Université du Québec à Montréal

Projet de session: Rétro-débogueur pour le langage d'assemblage PEP/8

Travail présenté à
M. Étienne Gagnon

Dans le cadre du cours
INF7741 – Machines Virtuelles

Frédéric Vachon
VACF30098405
Philippe Pépos Petitclerc
PEPP03049109

21 avril 2016

Table des matières

1 Introduction

Une grande partie du travail d'un développeur est d'effectuer le débogage des programmes. La plupart du temps, le développeur doit relancer plusieurs fois l'application avant de trouver la cause du bogue. Si le développeur avait accès à un débogueur qui lui permettait d'effectuer une exécution inverse, sa productivité globale augmenterait. Ce constat est également vrai dans un contexte de sécurité informatique où un développeur cherche à profiter d'une faille de sécurité dans un logiciel pour suivre le déploiement de son code injecté dans la mémoire du programme. C'est pour répondre à ces deux besoins que nous avons décidé d'élaborer un projet de débogage à reculons. L'aspect sécurité informatique nécessite que nous utilisions un langage très bas niveau. Puisque le langage assembleur *PEP/8* est enseigné à l'Université du Québec à Montréal et que ce langage ne possède qu'un ensemble d'instruction très simple, il était un candidat idéal pour développer notre projet dans le cadre d'un cours de maîtrise.

Le document suivant étaye le travail effectué au cours de la session sur l'élaboration d'un débogueur à reculons pour le langage d'assemblage *PEP/8*. Il présente les objectifs du projet et les éléments nécessaires à sa réalisation. Nous faisons ensuite une revue de littérature et les articles qui nous ont permis de dresser le paysage actuel des techniques de débogage à reculons. La section suivante explique ensuite l'approche choisie ainsi que les détails d'implémentation de notre outil. Nous présentons ensuite *PEPDB*, le résultat de notre projet de session accompagné d'une section sur les possibles améliorations au projet.

2 Objectifs

L'objectif du projet est de développer un débogueur du langage assembleur PEP/8. Le débogueur aura la capacité d'exécuter le programme débogué à sens inverse, ce qui permettra au programmeur de retracer la provenance des bogues a posteriori. Le débogueur supportera le débogage de code mutant.

Puisque nous ne partons pas d'un projet existant, nous allons devoir commencer par implémenter un assembleur et un désassembleur. De la sorte, un utilisateur pourra fournir un fichier source ou un fichier binaire à l'interpréteur.

Pour réaliser le débogueur, il est impératif de construire un environnement d'exécution de code machine PEP/8 qui offre la flexibilité nécessaire pour y ajouter les fonctionnalités de débogage mentionnées ci-haut. L'interpréteur va effectuer une capture d'informations, lesquelles seront utilisées par le débogueur afin d'effectuer l'exécution inverse du programme.

Le débogueur fournira les fonctionnalités normales d'un débogueur soit les points d'arrêts (breakpoints) et l'exécution pas-à-pas. Il permettra d'utiliser ces mêmes fonctionnalités en mode d'exécution inverse. Pour implémenter cette dernière fonctionnalité, nous allons utiliser une technique de capture d'états se basant sur le maintien des informations permettant la restitution de l'état précédent l'exécution de chaque instruction.

Cette implémentation supportera le code automodifiant.

3 Revue de littérature et état de l'art

En effectuant notre revue de littérature sur le sujet, nous avons d'abord remarqué qu'il s'agit d'un sujet qui intéresse les chercheurs en informatiques depuis des décennies. Nous présentons les trois articles qui présentent les trois techniques les plus utilisées afin d'effectuer du débogage à reculons.

Dans son article Efficient Algorithms for Bidirectional Debugging (2000)[2], Boothe présente une approche qui part du constat qu'un appel au débogueur via un trap coûte environ un million de cycles processeur. Il cherche donc à éviter le plus possible d'effectuer des interruptions. Au lieu d'injecter des interruptions dans le programme invité, Boothe ajoute des appels à des fonctions qui vont s'occuper de mettre à jour des compteurs qui vont permettre d'arrêter l'exécution du programme exactement là où l'utilisateur le désire. Cette technique est particulièrement efficace lorsqu'un utilisateur voudra arrêter le programme après un certain nombre d'itérations dans une boucle ou bien à une certaine profondeur de récursion, car les débogueurs font ce travail en effectuant des appels au débogueur pour chaque itération dans la boucle. Cette importante économie en cycle de processeur permet d'effectuer le débogage à reculons en réexécutant le programme, parfois même en plus d'une passe, pour arrêter l'exécution là où l'utilisateur l'a demandé. Afin de s'assurer du déterminisme de l'exécution, des captures doivent être effectuées dans le cas de certains appels systèmes. L'approche de Boothe se base donc sur la réexécution du programme afin de revenir à un état précédent de l'exécution du programme.

Akgul propose une autre approche dans son article Instruction-level Execution for Debugging (2002)[1] qu'il améliore par la suite par dynamic slicing dans son article suivant A Fast Assembly Level Reverse Execution Method via Dynamic Slicing (2004). Au lieu d'exécuter le programme de nouveau, il construit le programme inverse instruction par instruction. De cette façon, il ne faut pas garder un historique des états d'exécution du programme. Il faut uniquement pouvoir passer du programme d'origine au programme inverse. Bien entendu, il doit y avoir capture d'états dans le cas où des appels non déterministes sont effectués, mais c'est le cas de toutes les approches que nous avons étudié.

Le troisième article est An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach (2005) par Koju et al[3]. La technique qu'ils emploient se base sur deux modes d'exécution afin de ne pas occasionner de surcoût en performance pour un utilisateur qui veut exécuter le programme sans effectuer de débogage. Celui-ci peut changer vers un mode débogage lorsqu'il le souhaite. Ce qui nous intéresse particulièrement dans cet article est la capture d'états effectués à intervalle de temps dynamiquement ajusté selon une mesure de la durée requise pour la capture d'état et le surcoût toléré. Pour effectuer l'exécution inverse, le débogueur retourne au dernier état capturé et réexécute le code jusqu'au point souhaité.

4 Implémentation

4.1 Suite d'outils préliminaires

Nous avons implémenté les différents outils nécessaires à l'implémentation de notre débogueur en Nit. Nous avons développé chaque outil de façon isolée afin qu'il soit possible d'utiliser chacun indépendamment. Notre suite d'outils comprend donc un assembleur et un désassembleur de *PEP/8* et un interpréteur naïf. Nous avons décidé d'implémenter l'interpréteur naïvement puisque nous voulions être en mesure de déboguer le comportement exact du code assembleur écrit et pas une version optimisée. Aussi, une implémentation simple de l'interpréteur facilite beaucoup l'intégration des mécanismes d'historique pour le débogage à reculons.

4.2 Débogueur

Nous avons opté d'implémenter notre débogueur avec une approche de machine virtuelle. C'est à dire, d'introduire les mécanismes qui permettent le débogage du programme à l'intérieur même de son environnement d'exécution. Ceci nous permet de plus facilement intercepter les implémentations des différentes instructions assembleurs afin d'exécuter les routines de vérifications du débogueur. Autrement, nous aurions dût attacher notre débogueur au processus en cours d'exécution et d'y injecter des instructions binaires pour arriver au même résultat.

Les mécanismes que nous avons implémentés dans le débogueur sont les suivants : la gestion de points d'arrêt et la reprise d'exécution, l'exécution pas-à-pas, l'inspection de registres et de l'état de la mémoire ainsi que le désassemblage dynamique des octets en mémoire. Fonctionnalité d'exécution à reculons Nous avons choisi d'implémenter le débogage à reculons en sauvegardant l'effet de chaque instruction exécuté dans un historique d'exécution. En assembleur *PEP/8*, chaque instruction peut modifier au maximum un mot de deux octets en mémoire. Autrement, l'effet de l'instruction peut être de modifier le contenu d'un ou plusieurs registres. Nous avons donc implémenté un interpréteur spécialisé qui effectue le travail additionnel pour permettre l'exécution à reculons et qui supporte les points d'arrêts. Pour chaque instruction, nous sauvegardons donc le banc de registre, l'adresse mémoire affectée s'il y a lieu et la valeur qui s'y trouve. Le surcoût en mémoire de notre approche est donc tout à fait acceptable compte tenu des bénéfices de l'exécution à reculons.

Certaines instructions ont un effet non déterministe. En langage d'assemblage *PEP/8*, ces instructions sont les instructions de lecture de données usagers : `CHARI` et `DECI`. Afin de maintenir l'application dans le même chemin d'exécution, si après avoir enregistré une instruction non déterministe, l'usager demande de la rejouer en sens normal, nous rappliquons la lecture effectuée lors de la dernière exécution de l'instruction. Cette logique nous permet de déboguer le chemin d'exécution sans avoir à répéter chaque entrée à chaque

passage.

4.3 Séparation des tâches

Lors des phases d'implémentation, le travail à accomplir fût divisé en deux afin d'implémenter plus rapidement les différents composants du projet. Initialement, Frédéric Vachon implémenta le désassembleur et Philippe Pépos Petitclerc l'assembleur de code assembleur *PEP/8*. Ensuite, le travail sur l'interpréteur fût divisé en instructions. Frédéric Vachon a ensuite codé la majeure partie du débogueur initial. Auquel les deux membres du projet ont ultérieurement contribué lorsque nécessaires. La logique d'enregistrement d'état a d'abord été écrite par Philippe Pépos Petitclerc et retravaillée par Frédéric Vachon pour faciliter la ré-exécution d'instructions déjà enregistrées et gérer le cas des entrées usagers.

5 Présentation de l'outil

Nous présentons donc *PEPDB* un débogueur pour le langage d'assemblage *PEP/8* qui est basé sur une machine virtuelle sauvegardant les effets des instructions exécutés afin de permettre l'exécution inverse des instructions. L'outil consiste en une série d'outils écris en *Nit* qui permettent individuellement d'assembler, de désassembler, d'interpréter de de déboguer l'exécution d'un programme *PEP/8*.

Le débogueur (outil principal de la suite) supporte la correspondance entre le code source et le code octet compilé. Il est donc en mesure d'afficher les commentaires et les étiquettes même s'ils disparaissent à l'assemblage.

L'approche utilisée pour l'implémentation du débogueur et de la gestion de l'historique d'exécution permet facilement de supporter le code auto modifiant. Par contre, s'il y a réécriture du code original, le débogueur doit invalider la correspondance avec les fichiers sources.

Une utilisation typique du débogueur consiste en 3 étapes : Le chargement du code *PEP/8*, l'exécution jusqu'à l'occurrence du bogue recherché, l'exécution inverse jusqu'à l'instruction qui introduit le comportement imprévu. Le débogueur charge automatiquement le fichier source passé en argument en mémoire. Il est donc immédiatement possible de le désassembler pour voir le source via la commande `disass`. La gestion des points d'arrêts se fait via les commandes `break` ou `remove`. Le contrôle de l'exécution est possible via les commandes `run`, `continue` et `nexti`. Une fois que le bogue est survenu, l'utilisateur voudra reculer l'exécution via les versions `rev-` des mêmes commandes.

Le code des différents outils est trouvable dans le répertoire `src` du projet. Le code de l'assembleur et du désassembleur se trouve respectivement dans les fichiers `asm.nit` et `disasm.nit`. L'interpréteur et la version spécialisée pour le débogage se trouvent dans `interpreter.nit` et le débogueur lui-même dans `debugger.nit`. Le répertoire `src` contient également un fichier de description des instructions *PEP/8* et un

fichier de description des commandes du débogueur. Le fichier `Makefile` à la racine du projet compilera les différents outils et les placera dans le répertoire `bin` du projet.

6 Améliorations possibles

La première fonctionnalité additionnelle qui serait intéressante est la possibilité d'avoir un arbre d'exécution. Il serait donc possible de diverger de l'exécution originale en exécutant une nouvelle version d'une instruction d'entrée usager. Il serait possible de permettre à l'usager de choisir la branche d'exécution lorsque le débogueur arrive a un branchement et même de proposer d'en créer une nouvelle.

Une autre amélioration éventuelle serait de construire une interface graphique qui n'est pas en ligne de commande. Comme le langage d'assemblage *PEP/8* est surtout utilisé à des fins d'apprentissage et, à l'UQÀM, par des étudiants en début de programme, il serait plus simple pour eux d'avoir accès à une interface graphique.

7 Conclusion

Suite à une recherche sur les solutions en débogage à reculons, nous avons choisi que la solution la plus adaptée à nos besoins fût d'utiliser une machine virtuelle qui permettrait d'enregistrer les effets de l'exécution d'une instruction afin de permettre de l'exécuter dans n'importe quel sens par la suite. Nous avons donc implémenté une suite d'outils permettant de travailler avec le langage *PEP/8* et avons implémenté un débogueur offrant ces fonctionnalités. Le débogueur se sert d'un interpréteur spécialisé qui enregistre la différence d'état de la mémoire ainsi que les registres. L'outil proposé est également en mesure de faire la correspondance entre le code source original et le code octet assemblé. Il supporte aussi le débogage de code automodifiant.

Références

[1] Tankut Akgul and Vincent J. Mooney, III. Instruction-level reverse execution for debugging. *SIGSOFT Softw. Eng. Notes*, 28(1) :18–25, November 2002.

[2] Bob Boothe. Efficient algorithms for bidirectional debugging. *SIGPLAN Not.*, 35(5) :299–310, May 2000.

[3] Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 79–88, New York, NY, USA, 2005. ACM.