Heuristiques pour l'optimisation de l'exécution symbolique Wasm

Léo Andrès
OCamlPro
leo@ocamlpro.com

Ce stage offre à un·e étudiant·e curieux·se et motivé·e l'opportunité de participer à un projet de recherche combinant **programmation avancée** (en OCaml) et enjeux concrets d'analyse logicielle appliqués à des programmes WebAssembly, C et Rust.

Contexte

Le contexte de ce stage est celui de l'analyse statique de programmes WebAssembly (Wasm) [6] et plus précisément de l'outil **Owi** [1] développé chez OCamlPro en collaboration avec l'INESC-ID/Instituto Superior Técnico de l'Université de Lisbonne. Owi permet d'effectuer de l'**exécution symbolique** [8] de programmes Wasm. C'est-à-dire que l'utilisateur peut fournir un programme et obtenir automatiquement les valeurs d'entrées pour lesquelles le programme lève une erreur. Il est également possible d'analyser des programmes écrits dans d'autres langages tels que Rust ou C en les compilant vers Wasm, ce que l'outil est capable de faire directement. L'exécution symbolique dans Owi est particulière en ce qu'elle permet d'analyser des programmes inter-langages[1], qu'elle est effectuée en parallèle[2] et qu'aucune approximation n'y est faite.[3] Une description détaillée d'Owi est disponible dans plusieurs publications [2, 3, 7].

Sujet

Ce projet de recherche s'inscrit dans le prolongement des travaux menés sur Owi, son but est d'**améliorer l'ordre d'exploration des différents chemins d'exécution du programme**. Pour l'instant, les chemins rencontrés sont ajoutés à une file et traités d'une manière « FIFO ». Or, il a été montré dans plusieurs autres outils d'exécution symbolique que des ordres d'exploration plus fins peuvent mener à de bien meilleurs résultats [4, 5, 9]. Ces ordres d'exploration sont généralement basés sur des **heuristiques** qui permettent de mieux gérer le problème de **path explosion**, une difficulté courante en exécution symbolique où le nombre de chemins possibles croît de manière exponentielle avec la taille du programme.

Les objectifs du stage sont les suivants :

- remplacement de la file actuelle par une **file de priorité** ;
- développement d'**heuristiques permettant d'attribuer des scores** aux différents points de branchements[4] lors d'une passe d'analyse préliminaire ;
- ajout d'un **paramétrage en ligne de commande** afin de sélectionner les heuristiques à appliquer ;
- **analyse de l'impact** des différentes heuristiques grâce aux benchmarks présents dans Owi.

Si le temps le permet, il pourra être envisagé d'implémenter d'autres stratégies d'exploration classiques (DFS, BFS, NURS) ou d'implémenter une variante de l'algorithme A* comme décrit par de Castro Pinto [5].

[1] Par exemple, un programme fait à la fois de code Rust et de code C.
[2] Grâce à OCaml 5.
[3] Si l'analyse termine, on a la garantie d'une absence de bogue.
[4] Par exemple, le nombre d'assertions contenues dans chaque branche.

📚 Pré-requis

- niveau M1/M2 informatique ou équivalent
- bonne maîtrise d'un langage fonctionnel (idéalement OCaml)
- notions d'algorithmique des graphes et de compilation

🏆 Compétences développées

- découverte approfondie de WebAssembly
- programmation OCaml avancée
- techniques d'exécution symbolique

📍 Lieu du stage

L'étudiant·e effectuera les travaux dans les locaux d'OCamlPro, à l'adresse suivante :

```
OCamlPro SAS
21 rue de Châtillon
75014 Paris
```

📅 Dates

Le stage pourra avoir lieu entre mars et septembre.

👩‍🏫 Encadrants

- Léo Andrès (Ph. D.), ingénieur R&D.

Le télétravail est possible, généralement deux jours par semaine.

📜 Bibliographie

[1] Léo Andrès, Pierre Chambart, Filipe Marques, Eric Patrizio, Arthur Carcano, et Zhicheng Hui. 2021. Owi. Consulté à l'adresse https://github.com/ocamlpro/owi

[2] Léo Andrès, Filipe Marques, Arthur Carcano, Pierre Chambart, José Fragoso Femenin dos Santos, et Jean-Christophe Filliâtre. 2024. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming* 9, 2 (2024). https://doi.org/10.48550/arXiv.2412.06391

[3] Léo Andrès. 2024. Exécution symbolique pour tous ou Compilation d'OCaml vers WebAssembly.

[4] Cristian Cadar, Daniel Dunbar, et Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008. https://doi.org/10.5555/1855741.1855756

[5] Theo de Castro Pinto, Antoine Rollet, Grégoire Sutre, et Ireneusz Tobor. 2023. Guiding Symbolic Execution with~A-Star. In *Lecture Notes in Computer Science (Lecture Notes in Computer Science)*, novembre 2023. Springer, Eindhoven, Netherlands, 47-65. https://doi.org/10.1007/978-3-031-47115-5_4

[6] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, et JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017. 185-200. https://doi.org/10.1145/3062341.3062363

[7] Zhicheng Hui et Léo Andrès. 2025. Cross-Language Symbolic Runtime Annotation Checking. In *36es Journées Francophones des Langages Applicatifs (JFLA 2025)*, janvier 2025. Roiffé, France. Consulté à l'adresse https://inria.hal.science/hal-04798756

[8] James C. King. 1976. Symbolic Execution and Program Testing. *Communications of the ACM* (1976). https://doi.org/10.1145/360248.360252

[9] Wei Wang. 2024. Symbolic execution search strategy based on Katz centrality analysis. In *Fifth International Conference on Computer Communication and Network Security (CCNS 2024)*, 2024. 236-241. https://doi.org/10.1117/12.3038376