DOI:10.1145/2043174.2043196

Technical Perspective
Safety First!

By Xavier Leroy

SOFTWARE MISBEHAVES ALL too often. This is a truism, but also the driving force behind many computing techniques intended to increase software reliability, safety, and security, ranging from basic testing to full formal verification.

In this wide spectrum of approaches, a sweet spot is type and memory safety. Rather than attempting to rule out all bugs, type and memory safety focuses on strict enforcement of a few basic safety properties: a character string is not a code pointer; arrays are always accessed within bounds; memory blocks are not accessed after deallocation; pointers or object references cannot be forged from integers; and so on. Such properties are enforced through a combination of static (compile-time) type-checking, dynamic (runtime) checks such as array bound checks, and automatic memory management. These humble safety properties not only catch a number of common programming errors, but are also surprisingly effective at thwarting many security attacks such as buffer overrun attacks. Moreover, they can be leveraged to build software-enforced access control and isolation architectures such as the Java and .NET security managers; for if object references can be forged from integers, any software-only security infrastructure can be circumvented.

In the mid-1990s came the realization that type and memory safety is not just for high-level programming languages. Java and its bytecode verifier popularized the idea that the bytecode of a virtual machine can be made type-safe through a combination of load-time type-checking (bytecode verification) and runtime checks in the virtual machine. Going one step further "down," Morrisett, Walker, Crary and Glew introduced *Typed Assembly Language* (TAL), which guarantees type and memory safety directly at the level of assembly language for the ubiquitous x86 processor architecture. There are several benefits to enforcing type and memory safety at the level of bytecode or assembly language: the compilers no longer need to be trusted to preserve safety and therefore are no longer part of the trusted computing base. Moreover, type-safe interoperability between different source languages can be guaranteed.

The following work by Yang and Hawblitzel is a major milestone in an ambitious research project: that of guaranteeing end-to-end type and memory safety for a complete software stack. Leveraging the Bartok .NET-to-typed-x86 compiler and the corresponding TAL checker, it is possible to automatically obtain safety guarantees for most of the software stack written in C#—not just application code, but also large chunks of systems code such as network protocols. In particular, the paper shows that the major part of a safe, preemptive scheduler for multitasking can be developed this way, which may come as a surprise to many readers.

However, not all parts of an operating system and runtime system can be shown memory-safe using only TAL. A major offender is the memory manager (allocator, garbage collector, among others), which has to treat memory in an essentially untyped way. Similar issues occur in the lowest layers of operating systems (context switching, interrupt handling, among others). The standard approach at this point is to leave these components in the trusted computing base and validate them only by testing. Instead, Yang and Hawblitzel succeeded in formally verifying these components—which they call the "Nucleus" of their Verve operating system—against mathematical specifications (pre- and post-conditions), using the Boogie deductive program verifier.

The following work is a major milestone in an ambitious research project: that of guaranteeing end-to-end type and memory safety for a complete software stack.

The minimalistic design of the Nucleus is elegant, and the interplay between its specifications and the generic safety guarantees of the TAL code is subtle. Perhaps the most impressive aspect of this work, however, is the remarkable economy of means by which it achieves end-to-end type and memory safety. The high degree of automation offered by the Boogie verifier and Z3 automatic theorem prover does wonders here, resulting in an overall verification effort that is remarkably low by today's standards.

The formal verification of high-assurance software is making great progress lately. Yang and Hawblitzel's work, along with other recent breakthroughs in software verification such as the seL4 verified microkernel of Klein et al. (see *Communications*, June 2010, p. 107), were unthinkable 10 years ago. Little by little, one point at a time, these results sketch a promised land where, with mathematical certainty, software does behave properly after all.

Xavier Leroy (xavier.leroy@inria.fr) is a senior research scientist at INRIA Paris-Rocquencourt, France.