Analysis of Low-Level Code Using Cooperating Decompilers*

Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula

University of California, Berkeley, California, USA
{bec, matth, necula}@cs.berkeley.edu

Abstract. Analysis or verification of low-level code is useful for minimizing the disconnect between what is verified and what is actually executed and is necessary when source code is unavailable or is, say, intermingled with inline assembly. We present a modular framework for building pipelines of cooperating decompilers that gradually lift the level of the language to something appropriate for source-level tools. Each decompilation stage contains an abstract interpreter that encapsulates its findings about the program by translating the program into a higher-level intermediate language. We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc`, `gcj`, and `coolc` (a compiler for a pedagogical Java-like language) that share several low-level components. Finally, we discuss our experimental results that apply the BLAST model checker for C and the Cqual analyzer to decompiled assembly.

1 Introduction

There is a growing interest in applying software-quality tools to low-level representations of programs, such as intermediate or virtual-machine languages, or even on native machine code. We want to be able to analyze code whose source is either not available (e.g., libraries) or not easily analyzable (e.g., programs written in languages with complex semantics such as C++, or programs that contain inline assembly). This allows us to analyze the code that is actually executed and to ignore possible compilation errors or arbitrary interpretations of underspecified source-language semantics. Many source-level analyses have been ported to low-level code, including type checkers [23, 22, 8], program analyzers [26, 4], model checkers [5], and program verifiers [12, 6]. In our experience, these tools mix the reasoning about high-level notions with the logic for understanding low-level implementation details that are introduced during compilation, such as stack frames, calling conventions, exception implementation, and data layout. We would like to segregate the low-level logic into separate modules to allow for easier sharing between tools and for a cleaner interface with client analyses. To

* This research was supported in part by the National Science Foundation under grants CCF-0524784, CCR-0234689, CNS-0509544, and CCR-0225610; and an NSF Graduate Research Fellowship.

K. Yi (Ed.): SAS 2006, LNCS 4134, pp. 318–335, 2006.

better understand this issue, consider developing a type checker similar to the Java bytecode verifier but for assembly language. Such a tool has to reason not only about the Java type system, but also the layout of objects, calling conventions, stack frames, with all the low-level invariants that the compiler intends to preserve. We reported earlier [8] on such a tool where all of this reasoning is done simultaneously by one module. But such situations arise not just for type checking but essentially for all analyses on assembly language.

In this paper we propose an architecture that modularizes the reasoning about low-level details into separate components. Such a separation of low-level logic has previously been done to a certain degree in tools such as CodeSurfer/x86 [4] and Soot [28], which expose to client analyses an API for obtaining information about the low-level aspects of the program. In this paper, we adopt a more radical approach in which the low-level logic is packaged as a *decompiler* whose output is an intermediate language that abstracts the low-level implementation details introduced by the compiler. In essence, we propose that an easy way to reuse source-level analysis tools for low-level code is to decompile the low-level code to a level appropriate for the tool. We make the following contributions:

- We propose a decompilation architecture as a way to apply source-level tools to assembly language programs (Sect. 2). The novel aspect of our proposal is that we use decompilation not only to separate the low-level logic from the source-level client analysis, but also as a way to modularize the low-level logic itself. Decompilation is performed by a series of decompilers connected by intermediate languages. We provide a *cooperation* mechanism in order to deal with certain complexities of decompilation.
- We provide evidence for the modularity of this framework through the implementation of multiple decompilation pipelines for both x86 and MIPS assembly produced by `gcc` (for C), `gcj` (for Java), and `coolc` (for Cool [1], a Java-like language used for teaching) that share several low-level components (Sect. 3). We then compare with a monolithic assembly-level analysis.
- We demonstrate that it is possible to apply source-level tools to assembly code using decompilation by applying the BLAST model checker [18] and the Cqual analyzer [17] with our `gcc` decompilation pipeline (Sect. 4).

Note that while ideally we would like to apply analysis tools to machine code binaries, we leave the difficult issue of lifting binaries to assembly to other work (perhaps by using existing tools like IDAPro [19] as in CodeSurfer/x86 [4]).

Challenges. Just like in a compiler, a pipeline architecture improves modularity of the code and allows for easy reuse of modules for different client-analyses. Fig. 1 shows an example of using decompilation modules to process code that has been compiled from C, Java, and Cool. Each stage recovers an abstraction that a corresponding

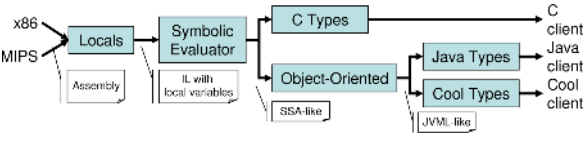

Fig. 1. Cooperating decompilers