IKOS: A Framework for Static Analysis based on Abstract Interpretation (Tool Paper)

Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet

NASA Ames Research Center, Moffett Field, CA 94035

Abstract. The RTCA standard (DO-178C) for developing avionic software and getting certification credits includes an extension (DO-333) that describes how developers can use static analysis in certification. In this paper, we give an overview of the IKOS static analysis framework that helps developing static analyses that are both precise and scalable. IKOS harnesses the power of Abstract Interpretation and makes it accessible to a larger class of static analysis developers by separating concerns such as code parsing, model development, abstract domain management, results management, and analysis strategy. The benefits of the approach is demonstrated by a buffer overflow analysis applied to flight control systems.

1 Introduction

Our goal is to enable the use of static analysis for the certification of avionic systems. The DO-333 extension to DO-178C lists Abstract Interpretation [4] as a possibility to obtain certification credits. Unfortunately, there are few available commercial static analyzers based on Abstract Interpretation. Moreover, they often lack precision and scalability for C/C++ code, or, they are restricted to strict subsets of C. Our goal is to define a framework that can be used to develop precise, scalable static analyses based on Abstract Interpretation for flight software systems.

Abstract Interpretation [4] is a theoretical framework that provides a methodology for constructing sound static analyses. It offers mathematical guarantee that all properties computed by the analyzer hold for all possible execution paths of the program. The core idea behind this theory is the careful use of the notion of approximation: all possible values that a variable can take at a certain program point are approximated by a set that can be compactly represented (e.g., an integer interval), thus ensuring the soundness of the analysis. However, infeasible value assignments of the variable may be introduced because of the approximation. This may result into false alarms, where the analyzer detects a potential problem at a statement when the program is actually safe. These false alarms need to be as rare as possible; otherwise, it defeats the usefulness of the analysis. Nevertheless, when a statement is deemed safe, it can never cause an error, which is a key property for certification.

In this paper, we give an overview of IKOS [1] (Inference Kernel for Open Static Analyzers), an open-source framework developed at the NASA

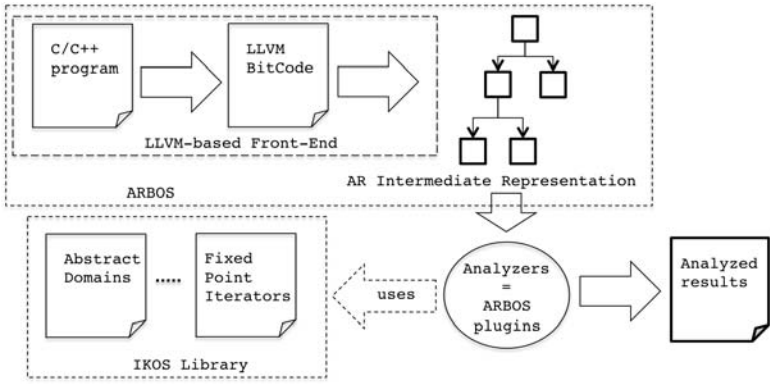

Fig. 1. The IKOS framework architecture overview.

Ames Research Center that supports the development of precise and scalable static analyses. IKOS provides abstract interpretation concepts for developing specialized analyzers, which helps drive down the number of false positives without compromising scalability. Designing a specialized analyzer using standard methods is long and difficult. IKOS facilitates this process by factoring out most of the expertise required to write the analyzer. The use of IKOS in developing precise and scalable static analyses is demonstrated through the implementation of a buffer overflow analysis and its application to flight control systems. Arrays and buffers are pervasively employed in flight software (navigation, communication) and errors are often very hard to catch during standard V&V activities, like testing or code review.

2 Framework Overview

The IKOS framework, shown in Figure 1, offers capabilities to facilitate the development and integration of the traditional elements of static analyzers. IKOS relies on the ARBOS plugin framework for parsing the source code, performing semantic resolution, and creating an intermediate representation (using the AR form) more suitable for analysis (i.e., semantic equations that need to be solved using fixpoint iterations). Analyses are developed as ARBOS plugins using the Abstract Interpretation concepts available in the IKOS library (a collection of abstract domains and fixpoint iteration algorithms). Currently, results are being stored in permanent storage in the form of text files or in an SQL database (SQLite). The SQL database is convenient to extract specific information from the results. We can also visualize the results using an external tool, called IkosView, which shows the location of the checks in the source code with the traditional color coding: green for safe checks, red for unsafe checks, and orange for warnings. In the following sections, we will describe ARBOS and its AR form, then IKOS, and finally some of the analyses we have developed.

2.1 The ARBOS Plugin Framework

ARBOS is a plugin framework that allows the definition of static analyses using the AR form as intermediate representation. Currently, ARBOS includes a front-end (based on LLVM [7]), which translates C/C++ code into AR, the abstract representation of programs, and the APIs that facilitate writing static analyses as ARBOS plugins. The workflow in Figure 1 shows the various phases to obtain the AR representation of the C/C++ source code to be analyzed.

2.2 Why LLVM?

LLVM [7] is essentially a high-level, platform-independent assembly language. Although it is simpler to process than the abstract syntax tree of a C/C++ program, it heavily relies on the static single assignment form (SSA) which cannot be readily used to design an abstract interpreter. The ϕ-nodes need to be eliminated and other inadequate constructs need to be simplified using various LLVM transformation passes. An intermediate representation like CIL [9] is far more adapted to the design of static analyzers based on Abstract Interpretation than LLVM, so why choose LLVM in the first place?

The single most important issue facing the user of a static analysis tool is getting the code through the tool's parser. *All* commercial static analyzers, sound or not, use their own parsers, which may not accept C/C++ dialects or idioms that are commonplace in the embedded world, including flight systems. For example, in our experiments the code for the UAS autopilot `Paparazzi` listed in Table 1 was rejected by all three commercial static analyzers we had licenses for. Getting the application through those tools would have required rewriting huge chunks of code, which is unacceptable in general and for the certification of flight software in particular, where code cannot be changed at all. However, `Paparazzi` would compile without any problem when using the GCC compiler. As a matter of fact, all the programs we have studied would compile without any modification using GCC whereas they would require some modifications when using commercial tools. There is a special version of GCC that has been modified so as to generate LLVM bitcode, which is why we are using LLVM. CIL has its own front-end and trips when parsing nonstandard code, very much like all other C/C++ front-ends but GCC.

2.3 The Abstract Representation

The Abstract Representation (AR) generated by ARBOS can be used as an input to abstract interpreters implemented using IKOS as ARBOS plugins. Compared to the LLVM Internal Representation (IR), the AR takes a different angle on how to express the semantics of a C program. For example, the SSA form is done away with, the instruction set is more regular and the control flow is expressed in a declarative way using nondeterministic choices and assertions rather than conditional branch

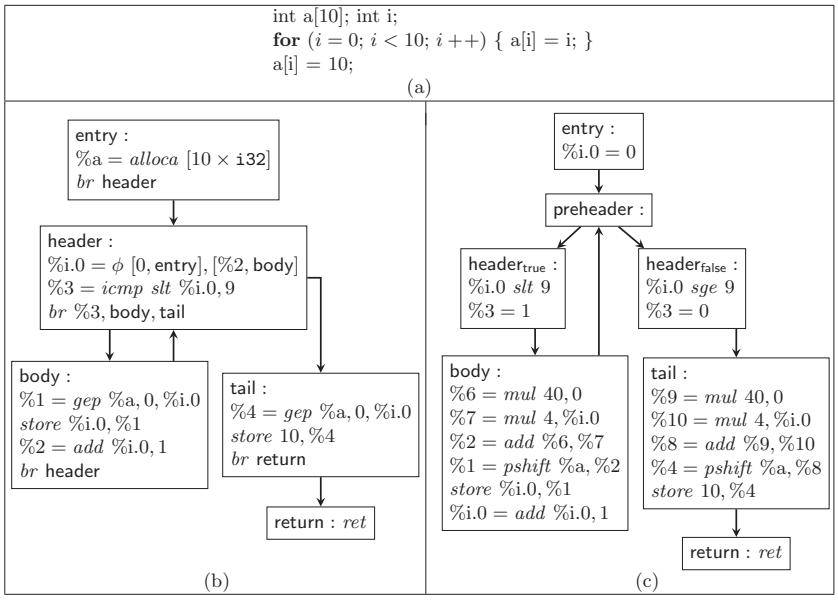

Fig. 2. Code snippet (a) with LLVM IR (b) and ARBOS AR (c) forms.

instructions. Due to space limitations, we cannot describe the entire AR form in this paper. Instead, we will highlight the main differences between the LLVM IR and AR using a simple example. It is worth noting that there is no loss of expressiveness when translating the LLVM IR into AR.

Figure 2(a) shows a simple piece of code performing an array initialization with special treatment for the last element. The loop is not correctly written, which causes an out-of-bounds array access at the last statement. This example is a redacted and simplified version of a problem identified in a real flight code during V&V activities. Figures 2(b) and 2(c) show the LLVM IR and ARBOS AR forms, which have also been simplified for readability. We will now go over the details of the translation from the LLVM IR into AR.

In SSA form variables can only be assigned once, which requires the introduction of ϕ-nodes to represent the values a variable can take at a merge point in the control-flow graph. In Abstract Interpretation, a merge point corresponds to the application of a join (or widening) operation in the abstract domain. A ϕ-node represents a partial disjunction over some program variables, which can be dealt with easily when considering non-relational domains (like intervals). However, relational abstractions (like polyhedra) describe properties over *all* program variables, which makes the treatment of ϕ-nodes extremely challenging. Since IKOS is meant to be a generic abstract interpretation framework, ϕ-nodes are removed from the AR by inserting assignment instructions that simulate the effect of the ϕ-nodes on the variables concerned.

In the LLVM IR, conditional branch instructions (*br*) are coupled with Boolean instructions that return their result in a register (*slt*, *sge*). This implies that processing the condition in the abstract domain (e.g., by using a linear constraint solver) should be done in conjunction with assigning a discrete value to a variable (the result of the operation) and propagating the invariant across basic blocks (to take care of both branches). Since IKOS is a generic static analyzer, we need to decouple these aspects so as not to make the structure of the fixpoint iterator dependent on any particular abstract domain. This is why branch instructions are eliminated for the AR and replaced by nondeterministic choices (over blocks $\mathsf{header}_{\mathsf{true}}$ and $\mathsf{header}_{\mathsf{false}}$ in our example).

Finally, complex instructions in the LLVM IR that model pointer arithmetic (*gep*) are replaced by atomic operations on the pointer offsets expressed in bytes (e.g., the pointer shift operation *pshift*). Once all these transformations have been applied, the resulting AR form can be processed by the generic algorithms of IKOS. Instantiating these algorithms with the domain of intervals provides enough precision to statically resolve all array access checks and identify the error in the example.

2.4 The IKOS Library

IKOS is a development platform for static analyzers based on Abstract Interpretation. IKOS is actually a large library of optimized Abstract Interpretation algorithms. It is accessible through a highly generic API. IKOS is meant to offer a cost-effective way of designing specialized static analyzers. The API for abstract domains provide the usual services from the abstract interpretation theory, i.e., abstract operators, comparison operators, lattice elements such as bottom and top, and narrowing and widening operators. Currently, IKOS offers implementations for the following numerical abstract domains: constants, intervals, arithmetic congruences [3], octagons [8] and discrete symbolic domains. Other domains are under development. IKOS also provides an API for fixpoint iterators.

3 Buffer Overflow Analysis

We have used IKOS to implement an interprocedural buffer-overflow analyzer for avionic codes in C. The analysis represents less than 600 lines of C++. This analysis is interprocedural and performs a full expansion of function calls, very much like Astrée [5]. We have run our buffer overflow analysis on a set of C flight control systems ranging in size from 35 KLOC to 278 KLOC. The analysis was conducted on a MacBook Pro with a 2.8 GHz Intel Core i7 processor and 16 GB of memory. In our results presented in Table 1, we include analysis times to give an idea of the speed of the analysis. We did not try to optimize the analysis speed. We focused on the precision, which is measured as the percentage of analysis checks that are classified (as safe or unsafe) with certainty as opposed to checks that yield warnings (there may or may not be a problem).

Code	Size	Analysis Time	Precision
`Paparazzi`	35 KLOC	22s	99%
`Gen2`	22 KLOC	1m03s	98%
`FLTz`	144 KLOC	10m30s	91%
`Arduplane`	278 KLOC	6m30s	94%

Table 1. Buffer Out-of-Range Analysis Results.

The goal was to create an analysis that would yield less than 10% false positives on flight control codes. The results on our test suite seem to indicate that we have reached this goal since we always have a precision higher than 90%. The results also show that we have not sacrificed analysis times for precision since all analyses are done in a matter of minutes or less. We are in the process of identifying a large embedded system code base so that we can truly characterize the scalability of the analysis. Note that our measure of analysis time is really coarse since we do not attempt to separate time spent analyzing the code from time spent logging results to a file or a database. Our past experience with C Global Surveyor (CGS) showed that logging results takes a significant amount of time. So, we find our measured analysis times encouraging.

4 Related Work

The closest related work comes from tools relying on the abstract interpretation framework, namely C Global Surveyor, CodeHawk, Astrée, and PolySpace Verifier.

C Global Surveyor [10] (CGS) is the ancestor of IKOS, and, it has had a large influence on the design of IKOS. However IKOS is a framework to build analyzers when CGS is an analyzer. The emphasis in IKOS is to factor out the difficult concepts (abstract domains, fixed-point iterators). We also changed the front-end (GCC/LLVM instead of EDG) and the database (SQLite instead of PostGReSQL). Finally, the precision of CGS is not on par with IKOS' precision since CGS generally produces about 20% warnings when IKOS is usually in the 1% to 2% range.

CodeHawk [6] is the closest tool to IKOS. It is also a framework for developing analyses based on abstract interpretation. CodeHawk is a commercial tool and little public data is available.

Astrée [5] was customized for specific Airbus codes. The impressive results it achieves inspired us to enable the construction of specialized static analyzers. The Airbus code is essentially composed of filters, which means that Astrée focused on floating-point computation, which is not yet addressed by IKOS. IKOS is addressing a much larger class of C programs. In all honesty, we only can compare to the original version of Astrée, not the current commercial one.

PolySpace Verifier was the first of this line of static analyzers based on Abstract Interpretation. In many ways, it paved the way for the current

generation of tools. Polyspace Verifier was very successful in analyzing Ada code but fell short for C and C++. In our own experience [2], scalability was a big issue and the number of warnings was also important (20% to 50% of all checks).

5 Conclusion

We have given an overview of IKOS, an open-source platform which facilitates the development of static code analyzers based on Abstract Interpretation. The front-end of IKOS relies on LLVM, but it can be easily replaced by other front-ends since the analyses run on our own intermediate representation. We demonstrated the precision and scalability of IKOS-based analyzers with an interprocedural buffer overflow analysis.

References

1. IKOS: Inference Kernel for Open Static Analyzers, http://ti.arc.nasa.gov/opensource/ikos/
2. Brat, G., Klemm, R.: Static Analysis of the Mars Exploration Rover Flight Software. In: Space Mission Challenge for Information Technology. pp. 321–326 (2003)
3. Bygde, S.: Abstract Interpretation and Abstract Domains with special attention to the congruence domain. Master's thesis, Mälardalen University, Sweden (2006)
4. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. pp. 238–252 (1977)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astreé Analyzer. In: ESOP. pp. 21–30 (2005)
6. Kestrel Technology: CodeHawk, http://www.kestreltechnology.com
7. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO'04 (2004)
8. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In: CC '02. pp. 213–228 (2002)
10. Venet, A., Brat, G.P.: Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. In: PLDI. pp. 231–242 (2004)