Certification of compiled assembly code by invariant translation

Xavier Rival

École Normale Supérieure, 45, rue d’Ulm, 75 230, Paris, France
e-mail: rival@di.ens.fr
Published online: 6 April 2004 – © Springer-Verlag 2004

Abstract. We present a method for analyzing assembly programs obtained by compilation and checking safety properties on compiled programs. It proceeds by analyzing the source program, translating the invariant obtained at the source level, and then checking the soundness of the translated invariant with respect to the assembly program. This process is especially adapted to the certification of assembly or other machine-level kinds of programs. Furthermore, the success of invariant checking enhances the level of confidence in the results of both the compilation and the static analysis. From a practical point of view, our method is generic in the choice of an abstract domain for representing sets of stores, and the process does not interact with the compilation itself. Hence a certification tool can be interfaced with an existing analyzer and designed so as to work with a class of compilers that do not need to be modified. Finally, a prototype was implemented to validate the approach.

Keywords: Static program analysis – Certified compilation – Abstract interpretation

1 Introduction

Critical software is concerned with safety; hence various static analysis methods have been developed and are applied to critical programs. However, these methods are usually applied to the source program and the source analysis may not be considered a trustable proof given that the compiler may be incorrect and the compiled program unsafe even if the source analysis succeeds in proving safety. Indeed, modern compilers turn out to be very complex due to the size of their source code and to their perpetual evolution (for instance, the code of the current versions of gcc amounts to about 500 000 lines). Therefore, most critical applications like avionics require the certification of the form of the program that is actually executed, i.e., the assembly code itself.

Moreover, the safety properties of interest usually concern the very execution of the program; hence, checking it on the compiled program (i.e., the version that is actually executed) yields more trustable proofs of safety. For instance, the semantics of errors is defined at the machine level first. The memory access errors (out-of-bound array access or void pointer dereference in C programs) are the source language counterpart for some assembly errors (attempt to access a wrong part of memory). If we prove that a source C program does not yield any memory access error, then we can deduce that a compiled form of this program is memory safe only under some additional assumptions, i.e., mainly that the program is compiled in a correct way for some definition of “correct” that should be made explicit and that the memory allocation is done at the assembly level in a safe way, which should also be made explicit. Furthermore, the nature of the undesirable behaviors may be compiler or even architecture dependent, as is the case for overflows: the size of registers depends on the target processor and the way integer data types are compiled affects the overflows that occur in the compiled program (this is especially true for data types that do not correspond to the size of registers like short integer data types). Languages like C leave many error cases as unspecified in order to leave the compiler implementator free when designing more optimizations. For example, an out-of-bound array index in a C program results in an undefined behavior, which may be an immediate error or a wrong yet continued execution. Therefore, checking safety properties at the assembly level is noticeably advantageous – in particular when dealing with highly critical software.

As a way to achieve that, we may envisage certifying the assembly program directly. However, analyzing directly and efficiently precise high-level properties of as-
assembly programs may be quite difficult due to a loss of
structure at compile time. In particular, the control struc-
ture of assembly programs is based on gotos, which are
much more complicated to analyze than loops. Static ana-
lysis methods for improving speed and precision apply in
an easier way to well-structured loops than general con-
roll flow graphs. Furthermore, the data structures (like
arrays, records, or enums) are translated into more com-
licated assembly structures since everything turns into
a sequence of memory cells and low-level details should be
taken into account (as memory cell alignments). On the
other hand, the formal (semiautomatic) proof of a full C
compiler cannot be envisaged on account of the work task
that would be involved in such a project and because any
modification or evolution of the compiler would make the
proof dated (proving a commercial compiler is not a real-
istic solution). The last limitation also applies to a system
that would translate a proof of safety at compile time.

The solution proposed here is to analyze the source
version of the program using an automatic tool and to
derive automatically a “candidate invariant” for the
assembly program. This invariant is obtained by translating
the source invariant thanks to some information about
the way the program is compiled (in most cases, this ad-
ditional information can be found in the debugging in-
formation provided by the compiler, which describes the
correspondence between source and target variables and
program points). Then an automatic tool checks that the
candidate invariant is semantically sound: it is an upper
approximation of the set of reachable states of the pro-
gram. If program $P_c$ is obtained by compiling program
$P_s$, the method proceeds as follows. A source analyzer
generates an invariant $\mathcal{P}_s$ for the source program
and an external tool derives the candidate invariant $\mathcal{P}_c$; then
an assembler checks to prove that property $\mathcal{P}_c$
holds for program $P_c$. Afterwards, property $\mathcal{P}_c$ can be
used for verifying that $P_c$ satisfies the desired safety prop-
erties. Note that this approach allows one to derive ben-
fit from existing fast and precise source analyzers (like
those of [3, 4]).

Our method does not require the instrumentation of
the compiler; if the debugging information format is stan-
dard, we can even consider designing a tool that would
translate invariants for certifying assembly programs pro-
duced by a class of compilers. Moreover, we need to cope
with the specifics of assembly programs for the checking
of invariants only and not for their inference. When the
checking succeeds, the translated invariant can be con-
considered correct under only one assumption: the checker
must be correct. Therefore, the security level achieved by
this approach is the same as that of a direct analysis of
the assembly code. Moreover, the success of the check-
ing entails a correctness result about the compilation: the
target program presents behaviors similar to those of the
source program (in the abstract semantics point of view).
On the other hand, the method is incomplete: a failure
of the invariant checking does not entail that the com-
piler is buggy; it may be due to a loss of precision at
translation time or at checking time. The approach pro-
posed here is formalized inside the abstract interpretation
framework [10, 11], which provide an integrated view in
a single framework of both static analysis [4, 7] and pro-
gram transformations [13] (hence compilation). Furth-
more, we validated our approach by designing a proto-
type aimed at checking the absence of runtime errors and
undefined behaviors in PowerPC assembly programs ob-
tained by compiling realistic C programs. Our choice of
the C language was justified by the use of this language in
safety-critical systems.

**Plan.** The rest of the paper is organized as follows. Sec-
tion 2 presents preliminaries and describes the source and
assembly languages considered in subsequent sections of
the paper. We formalize the compilation correctness in
Sect. 3. Section 4 describes a class of static analyses large
enough for answering most of the safety questions about
imperative source programs and shows how an invariant
can be derived at the assembly level from a source in-
variant. Section 5 discusses the problem of checking the
translated invariant independently of the source analysis.
In Sect. 6 we detail the practical problems that arise when
checking the invariant at the assembly level. The proto-
type we implemented is described in Sect. 7. Section 8
concludes the paper.

**Related work.** Most attempts at formally proving a com-
piler have concentrated on rather high-level languages
and byte code assembly languages [27] or on toy compi-
lers written for that purpose [5]. The lack of automation
of theorem provers severely limits the possibility of proving
large programs in general and compilers in particular.

Among direct static analyses of assembly programs,
we can cite the determination of properties about cache
usage (cache misses and cache hits) presented in [1], the
analysis of pipeline behavior in [30], and the combination
of these two analyses in [31]: precise information could be
inferred about the worst-case execution time of assembly
programs by taking into account many complex aspects
of the architecture. However, we are not aware of any ex-
ample of direct analysis for high-level properties at the
assembly level.

The idea of translating at compile time semantic in-
formation about the source program into information
about the assembly program was developed in the Proof-
Carrying Code approach described in [2, 22]. In this ap-
proach an untrusted compiler is supposed to provide an-
notations with the assembly code it produces. Before it
executes the target program, the code consumer gener-
ates verification conditions to check that the assembly
program does not violate the safety policy, and the code
consumer attempts to prove them using the annotations
supplied by the compiler. If it succeeds, then the assembly
code obeys the safety policy and can be executed safely.
The compiler of [24] implements this methodology. In this
case, the compiler annotations are type information.