plus concise et à un plus haut niveau que dans le travail de Skorstengaard

et al. En comparaison avec l'article présent, celle-ci est toutefois significativement plus compliquée que celle présentée ici, car plus expressive : elle permet de raisonner sur des propriétés de "bon parenthésage" des appels de fonction.

Certains langages de haut niveau permettent également l'utilisation de "capacités objet" pour protéger un état privé lors de l'interaction avec du code arbitraire. (Typiquement implémentées grâce à l'encapsulation fournie par les clôtures du langage.) Devriese et al. [DBP16] définissent une notion de "sûreté des capacités" pour un sous-ensemble de Javascript (incluant une notion d'effets observables) à l'aide d'une relation logique, et montrent qu'elle permet de raisonner sur différents exemples concrets. En terme d'expressivité, leur relation logique est plus proche de celle pour une convention d'appel sûre [GGVS+21] que celle présentée ici. Elle est toutefois énoncée sur papier et n'a pas été mécanisée.

Plus récemment, Swasey et collaborateurs [SGD17] présentent une logique de programme permettant de raisonner sur une forme de "capacités objets" dans un langage de haut niveau. Leur méthodologie est extrêmement similaire à la notre : les principes de raisonnement logiques sont essentiellement les mêmes, mais ils se placent dans le cadre d'un langage de haut niveau, là où nous utilisons des capacités objet sur une machine à capacités.

Par exemple, Swasey et al. définissent deux prédicats pour décrire une référence : un prédicat pour des références "haute intégrité" ($\ell \hookrightarrow v$) et un pour des références "faible intégrité" ($\mathsf{lowloc}\ v$). Les premières donnent accès exclusif à la référence et ne sont pas partageables avec un adversaire ; les deuxièmes sont partageables avec un adversaire mais ne peuvent être utilisées que pour lire et écrire des valeurs "faible intégrité". Dans notre formalisation, une référence "haute intégrité" correspond alors à une ressource mémoire $a \mapsto w$, et une référence "faible intégrité" correspond à l'invariant utilisé dans la définition de $\mathcal{V}$: $\boxed{\exists w, a \mapsto w * \mathcal{V}(w)}$. Via cette correspondance, nos définitions satisfont les mêmes règles de raisonnement que celles énoncées par Swasey et al. ; en particulier, les différents "object capability patterns" qu'ils vérifient seraient également implémentables et vérifiables de manière similaire dans le cadre d'une machine à capacités.

Nienhuis et al. [NJB+20] vérifient formellement un certain nombre de propriétés "architecturales" des machines à capacités CHERI. Il s'agit d'un effort de formalisation conséquent : les auteurs considèrent une sémantique détaillée et réaliste de CHERI, significativement plus complexe que le modèle minimal que l'on utilise ici. L'approche de Nienhuis et al. est différente de la notre : ceux-ci énoncent les propriétés de sécurité qu'ils vérifient comme des propriétés de trace, en fonction d'une trace d'"actions abstraites" décrivant les capacités transitant dans la machine. Cette approche permet de formuler les propriétés voulues de façon très concrète et explicite. Par exemple, ceux-ci énoncent et prouvent une propriété de "monotonicité des capacités" : lors de l'exécution, l'autorité des capacités accessibles ne peut pas augmenter (autrement dit, la machine n'autorise pas à forger des capacités). Cela semble être une propriété raisonnable et nécessaire au bon fonctionnement de la machine à capacités. Pourtant, formellement, cette propriété est invalidée par les appels entre composants (dans notre cas, sauter à une capacité E). La propriété prouvée par Nienhuis et al. est donc restreinte à un fragment de trace ne comportant pas d'appel à un composant séparé. Notre méthodologie est moins explicite, mais plus expressive. L'énoncé (très extensionnel) de notre théorème fondamental, qui peut être vu comme un théorème de "bon fonctionnement de la machine", est difficile à comprendre en terme de la sémantique opérationnelle de la machine. Malgré tout, celui-ci permet *in fine* de raisonner sur une exécution complète de la machine avec un nombre arbitraire d'appels entre composants différents.

Remerciements Merci à Léon Gondelman et Pierre Pradic pour les commentaires et remarques sur des versions précédentes de ce document.

Références

[CKD94] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 319–327. ACM, 1994.

[DBP16] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about Object Capabilities Using Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*. IEEE, 2016.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3) :143–155, March 1966.

[GGVS+21] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and Provable Local Capability Revocation using Uninitialized Capabilities. In *POPL*, 2021. (Conditionally accepted).

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28 :e20, 2018.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[NJB+20] Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security : Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP)*, May 2020.

[SDB19] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a Machine with Local Capabilities : Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems*, 42(1) :5 :1–5 :53, December 2019.

[SGD17] David Swasey, Deepak Garg, and Derek Dreyer. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM, 2017.

[WNW+16] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera. Fast Protection-Domain Crossing in the CHERI Capability-System Architecture. *IEEE Micro*, 36(5) :38–49, September 2016.

[WNW+19] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions : CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.