Infer: An Automatic Program Verifier for Memory Safety of C Programs

Cristiano Calcagno and Dino Distefano

Monoidics Ltd, UK

Abstract. Infer[1] is a new automatic program verification tool aimed at proving memory safety of C programs. It attempts to build a compositional proof of the program at hand by composing proofs of its constituent modules (functions/procedures). Bugs are extracted from failures of proof attempts. We describe the main features of Infer and some of the main ideas behind it.

1 Introduction

Proving memory safety has been traditionally a core challenge in program verification and static analysis due to the high complexity of reasoning about pointer manipulations and the heap. Recent years have seen an increasing interest of the scientific community for developing reasoning and analysis techniques for the heap. One of the several advances is separation logic, a formalism for reasoning about mutable data structures and pointers [14], and based on that, techniques for automatic verification. Infer is a commercial program analyzer aimed at the verification of memory safety of C code. Infer combines many recent advances in automatic verification with separation logic. Some of the features are:

- It performs *deep-heap analysis* (a.k.a. shape analysis) in presence of dynamic memory allocation. Infer's abstract domain can precisely reason about complex dynamic allocated data structures such as singly/doubly linked lists, being them circular or non-circular, nested with other lists, etc.
- It is *sound* w.r.t. the underlying model of separation logic. Infer synthesizes sound Hoare triples which imply memory safety w.r.t. that model.
- It is *scalable*. Infer implements a compositional inter-procedural analysis and has been applied to several large software projects containing up to several millions of lines of code (e.g. the Linux kernel).
- It is completely *automatic*: the user is not required to add any annotations or modify the original source code. Moreover, for large software projects, Infer exploits the information in the project's build to perform the analysis.
- It can analyze *incomplete code*. Infer can be applied to a piece of code in isolation, independently from the context where the code will be used.

[1] Special thanks to Dean Armitage, Tim Lownie, and John Lownie from Monoidics USA, Richard Retting and Bill Marjerison from Monoidics Japan, and Hongseok Yang, for their invaluable contributions to Infer.

M. Bobaru et al. (Eds.): NFM 2011, LNCS 6617, pp. 459–465, 2011.

Being an automatic program verifier, when run, Infer attempts to build a proof of memory safety of the program. The outcomes of the attempt can be several:

- Hoare triples of some procedures are found. Then, because of soundness, one can conclude that those procedures will not make erroneous use of memory in any of their executions. If a triple for the top level procedure (e.g., `main`) is found, one can conclude the memory safety of the entire program.
- The proof attempt fails for some procedures. Infer extracts from this failed attempt the possible reasons which prevented it to establish memory safety. These findings are then returned to the user in the form of a bug report.
- For some procedures the proof attempt fails due to internal limitations of Infer (e.g., expressivity of the abstract domain, or excessive over-approximation). In this case nothing can be concluded for those procedures.

Infer's theoretical foundations are mainly based on [5], but also include techniques from [8,2,4,11]. This paper focusses on the tool mainly from a user perspective. We refer the interested reader to the above articles for the underlying theory.

2 Procedure-Local Bugs

In this section we illustrate Infer's concept of *procedure-local* bugs by example, focusing on memory leaks in the context of incomplete code (e.g. without `main`) and inter-procedural reasoning. The usual notion that all memory allocated must be eventually freed does not apply in the context of incomplete code. The question then arises of what is a memory leak in this context and how to assign blame. We will apply the following general principle: when a new object is allocated during the execution of a procedure, it is the procedure's responsibility to either deallocate the object or make it available to its callers; there is no such obligation for objects received from the caller.

Consider the function `alloc0()` in Figure 1. It allocates an integer cell and stores its address into `i` when the flag `b` is true, or sets `i` to zero when `b` is false. This function by itself does not cause memory leaks, because it returns the newly allocated cell to the caller using the reference parameter `i`. It is the caller's responsibility to make good use of the returned cell. `example1()` shows a first example of procedure-local bug. The first call to `alloc0()` sets the local variable `i` to zero. However, after the second call, `i` will point to a newly allocated integer cell. This cell is then leaked since it is not freed before `example1()` completes. Infer blames `example1()` for leaking the cell pointed to by `i`. The bug is fixed in `example2()` where `i` is returned to the caller. It becomes the caller's responsibility to manage the cell, perhaps by freeing it, perhaps by making it available to its own caller. This passing of responsibility carries on, up the call chain, as long as source code is available. In the extreme case, when the whole program with a `main` function is available, we recover the usual (global) notion of memory leak. Even in that case, it is important to blame the appropriate procedure. A more subtle leak is present in `example3()`, which is a slight modification of `example2()`. As in the previous case, after the second call to `alloc0()`, `i` points

```
void alloc0(int **i, int b) {
  if (b) *i = malloc(sizeof (int));
  else *i = 0;
}

void example1() {
  int *i;
  alloc0(&i, 0); //ok
  alloc0(&i, 1); // memory leak
}

int *example2() {
  int *i;
  alloc0(&i, 0); // ok
  alloc0(&i, 1); // ok, malloc
  return i; // no memory leak }
```

```
int *example3() {
  int *i;
  alloc0(&i, 0); // ok
  alloc0(&i, 1); // ok, malloc
  alloc0(&i, 1); // leak: i overwritten
  return i; }

int *global;

int *example4() {
  int *i;
  alloc0(&i, 0); // ok
  alloc0(&i, 1); // ok, malloc
  global = i;
  alloc0(&i, 1); // ok, i in global
  return i; }
```

Fig. 1. Examples of procedure-local bugs

to a newly allocated cell. However, the third call to `alloc0()` creates a second cell and makes the first one unreachable and therefore leaked. The problem is fixed in `example4()` where the first cell is stored in a global variable, and the second one passed to the caller. Hence, `example4()` does not leak any memory.

Specifications. Infer automatically discovers specs for the functions that can be proven to be memory safe (in this case, those which do not leak memory). The spec discovered for `example2()` is[2] $\{\texttt{emp}\}$ `example2()` $\{\texttt{ret}\mapsto -\}$ meaning that the return value points to an allocated cell which did not exist in the (empty) precondition. The spec discovered for `example4()` is

$$\{\texttt{\&global}\mapsto -\}\ \texttt{example4()}\ \{\exists x.\ \texttt{\&global}\mapsto x\ *\ x\mapsto -\ *\ \texttt{ret}\mapsto -\}$$

meaning that variable `global` in the postcondition contains the address `x` of some memory cell, and the return value points to a separate cell. Notice that a call to `example4()` could produce a leak if `global` were overwritten. In that case, since the leaked cell exists before the call, Infer would blame the caller.

3 Infer

3.1 Bi-abduction and Compositional Analysis

The theoretical notion used by Infer to automatically synthesize specifications is *bi-abductive inference* [5]. It consists in solving the following extension of entailment problem: $H * X \vdash H' * Y$. Here H and H' are given formulae in separation

[2] We use the standard separation logic notation: $x\mapsto y$ describes a single heap-allocated cell at address x whose content is y; we write "–" to indicate *some* value; *emp* represents the empty heap; the $*$ operator separates allocated cells.

logic and X (anti-frame) and Y (frame) needs to be deduced. The usefulness of bi-abductive inference derives from the fact that it allows to discover the part of the heap missing (the anti-frame in the above entailment) for using a specification within a particular calling context of another procedure. For example, consider the specification

$$\{x \mapsto -\}\ \texttt{void use\_cell(int *x)}\{emp\}$$

and the function `void g(int *x, int y) { y=0; use_cell(x);}` . Infer uses bi-abductive inference for comparing the calling heap of `use_cell` within `g` against the spec's preconditon. In doing so, Infer will understand by means of the resulting anti-frame that `g`'s precondition must require `x` to be allocated. The construction of `g`'s specification is therefore compositional (that is: the specs of a procedure are determined by the specs of the procedures it calls). The preconditions of specs computed with bi-abductive inference approximate the procedures' footprint (the parts of memory that a procedure uses). One consequence of this feature when combined with the principle of local reasoning is that these specs can be used independently of the calling context in the program. Moreover, these specs can be used as procedure summaries for implementing an inter-procedural shape analysis. Such analysis can be seen as the attempt to build proofs for Hoare triples of a program. The triples are constructed by symbolically executing the program and by *composing* triples of procedures in the program in a bottom-up fashion according to the call graph. For mutually-recursive procedures an iterative fixed-point computation is performed. This bottom-up analysis provides Infer with the ability to analyze *incomplete code* (a useful features since in practice the entire program is not always available).

An immediate consequence of Infer's compositional nature is its great ability to scale. Procedures are analyzed in isolation, and therefore, when analyzing large programs, only small parts of the source code needs to be loaded into memory. Infer has a low memory requirement even when analyzing programs composed by millions of lines of code. Moreover, the analysis results can be reused: Infer implements an *incremental analysis*, and in successive runs of the analysis of the same program, only the modified procedures need to be re-analyzed. The results of previous analyses for unchanged procedures are still valid.

3.2 Infer's Architecture

Figure 2 shows Infer's basic architecture. Infer refers to a collection of related source code files to be analyzed together as a 'project'. When analyzing a project, a number of intermediate files are created and stored in the "Project Results Directory" for use between different phases of the verification.

The Infer system consists of three main components: InferCapture, InferAnalyze, and InferPrint. These implement the three phases of the verification.

Capture Phase. Like ordinary compilation, the source code is parsed and converted to an internal representation necessary for analysis and verification.

Analysis Phase. This phase performs the actual static and verification analysis, based on the internal representation produced during the Capture Phase.

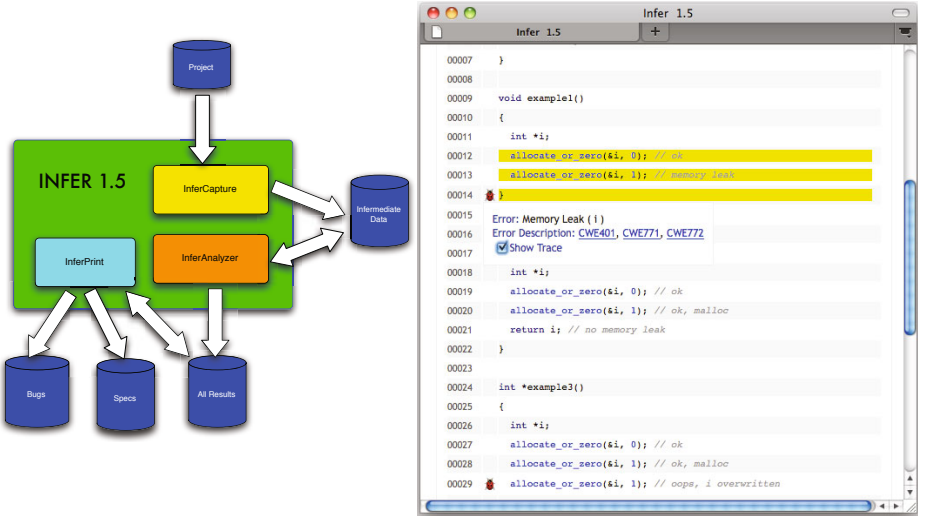

Fig. 2. Left: Infer's architecture. Right: screenshot of error trace high-lighted in yellow.

Results Post-Processing Phase. For the set of analyzed procedures where bugs have been identified, a list of bugs can be output in CSV or XML format. In addition, the list of specifications obtained for the set of procedures can be generated in either text or graphical format for use with external tools.

3.3 Implementation

Infer is available as a command line tool, in the cloud, or as an Eclipse plug-in.
Command-line. Infer can be used in a terminal as a command-line tool. This facilitates the integration in an existing tool chain. The output are lists of errors and procedure specifications in several formats (CSV, XML, dotty, SVG).
In the cloud. Alternatively, Infer comes with a GUI which can be used with any web-browser. In this version, the core back-end and the GUI are hosted in servers in the cloud. Users can login into their account, upload projects and run the analysis from anywhere. The GUI visualizes errors and procedure specifications in a user-friendly way (see the screenshot on the right of Fig. 2). Moreover, statistics comparing results of different runs of the analysis on the same projects are also visualized. This makes it easy for programmers or managers to track improvements of the reliability of their software during the development process.
Eclipse plug-in. Infer can also be used within Eclipse with a special plug-in. The developer can benefit from a complete integrated environment which goes from editing, to compilation, to verification. Since Infer can analyze incomplete code, the developer can constantly check his code before committing to the central repository of his organization and avoid critical errors at very early stages.

4 Related Work and Conclusions

Recent years have seen impressive advances in automatic software verification thanks to tools such as SLAM [1] and BLAST [10], which have been used to verify properties of real-world device drivers and ASTREE [3] applied to avionics code. However, while striking in their domain, these tools either eschew dynamic allocation altogether or use coarse models for the heap that assume pointer safety. Instead, as shown in this paper, these are the areas of major strength of Infer. Several academic tools have been proposed for automatic deep-heap analysis but only on rare cases some of these have been applied to real industrial code [12,9,6,13,7,11]. To our knowledge, Infer is the first industrial-strength tool for automatic deep-heap analysis applicable to C programs of any size.

Conclusions. This paper has presented Infer, a commercial tool for proving memory safety of C code. Based on separation logic, Infer is precise in the presence of deep-heap updates and dynamic memory allocation. Thanks to the compositional nature of its analysis, Infer is able to scale to large industrial code.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of C programs. In: PLDI, pp. 203–213 (2001)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
4. Calcagno, C., Distefano, D., O'Hearn, P.W., 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.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300. ACM, New York (2009)
6. Chang, B., Rival, X., Necula, G.: Shape analysis with structural invariant checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
7. Chin, W.-N., David, C., Nguyen, H., Qin, S.: Enhancing modular OO verification with separation logic. In: Proceedings of POPL, pp. 87–99. ACM, New York (2008)
8. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H. (ed.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
9. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL, pp. 310–323 (2005)
10. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
11. 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)
12. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: SAS 2000. LNCS, vol. 1824, pp. 280–302. Springer, Heidelberg (2000)

13. Marron, M., Hermenegildo, M., Kapur, D., Stefanovic, D.: Efficient context-sensitive shape analysis with graph based heap models. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 245–259. Springer, Heidelberg (2008)
14. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)