



L'interprète, le JIT et la licorne

Frédéric Recoules, Sébastien Bardin

► To cite this version:

Frédéric Recoules, Sébastien Bardin. L'interprète, le JIT et la licorne. 35es Journées Francophones des Langages Applicatifs (JFLA 2024), Jan 2024, Saint-Jacut-de-la-Mer, France. hal-04407147

HAL Id: hal-04407147

<https://hal.science/hal-04407147>

Submitted on 22 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

L'interprète, le JIT et la licorne

Frédéric Recoules et Sébastien Bardin

Université Paris Saclay, CEA, List
Saclay, France
frederic.recoules@cea.fr sebastien.bardin@cea.fr

L'exécution symbolique, une méthode populaire d'analyse de programmes, a du mal à passer à l'échelle sur de gros codes. Outre les traditionnels problèmes d'explosion de chemins et de résolution des prédicats logiques, d'aucuns se plaignent du coût d'interprétation du langage cible et se tournent par conséquent vers la compilation, qu'elle soit statique ou à la volée, pour améliorer les performances. Sceptiques mais non moins curieux, nous saisissons ce prétexte pour rajouter un peu de compilation à la volée dans le moteur d'exécution symbolique de la plateforme BINSEC, écrite en OCaml. Permettez-nous ainsi d'introduire JITPSI, une petite bibliothèque qui tire son inspiration de MetaOCaml et ocaml-jit. JITPSI concourt à compiler *harmonieusement* en x86-64, directement depuis OCaml, une séquence d'appels de fonctions. Nous utilisons JITPSI pour transformer à la volée l'interpréteur d'arbres syntaxiques abstraits de BINSEC en sous-programmes filetés (*threaded code*). Notre intention première est ici d'améliorer la résolution du défi de rétro-ingénierie licorne issu du *France CyberSecurity Challenge 2022* proposé par l'ANSSI ; soit une accélération d'environ 40%.

1 Introduction

L'exécution symbolique [CS13] est une analyse précise mais coûteuse. L'analyse évalue le programme avec des entrées symboliques et construit le long de chaque chemin un prédicat logique exprimant l'ensemble des valeurs d'entrée qui mènent à ce chemin. La résolution de ces prédicats se fait traditionnellement à l'aide de solveurs SMT (*Satisfiability Modulo Theory*), ce qui permet de vérifier si un point du programme est atteignable et d'obtenir un exemple de valeurs concrètes le cas échéant. Ce procédé prend toutefois un temps considérable. Pour ne rien arranger, l'opération est répétée un grand nombre de fois. Le nombre de chemins croît en effet exponentiellement avec le nombre de conditions rencontrées. Le raisonnement logique représente ainsi généralement la quasi-totalité du temps d'analyse, au point que toute considération de performance en dehors de l'amélioration de ces deux facteurs est d'ordinaire considérée comme de l'optimisation prématurée.

La combinaison avec le *fuzzing* [MZH20] est venue rebattre les cartes. Il s'est développé une forme particulière d'exécution symbolique [SGS⁺16] qui se concentre sur une seule trace d'exécution et vise à inverser un petit nombre de conditions, souvent rendues plus simples à l'aide de sous-approximations (concrétisations). Dans ce contexte où il n'y a plus d'explosion de chemins ni de raisonnements interminables, le coût d'interprétation du langage peut largement devenir prédominant. De nouvelles approches basées sur la spécialisation du moteur symbolique pour un programme donné ont ainsi vu le jour, que ce soit à base de compilation statique [PF20, WJG⁺23] ou d'instrumentation dynamique [YLX⁺18, PF21].

Problème. L'interprétation du langage peut devenir le goulot d'étranglement de l'analyse, même en dehors de la combinaison avec le *fuzzing*. Nous avons pu expérimenter cette « inversion des coûts » dans la plateforme d'analyse de code binaire BINSEC [DB15] lors du *France CyberSecurity Challenge 2022*. Tenter de résoudre un défi de rétro-ingénierie consiste à rechercher des valeurs d'entrée qui déclencheront « la » bonne sortie du programme. Bien que souvent mise à mal par des contre-mesures, l'exécution symbolique se prête plutôt bien à l'exercice. Ici, la subtilité du défi licorne [Ver22] réside dans son utilisation d'Unicorn [ND15] (machine virtuelle x86-64) pour émuler du code ARM caché. Or, si le prédicat de chemin de ce dernier est, *in fine*, trivial à résoudre pour un solveur SMT (une milliseconde), construire ce prédicat nécessite de propager symboliquement les entrées du programme le long d'une trace d'exécution d'environ un milliard d'instructions (majoritairement issues de la virtualisation). Dans sa version 0.6.0, BINSEC prenait jusqu'à trois heures pour en arriver à bout.

Objectif. Bien que BINSEC ait fait de gros progrès depuis (la version 0.8.0 est 15 fois plus rapide sur cette trace), nous souhaitons réduire encore davantage le coût d'interprétation de notre langage intermédiaire à l'aide de compilation à la volée. Nous nous intéressons particulièrement à l'évaluation des blocs de base, éléments simples qu'il nous est facile d'optimiser agressivement. Nous voulons spécialiser, au sens de la projection de Futamura [Fut99], la boucle d'évaluation récursive des instructions pour chacun des blocs rencontrés. En remplaçant le filtrage par motif (*pattern matching*) par des appels directs, nous espérons réduire le nombre d'indirections afin d'atteindre le même niveau de performance que si nous avions écrit ce bloc de base à la main sous la forme de sous-programmes filetés (*Subroutine Threaded Code* [Bel73, SYZZ⁺14]). Dans les cas favorables, cette forme permet également de transformer des variables temporaires du programme interprété en variables locales OCaml, économisant ainsi des opérations d'écriture et de lecture dans l'environnement symbolique.

Pour ce faire, nous avons besoin d'un moyen de dérouler la boucle d'évaluation récursive de BINSEC et d'assembler la séquence d'appels aux fonctions de manipulation de l'environnement symbolique sous forme d'une fonction OCaml native.

Défis. Malgré l'apparente simplicité de la tâche, les solutions existantes ne sont pas adaptées. Nous écartons d'emblée l'utilisation de *llvm* [LA04] en raison des spécificités du langage OCaml, notamment, de sa convention d'appel atypique et de sa gestion du glaneur de cellules (*garbage collector* [DL93]). *MetaOCaml* [Kis18] est un projet qui avait tout du candidat idéal, mais deux points négatifs viennent ternir le tableau :

1. *MetaOCaml* génère *littéralement* du code OCaml à compiler plus tard, ce qui rend le programme dépendant à l'exécution de la disponibilité des interfaces (*cmi*) – son utilisation dans un *foncteur*, dont le résultat n'est connu qu'au cours de l'exécution, est par ailleurs une difficulté que nous ne sommes humblement pas arrivés à surmonter ;
2. la génération du code machine est quelque peu inefficace, en particulier pour les petites fonctions – *MetaOCaml* passe en effet par les étapes classiques du compilateur OCaml, invoque le processus assembleur (*as*) puis charge le résultat en mémoire (*Dynlink*).

Le projet *ocaml-jit* résout quant à lui ce second inconvénient. Son objectif est d'accélérer l'interpréteur interactif de code OCaml (*utop*), ce qu'il fait en générant le code machine x86-64 directement en mémoire avant de le lier à l'application sans jamais avoir recours à un processus externe. Il nous manque cependant toujours une interface programmatique permettant d'assembler une nouvelle fonction à partir des valeurs accessibles durant l'exécution sans passer par du code OCaml textuel.

Proposition. Nous remédions à ce manque grâce à un petit langage dédié [FP11] que nous exposons dans une bibliothèque à travers les primitives *lambda*, *const*, *apply* et *return*.

En se basant sur les couches basses du compilateur OCaml, la bibliothèque JITPSI (*Just In Time Partial Specialization for Interpreter*) permet de générer du code x86-64 qui

s'intègre avec harmonie à son environnement, respectant sa convention d'appel et son modèle mémoire.

La conception de cette bibliothèque répond aux deux critères suivants :

- *La simplicité.* Que ce soit à l'utilisation, avec les garanties de la sûreté du typage, ou lors de son développement, en choisissant le niveau d'abstraction le plus adapté dans le compilateur OCaml ([Lambda](#)) ;
- *L'efficacité.* En s'inspirant de la génération de code d'ocaml-jit pour assembler la fonction et allouer sa fermeture (*closure*) sans quitter le monde d'OCaml.

Contributions. En résumé, cet article fait état des contributions suivantes :

- la conception de la bibliothèque JITPSI qui permet d'assembler programmatiquement le résultat de la composition de plusieurs appels de fonctions et de produire un code natif aussi performant, *si ce n'est plus*, que s'il avait été écrit en OCaml à la main ;
- la preuve de concept de son application dans le moteur d'exécution symbolique de BINSEC avec un effet positif et notable sur (*au moins*) l'exemple licorne, défi de rétro-ingénierie issu du *France CyberSecurity Challenge 2022*.

Les codes de JITPSI et de BINSEC sont accessibles sur GitHub, respectivement à <https://github.com/recoules/jitpsi> et <https://github.com/binsec/binsec/tree/jitpsi>.

Discussion. Nous abordons le thème de la compilation à la volée avec le prisme restreint de l'exécution symbolique. Cet article est avant tout l'occasion de partager le fruit de diverses expérimentations qui nous tenaient à cœur, bien plus que ce qu'elles ne se justifiaient. Pourtant, nous pensons que la compilation à la volée peut avoir de nombreux cas d'usages et nous espérons relancer l'intérêt de la communauté pour cette thématique mais aussi récolter de précieux retours sur l'utilité de JITPSI et les pistes d'améliorations pour répondre à de nouveaux besoins.

2 Exemple et motivations

Illustrons tout d'abord le rôle de JITPSI à travers la spécialisation d'un interpréteur jouet. La représentation intermédiaire de ce petit interpréteur de code x86-64 est donnée en Figure 1a. Un programme est représenté sous forme d'un graphe de micro-instructions permettant de manipuler un état mémoire composé de différents registres et d'un drapeau de condition. Le langage permet de calculer des expressions simples à base de registres et de constantes et de tester si une expression est égale à zéro. Le langage inclut également l'instruction [Builtin](#). Elle permet d'appeler une fonction OCaml en charge de mettre à jour l'état du programme. Elle pourra servir à réaliser des tâches complexes ou difficilement exprimables dans ce langage, par exemple la division, ou comme nous allons le voir, à optimiser l'exécution d'une suite d'opérations connue. Le code de la Figure 1b permet d'évaluer le résultat concret d'un appel de fonction.

Pour la suite, nous allons nous intéresser à la fonction `fibonacci` dont le code assembleur est donné en Figure 3a, et dont la sémantique dans notre langage apparaît dans la Figure 3b.

V0. Afin d'exécuter cette fonction dans notre petit interpréteur, nous pouvons traduire littéralement cette sémantique en termes OCaml comme défini dans la Figure 2a. En moyenne, cette première mouture calcule le 92^{ème} élément de la suite de `fibonacci` en 66 mille cycles d'horloge (unité sans importance dans l'absolu).

V1 et V2. Commençons notre quête de performance. Au vu de la structure du code, la majeure partie du temps se déroule dans le corps de la boucle (instructions i4 à i9). Ces instructions génèrent toujours la même séquence d'invocation des primitives de notre langage (`R.find`, `R.add`, etc.). En revanche, notre fonction d'évaluation se voit obligée de

```

module V = Stdint.Uint64
type register = Rax | Rdx | Rdi | Tmp0
type operator = Plus | Minus
type expression =
  | Cst of V.t
  | Reg of register
  | Op of operator * expression * expression
module R = Map.Make
(struct
  type t = register
  let compare = compare
end)
type instruction =
  | SetR of register * expression * instruction
  | SetZ of expression * instruction
  | IfZ of instruction * instruction
  | Ret of register
  | Builtin of
    (V.t R.t -> bool -> instruction -> V.t) *
    instruction

type code = Fun of instruction

let rec eval regs = function
  | Cst v -> v
  | Reg r -> R.find r regs
  | Op (Plus, x, y) ->
    V.add (eval regs x) (eval regs y)
  | Op (Minus, x, y) ->
    V.sub (eval regs x) (eval regs y)
let rec step regs z = function
  | SetR (r, e, i) ->
    step (R.add r (eval regs e) regs) z i
  | SetZ (e, i) ->
    step regs V.(compare (eval regs e) zero = 0) i
  | IfZ (i, i') ->
    if z then step regs z i else step regs z i'
  | Ret r -> R.find r regs
  | Builtin (f, i) -> f regs z i
let run rdi (Fun i) =
  step (R.singleton Rdi rdi) false i

```

(a) Représentation intermédiaire

(b) Fonctions d'évaluation

Figure 1. Interpréteur simplifié de code x86-64 écrit en OCaml

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and i4 = SetR (Tmp0, Reg Rdx, i5)
and i5 = SetR (Rdx, Reg Rax, i6)
and i6 = SetR (Rax, Reg Tmp0, i7)
and o1 = Op (Plus, Reg Rax, Reg Rdx)
and i7 = SetR (Rax, o1, i8)
and o2 = Op (Minus, Reg Rdi, Cst V.one)
and i8 = SetR (Rdi, o2, i9)
and i9 = SetZ (Reg Rdi, i10)
and i10 = IfZ (i6, i4)
and i11 = Ret Rax

```

(a) Définition V0

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop r _ i =
  let v0 = V.zero in
  let v1 = V.one in
  let v2 = R.find Rdx r in
  let v3 = R.find Rax r in
  let v4 = R.find Rdi r in
  let v5 = V.add v2 v3 in
  let v6 = V.sub v4 v1 in
  let r0 = R.add Rdx v3 r in
  let r1 = R.add Rax v5 r0 in
  let r2 = R.add Rdi v6 r1 in
  let z0 = V.compare v6 v0 = 0 in
  step r2 z0 i
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(c) Définition V2

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop r _ i =
  let v0 = R.find Rdx r in
  let r0 = R.add Tmp0 v0 r in
  let v1 = R.find Rax r0 in
  let r1 = R.add Rdx v1 r0 in
  let v2 = R.find Tmp0 r1 in
  let r2 = R.add Rax v2 r1 in
  let v3 = R.find Rax r2 in
  let v4 = R.find Rdx r2 in
  let v5 = V.add v3 v4 in
  let r3 = R.add Rax v5 r2 in
  let v6 = R.find Rdi r3 in
  let v7 = V.one in
  let v8 = V.sub v6 v7 in
  let r4 = R.add Rdi v8 r3 in
  let v9 = R.find Rdi r4 in
  let va = V.zero in
  let z0 = V.compare v9 va = 0 in
  step r4 z0 i
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(b) Définition V1

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop = compile V0.[ i4; i5; i6; i7; i8; i9 ]
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(d) Définition V3

Figure 2. Représentation intermédiaire de la fonction fibonacci en OCaml

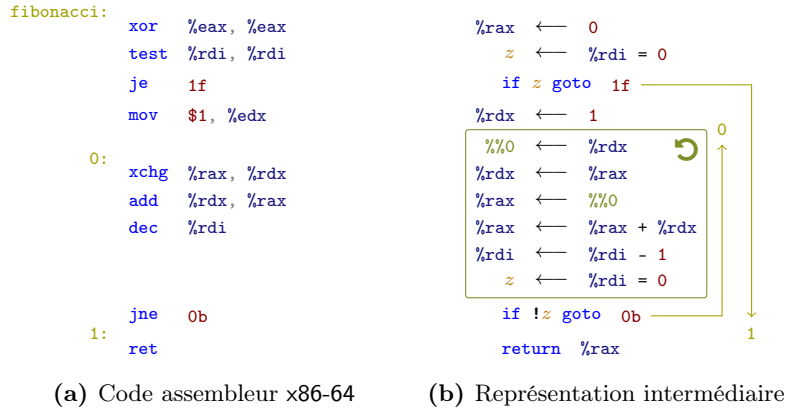


Figure 3. Calcul de la suite de Fibonacci

filtrer les différents motifs pour savoir quelle primitive appeler. Or, il est bien connu que les indirections sont les grandes ennemies des interpréteurs [RLV⁺96]. La seconde version présentée dans la Figure 2b supprime le filtrage de motifs et déroule manuellement le corps de la boucle. Pour aller plus loin, nous pouvons également observer que certaines invocations de primitives se révèlent superflues au regard du bloc dans son ensemble. Aussi, il est par exemple inutile de créer la valeur `v3` en allant lire la variable `%rax` car nous venons d’y écrire la valeur `v2` (lecture sur écriture). Or, maintenant que nous n’allons plus lire l’environnement pour récupérer `v3`, l’écriture de `v2` dans `%rax` devient inutile. En effet, une nouvelle écriture dans `%rax` a lieu plus tard et seule cette dernière sera retenue une fois le bloc exécuté (écriture sur écriture). En déroulant le raisonnement, nous aboutissons à la version de la Figure 2c. Cette version est plus compacte, utilise moins de primitives et a complètement fait disparaître la variable temporaire `%%0`. Tout cela est rendu possible grâce à une capacité que notre boucle d’évaluation n’a pas : nous pouvons garder autant de résultats intermédiaires que nous le voulons sans les écrire dans l’environnement du programme. Les performances sont également au rendez-vous. En moyenne, cette version produit son résultat en 39 mille cycles, soit une accélération d’environ 70%.

V3. *Au sein de cet interpréteur*, il nous paraît difficile de faire beaucoup mieux que la version précédente. En revanche, il reste un point sur lequel nous pouvons encore gagner : l’automatisation. En effet, il n’est pas envisageable d’écrire et d’optimiser la représentation intermédiaire de notre langage à la main. C’est donc ici que JITPSI rentre en jeu.

Sur le modèle de nos fonctions d’évaluation, JITPSI nous permet d’écrire un générateur de fonctions d’évaluation. Le code de la Figure 4 dépeint l’usage de JITPSI couplé, comme nous l’avons fait à la main, à de la propagation d’expressions. Notre fonction `compile` prend en argument une liste d’instructions `il` et renvoie la fonction « `Builtin` » équivalente à l’évaluation récursive de ces instructions par la fonction `step` (Figure 1b).

Dans les grandes lignes, nous allons suivre les mêmes étapes que le code de la version **V2** (Figure 2c). La primitive `lambda3` (ligne 11) permet de créer une nouvelle fonction d’arité 3. Cette primitive prend en argument une fonction OCaml en charge de définir notre valeur de retour à partir de ces trois arguments. Nous commençons par récupérer la valeur des registres à l’aide de `R.find`, ce qui se traduit par un appel à la primitive `apply2` (ligne 13). Les deux premiers arguments, `R.find` et `r` sont des constantes, au sens de JITPSI, et sont récupérés depuis l’environnement OCaml à l’aide de la primitive `const`. Nous calculons ensuite les valeurs intermédiaires en déroulant la liste d’instructions `il` (ligne 14-19). La fonction `fold` incorpore ainsi les appels aux fonctions `V.add` et `V.sub` à la ligne 9. L’état mémoire est finalement mis à jour avec la nouvelle valeur des registres (`R.add`) et du drapeau `z` si nécessaire avant de terminer par l’appel final (`tailcall`) à la fonction `step` (ligne 23).

```

1  type 'a param = ('a, value) t
2  let is_zero v = V.compare v V.zero = 0
3  let j_add = const V.add and j_sub = const V.sub and j_is_zero = const is_zero
4  and j_assign = const R.add and j_lookup = const R.find
5
6  let rec fold e env = match e with
7  | Cst v -> const v
8  | Reg r -> R.find r env
9  | Op (op, x, y) -> apply2 (match op with Plus -> j_add | Minus -> j_sub) (fold x env) (fold y env)
10 let compile (il : instruction list) : V.t R.t -> bool -> instruction -> V.t =
11 return (lambda3 (fun (regs : V.t R.t param) (z : bool param) (i : instruction param) ->
12   let env = List.fold_left
13     (fun env r -> R.add r (apply2 j_lookup (const r) regs) env) R.empty [ Rax; Rdx; Rdi ] in
14   let env, zf = List.fold_left
15     (fun (env, z) -> function
16       | SetR (r, e, _) -> (R.add r (fold e env) env, z)
17       | SetZ (e, _) -> (env, Some (fold e env))
18       | IfZ _ | Ret _ | Builtin _ -> failwith "unexpected instruction kind")
19     (env, None) il in
20   let env = R.remove Tmp0 env in
21   let regs = R.fold (fun r v regs -> apply3 j_assign (const r) v regs) env regs in
22   let z = match zf with None -> z | Some v -> apply j_is_zero v in
23   apply3 (const step) regs z i))

```

Figure 4. Compilation d'un bloc d'instructions

La dernière version présentée dans la Figure 2d recourt à cette fonction `compile` pour optimiser automatiquement le morceau de code original (V0). Conformément à nos attentes, cette nouvelle version s'exécute, tout comme la précédente, en moyenne en 39 mille cycles.

Avant d'en finir avec la fonction `fibonacci`, il nous faut encore aborder un petit désagrément. Comme tout procédé de compilation à la volée, notre fonction `compile` a un coût et il ne faut pas le négliger. Ainsi, pour produire automatiquement notre version V3, il aura fallu en moyenne 1 million 770 mille cycles. Ce temps de *préparation* est bien plus important que le temps nécessaire à faire tourner la version originale V0. Aussi, rapportée à un tour de boucle, la compilation ne devient ici intéressante qu'à partir de la 5950^{ème} itération. La légèreté de JITPSI permet néanmoins de limiter cet impact négatif comparé à des approches utilisant un assembleur externe (MetaOCaml peut être 20 à 50 fois plus lent, voir Section 5).

Maintenant que nous en avons fini avec la fonction `fibonacci`, intéressons-nous un petit peu aux détails de la bibliothèque JITPSI.

3 Conception d'un générateur de code léger en OCaml

La bibliothèque JITPSI se veut simple et efficace en reposant sagement sur les couches basses du compilateur OCaml. En interne, JITPSI manipule les valeurs OCaml pour ce qu'elles sont à l'exécution, des entiers, des pointeurs et des fermetures. Le code de JITPSI s'organise en deux parties principales : l'interface utilisateur et l'émetteur de code. La Figure 5 montre comment JITPSI s'intègre dans le schéma du compilateur OCaml.

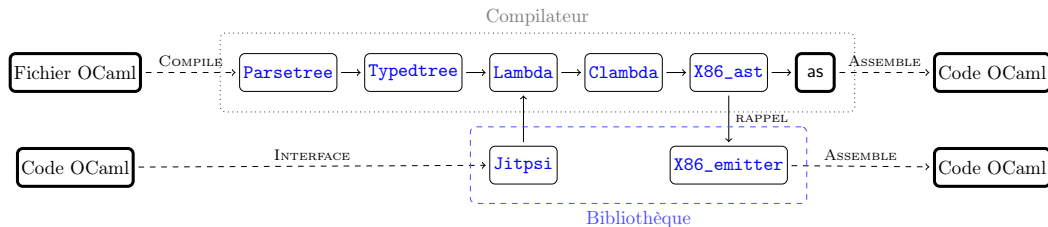


Figure 5. Organisation de JITPSI


```

type value and lambda and (+'a, 'b) t
val const : 'a -> ('a, value) t
val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t
val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t
val return : ('a, lambda) t -> 'a

```

Figure 6. Interface typée de JITPSI

L'interface. L'interface de JITPSI est donnée en Figure 6. Elle propose des primitives qui garantissent la sûreté du typage à l'aide de types fantômes. Le typage se fait ainsi à la compilation initiale du programme, économisant une partie du travail à l'exécution tout en prévenant les erreurs d'inattention.

En interne, l'interface est en charge de construire des termes `lambda` bien formés. Le type `t` représente une expression, soit d'une valeur (`value`), soit d'une fonction (`lambda`). La primitive `const` a le rôle de la fonction identité. La primitive `apply` permet d'appeler une fonction avec un argument. La création d'une nouvelle fonction passe par la primitive `lambda`. Elle prend en argument une fonction OCaml en charge de définir la valeur de retour à partir de son argument. Cette valeur de retour peut prendre la forme d'une autre expression `lambda`, ce qui permettra de construire une fonction de plus grande arité (par exemple `lambda2 body` est équivalent à `lambda (fun a -> lambda (body a))`). Finalement, une expression `lambda` pourra être assemblée en mémoire à l'aide de la primitive `return`.

Afin de faire le lien avec l'environnement OCaml, JITPSI initialise les expressions constantes à partir d'éléments fictifs (*placeholder*) et ce depuis l'extérieur de l'environnement lexical de l'expression `lambda`. Ce procédé aura pour effet de créer des entrées dans sa fermeture.

Émission de code. Comme illustré en Figure 5, le module `X86_emitter` reçoit de la part du compilateur OCaml (*callback*) un programme assembleur de type `X86_ast`. JITPSI interprète ce programme dont la structure est décomposée en trois segments :

1. *Le corps de la fonction.* Les mnémoniques sont directement traduites en opcodes sur le même modèle qu'`ocaml-jit`. Notons qu'ici, la simplicité des constructions utilisées limite naturellement la variété des instructions à quelques mnémoniques (`call`, `mov`, `lea`, `add`, `sub`, `cmp`, `jcc`, `jmp` et `ret`) ;
2. *L'allocation de la fermeture.* Les mnémoniques sont ici évaluées sommairement plutôt que traduites. JITPSI suit ainsi simplement la recette concoctée par le compilateur pour initialiser la fermeture. Durant cette phase, JITPSI simule les accès aux éléments fictifs et les remplace par les valeurs OCaml issues de l'environnement hôte ;
3. *Les métadonnées.* Les informations nécessaires au glaneur de cellules (`frametable`) sont extraites afin d'être déclarées auprès de l'environnement d'exécution d'OCaml.

Cette méthode permet de renvoyer une fermeture complète qui s'intègre parfaitement au monde OCaml et s'utilise comme le serait n'importe quelle fonction nativement compilée.

Les petits plus. Le code généré par JITPSI s'accompagne des deux avantages suivants :

- une fonction qui n'est plus utilisée peut être collectée par le glaneur de cellule et la mémoire allouée pour son code et ses métadonnées libérées ;
- de nature dynamique, JITPSI peut déterminer l'adresse réelle et le nombre d'arguments des fonctions qu'il manipule, connaissance qui offre plus de latitude au compilateur pour générer des appels directs en lieu et place des primitives de type `caml_apply`.

Limites actuelles et extensions envisagées. Sous sa forme actuelle, JITPSI souffre des limitations suivantes.

Puissance du langage. JITPSI permet pour le moment de composer des appels de fonction OCaml présentes dans l'environnement, mais ne propose pas d'autres opérations de manière

native. Ainsi, par exemple, les branchements conditionnels ne sont pas disponibles en toute généralité. On peut cependant gérer des cas simples en ajoutant une fonction de choix *if-then-else* (`val ite : bool -> 'a -> 'a -> 'a`), mais, dans ce cas, les arguments seront tous les deux évalués. L'extension du moteur pour une gestion plus générale du flot de contrôle est une des priorités d'extension ;

Autres architectures. L'émission de code est un processus qui dépend fortement de l'architecture cible, ici le x86-64. Supporter une nouvelle architecture demande de traduire de nouvelles mnémoniques en opcodes. À titre indicatif, pour x86-64, nous devons traduire 8 grandes classes de mnémoniques et la traduction prend ~600 lignes d'OCaml. Cependant, d'un point de vue pratique, la tâche serait rendue plus difficile pour une autre architecture car le compilateur OCaml n'a pas d'équivalent du module `X86_ast` pour les autres architectures et émet le code assembleur directement sous forme textuelle.

4 Application à l'exécution symbolique de BINSEC

Le moteur d'exécution de la plateforme BINSEC a sensiblement évolué depuis deux ans. Des progrès ont notamment été faits dans sa manière d'interpréter son langage intermédiaire [BHL⁺11]. La version actuelle (0.8.0) ne fait toutefois pas de Partage de Sous-expressions Communes (*local value numbering* [CT04]), que nous abrègerons en PSC. Or, avec l'aide de JITPSI, faire du PSC devient aussi facile que de faire du partage maximale d'expression (*hash consing*). En effet, JITPSI permet de réordonner toutes les opérations de lecture avant les opérations d'écriture. Ce scénario simplifie grandement la propagation d'expressions (comme illustré en Figure 4) qui n'a plus à se soucier des effets de bord des langages impératifs. Dans le but d'obtenir une base de comparaison et, bien que moins évident à mettre en place, nous avons également rajouté un PSC « classique » dans BINSEC.

Nous avons mesuré à l'aide de `Landmark` le temps de résolution du défi licorne avec trois configurations : une version classique de BINSEC, une version avec un PSC classique et une version avec JITPSI (incluant du PSC maximal). La Table 1 résume les résultats obtenus et montre le nombre et le coût des différents appels aux primitives de l'environnement symbolique. Nous pouvons y voir que la combinaison du PSC et de JITPSI permet d'accélérer de 43% l'exécution de ce programme par rapport à la version classique de BINSEC. Ce gain est en partie imputable au principe du PSC mais ne s'y limite pas. La version utilisant JITPSI est en effet 20% plus rapide que notre implémentation standard de PSC. Nous expliquons cela par le fait qu'il y a, d'une part, moins de variables temporaires (réduction du nombre d'écritures) et d'autre part, que JITPSI permet de factoriser les lectures de variables (réduction drastique du nombre de lectures), ce qui n'est pas possible par voie classique.

Table 1. Comparaison de différentes versions de BINSEC sur licorne

	Occurrences			# Cycles d'horloge		
	Classique	PSC	JITPSI	Classique	PSC	JITPSI
Écriture d'une variable	2 074 M	1 006 M	812 M	427 G	264 G	164 G
Lecture d'une variable	2 964 M	1 825 M	457 M	404 G	294 G	63 G
Écriture en mémoire	139 M	139 M	139 M	352 G	373 G	348 G
Lecture en mémoire	208 M	207 M	207 M	170 G	184 G	164 G
Résolution complète				10m24s	8m42s	7m15s

PSC : partage des sous-expr. communes – la version JITPSI intègre le PSC et le JIT

5 Positionnement et discussion

JITPSI répond à un besoin précis. Il apporte une solution simple et efficace, mais qui, par la sobriété de son langage, limite à l'heure actuelle le champ des possibles. Nous envisageons

toutefois d'enrichir ce langage avec les expressions *if-then-else* et quelques opérations sur les types de bases (`bool`, `int`, etc.).

À notre connaissance, *JITPSI* est la seule approche à proposer une interface programmatique permettant de manipuler les valeurs du programme en toute autonomie à l'exécution.
















D'autres travaux connexes traitent de la thématique de la compilation à la volée en OCaml. Le projet OCamlJIT2 [Meu11] vise à accélérer l'exécution d'OCaml en version *bytecode*. Il ne semble toutefois plus maintenu depuis plus d'une dizaine d'années.

MetaOCaml [Kis18] est une grande source d'inspiration pour l'interface de JITPSI. En théorie, MetaOCaml permet de faire tout ce pour quoi JITPSI a été conçu, et bien plus encore. En revanche, dans la pratique, nous avons eu le plus grand mal à expérimenter avec. Les problèmes rencontrés sont des erreurs qui ont lieu à l'exécution à cause d'une interface introuvable ou d'une valeur difficile à sérialiser (*cross-stage persistence*). Notons que MetaOCaml est naturellement multi-architecture puisqu'il génère du code OCaml. Pour ce qui est du temps d'émission de code, nous avons mesuré combien de cycles d'horloge étaient nécessaires à MetaOCaml et à JITPSI pour compiler la fonction `spower` présentée dans son exemple d'introduction. Il se trouve que JITPSI est bien plus rapide, avec une accélération allant de 20 (grandes valeurs de `n`) à 50 fois pour `n = 2`.

Le projet `ocaml-jit` est la source principale d'inspiration de l'émetteur de code de JITPSI. Sa cible, l'interpréteur interactif d'OCaml, est cependant assez éloigné de nos besoins.

La Table 2 résume les différences entre ces approches sur quelques aspects clés.

Table 2. Comparaison des approches

	 MetaOCaml	 ocaml-jit	 JITPSI
Autonomie à l'exécution			
Temps d'émission de code			
Multi-architecture			
 : relativement lent –  : très rapide –  : difficilement extensible			

Applications potentielles. Le cadre d'usage classique de JITPSI est l'optimisation d'interpréteurs au sens large, et notamment la spécialisation de l'interprétation pour des séquences d'instructions données (et souvent ré-appelées). Des exemples possibles d'application incluent l'exécution symbolique, les simulateurs, émulateurs et autres machines virtuelles pour des gains en performance (vitesse), ou encore les interpréteurs abstraits, pour lesquels définir des transformateurs abstraits spécialisés pour les blocs du programme à analyser pourrait permettre des gains notables de précision. Nous sommes preneurs de retour pour tout autre domaine d'application intéressant.

6 Conclusion

Nous avons présenté dans cet article JITPSI, une bibliothèque de compilation à la volée pour OCaml. JITPSI expose une interface programmatique simple permettant la composition de plusieurs appels de fonctions dans le but de produire un code natif aussi performant que s'il avait été produit par compilation depuis des sources OCaml. Un domaine typique d'utilisation visé est l'optimisation d'interpréteurs (au sens large) via encodage dédié par blocs d'instructions plutôt que par instructions. Nous utilisons par exemple JITPSI pour optimiser l'exécuteur symbolique BINSEC sur de longs chemins d'exécution, permettant d'accélérer la résolution du défi de rétro-ingénierie licorne (tiré du *France CyberSecurity Challenge 2022* de l'ANSSI) d'environ 40%. Nous espérons ainsi raviver l'intérêt de la communauté OCaml pour la compilation à la volée et trouver de nouveaux cas d'usages et pistes d'améliorations.

Références

- [Bel73] James R. BELL : Threaded code. *Commun. ACM*, 1973.
- [BHL⁺11] Sébastien BARDIN, Philippe HERRMANN, Jérôme LEROUX, Olivier LY, Renaud TABARY et Aymeric VINCENT : The BINCOA framework for binary code analysis. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011.
- [CS13] Cristian CADAR et Koushik SEN : Symbolic execution for software testing : Three decades later. *Commun. ACM*, 2013.
- [CT04] Keith D. COOPER et Linda TORCZON : *Engineering a Compiler*. Morgan Kaufmann, 2004.
- [DB15] Adel DJOUDI et Sébastien BARDIN : BINSEC : binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS, 2015*.
- [DL93] Damien DOLIGEZ et Xavier LEROY : A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.
- [FP11] Martin FOWLER et Rebecca PARSONS : *Domain-specific languages*. Addison-Wesley, 2011.
- [Fut99] Yoshihiko FUTAMURA : Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 1999.
- [Kis18] Oleg KISELYOV : Reconciling abstraction with high performance : A metaocaml approach. *Foundations and Trends in Programming Languages*, 2018.
- [LA04] Chris LATTNER et Vikram ADVE : Llvm : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization*, 2004.
- [Meu11] Benedikt MEURER : Ocamljit 2.0 - faster objective caml, 2011.
- [MZH20] Barton P. MILLER, Mengxiao ZHANG et Elisa R. HEYMANN : The relevance of classic fuzz testing : Have we solved this one? *IEEE Transactions on Software Engineering*, 2020.
- [ND15] Anh Q. NGUYEN et Hoang V. DANG : Unicorn : Next generation cpu emulator framework. BlackHat USA, 2015. <https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>.
- [PF20] Sebastian POEPLAU et Aurélien FRANCILLON : Symbolic execution with symcc : Don't interpret, compile! In *29th USENIX Security Symposium, USENIX Security*, 2020.
- [PF21] Sebastian POEPLAU et Aurélien FRANCILLON : Symqemu : Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.
- [RLV⁺96] Theodore H. ROMER, Dennis LEE, Geoffrey M. VOELKER, Alec WOLMAN, Wayne A. WONG, Jean-Loup BAER, Brian N. BERSHAD et Henry M. LEVY : The structure and performance of interpreters. *SIGOPS Oper. Syst. Rev.*, 1996.
- [SGS⁺16] Nick STEPHENS, John GROSEN, Christopher SALLS, Andrew DUTCHER, Ruoyu WANG, Jacopo CORBETTA, Yan SHOSHITAISHVILI, Christopher KRÜGEL et Giovanni VIGNA : Driller : Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.
- [SYZZ⁺14] Gulfem SAVRUN YENICERI, Wei ZHANG, Huahan ZHANG, Eric SECKLER, Chen LI, Stefan BRUNTHALER, Per LARSEN et Michael FRANZ : Efficient hosted interpreters on the jvm. *ACM Trans. Archit. Code Optim.*, 2014.

- [Ver22] Mathéo VERGNOLLE : FCSC 2022 : Licorne. France Cyber-Security Challenge, 2022. https://github.com/binsec/binsec/blob/34e2897f2d813ae203b01a7273edafd747547813/doc/sse/fcsc_licorne.md.
- [WJG⁺23] Guannan WEI, Songlin JIA, Ruiqi GAO, Haotian DENG, Shangyin TAN et Oliver BRAČEVAC : Compiling parallel symbolic execution with continuations. In *45th International Conference on Software Engineering, ICSE*, 2023.
- [YLX⁺18] Insu YUN, Sangho LEE, Meng XU, Yeongjin JANG et Taesoo KIM : Qsym : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.