Space Invading Systems Code

Cristiano Calcagno[2], Dino Distefano[1], Peter O'Hearn[1], and Hongseok Yang[1]

[1] Queen Mary University of London
[2] Imperial College

1 Introduction

Space Invader is a static analysis tool that aims to perform accurate, automatic verification of the way that programs use pointers. It uses separation logic assertions [10,11] to describe states, and works by performing a proof search, using abstract interpretation to enable convergence. As well as having roots in separation logic, Invader draws on the fundamental work of Sagiv et. al. on shape analysis [12]. It is complementary to other tools – e.g., SLAM [1], Blast [8], ASTRÉE [6] – that use abstract interpretation for verification, but that use coarse or limited models of the heap.

Space Invader began life as a theoretical prototype working on a toy language [7], which was itself an outgrowth of a previous toy-language tool [3]. Then, in May of 2006, spurred by discussions with Byron Cook, we decided to move beyond our toy languages and challenge programs, and test our ideas against real-world systems code, starting with a Windows device driver, and then moving on to various open-source programs. (Some of our work has been done jointly with Josh Berdine and Cook at Microsoft Research Cambridge, and a related analysis tool, SLAyer, is in development there.)

As of the summer of 2008, Space Invader has proven pointer safety (no null or dangling pointer dereferences, or leaks) in several entire industrial programs of up to 10K LOC, and more partial properties of larger codes. There have been three key innovations driven by the problems encountered with real-world code.

- *Adaptive analysis.* Device drivers use complex variations on linked lists – for example, multiple circular lists sharing a common header, several of which have nested sublists – and these variations are different in different drivers. In the adaptive analysis predicates are discovered by scrutinizing the linking structure of the heap, and then fed to a higher-order predicate that describes linked lists. This allows for the description of complex, nested (though linear) data structures, as well as for adapting to the varied data structures found in different programs [2].
- *Near-perfect Join.* The adaptive analysis allowed several driver routines to be verified, but it timed out on others. The limit was around 1K LOC, when given a nested data structure and a procedure with non-trivial control flow (several loops and conditionals). The problem was that there were thousands of nodes at some program points in the analysis, representing huge disjunctions. In response, we discovered a partial join operator which lost enough

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 1–3, 2009.

information to, in many cases (though crucially, not always), leave us with only one heap. The join operator is *partial* because, although it is often defined, a join which *always* collapses two nodes into one will be too imprecise to verify the drivers: it will have false alarms. Our goal was to *prove* pointer safety of the drivers, so to discharge even 99.9% of the heap dereference sites was considered a failure: *not* to have found a proof.

The mere idea of a join is of course standard: The real contribution is existence of a partial join operator that leads to speed-ups which allow entire drivers to be analyzed, while retaining enough precision for the goal of proving pointer safety with zero false alarms [9].

– *Compositionality.* The version of Space Invader with adaptation and join was a top-down, whole-program analysis (like all previous heap verification methods). This meant the user had to either supply preconditions manually, or provide a "fake main program" (i.e., supply an environment). Practically, the consequence was that it was time-consuming to even get started to apply the analysis to a new piece of code, or to large codes. We discovered a method of inferring a preconditon and postcondition for a procedure, without knowing its calling context: the method aims to find the "footprint" of the code [4], a description of the cells it accesses. The technique – which involves the use of abductive inference to infer assertions describing missing portions of heap – leads to a compositional analysis which has been applied to larger programs, such as a complete linux distribution of 2.5M LOC [5].

The compositional and adaptive verification techniques fit together particularly well. If you want to *automatically* find a spec of the data structure usage in a procedure in some program you don't know, without having the calling context of the procedure, you really need an analysis method that will find heap predicates for you, without requiring you (the human) to supply those predicates on a case-by-case basis. Of course, the adaptive analysis selects its predicates from some pre-determined stock, and is ultimately limited by that, but the adaptive capability is handy to have, nonetheless.

We emphasize that the results of the compositional version of Space Invader (code name: Abductor) are partial: it is able to prove some procedures, but it might fail to prove others; in linux it finds Hoare triples for around 60,000 procedures, while leaving unproven some 40,000 others[1]. This, though, is one of the benefits of compositional methods. It is possible to get accurate results on parts of a large codebase, without waiting for the "magical abstract domain" that can automatically prove all of the procedures in all of the code we would want to consider.

[1] Warning: there are caveats concerning Abductor's limitations, such as how it ignores concurrency. These are detailed in [5].

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys. (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis of composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Automatic modular assertion checking with separation logic. In: 4th FMCO, pp. 115–137 (2006)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Footprint analysis: A shape analysis that discovers preconditions. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 402–418. Springer, Heidelberg (2007)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
7. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL: Principles of Programming Languages (2002)
9. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
10. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
12. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM TOPLAS 20(1), 1–50 (1998)