Sulong, and Thanks for All the Fish∗

Manuel Rigger  
Johannes Kepler University Linz  
Austria  
manuel.rigger@jku.at

Roland Schatz  
Oracle Labs  
Austria  
roland.schatz@oracle.com

Jacob Kreindl  
Johannes Kepler University Linz  
Austria  
jacob.kreindl@jku.at

Christian Häubl  
Oracle Labs  
Austria  
christian.haeubl@oracle.com

Hanspeter Mössenböck  
Johannes Kepler University Linz  
Austria  
hanspeter.moessenboeck@jku.at

ABSTRACT
Dynamic languages rely on native extensions written in languages such as C/C++ or Fortran. To efficiently support the execution of native extensions in the multi-lingual GraalVM, we have implemented Sulong, which executes LLVM IR to support all languages that have an LLVM front end. It supports configurations with respect to memory-allocation and memory-access strategies that have different tradeoffs concerning safety and interoperability with native libraries. Recently, we have been working on balancing the tradeoffs, on supporting inline assembly and GCC compiler builtins, and on executing a complete libc on Sulong.

CCS CONCEPTS
• Software and its engineering → Virtual machines; Interpreters; Dynamic compilers;

KEYWORDS
Sulong, LLVM, GraalVM

ACM Reference Format:

1 INTRODUCTION
The GraalVM [19, 20] is a multilingual virtual machine that can execute Ruby [14], JavaScript, R [16], and Python [21]. The individual language implementations are built on a common framework (called Truffle), which is written in Java [18]. Thus, a common compiler (called Graal) can optimize the execution of these languages during run time [2, 7, 15, 17].

Programmers of dynamic languages make frequent use of foreign function interfaces to call native extensions written in languages such as C/C++ or Fortran. To support native extensions efficiently, executing them on Truffle would be desirable, as the compiler could optimize across language boundaries and mitigate the overhead that other native function interfaces incur [4, 5]. To tackle this issue and to provide an efficient native function interface, we have implemented a Truffle language implementation as part of project Sulong [9].

Sulong’s Truffle interpreter executes LLVM IR [6], which is a machine-independent format that specifies an instruction set resembling RISC assembly. By executing LLVM IR, Sulong can execute programs written in many languages, including C/C++ and Fortran. Recently, we have been working on enabling the use of native libraries in programs while still providing a partially safe execution, on supporting execution of a complete libc, and on supporting inline assembly and GCC builtins.

2 ARCHITECTURE
Figure 1 gives an overview of the Sulong system.

LLVM. Sulong’s interpreter executes LLVM IR, which is part of the LLVM compiler framework [6] and can be produced by many language front ends. For example, LLVM’s Clang front end can compile C/C++ code. Flang and GCC (with the DragonEgg plugin) can compile Fortran code to LLVM IR.

LLVM IR interpreter. Sulong’s interpreter is written in Java using the Truffle language implementation framework. Truffle is a language implementation framework that facilitates the implementation of interpreters based on Abstract Syntax Trees (ASTs). Each operation is implemented as a node that returns its result in an execute() method. If a function is executed frequently, Truffle uses the Graal compiler to compile it to machine code, by inlining through all nodes’ execute() methods (which is a form of partial evaluation [3]). As described below, Sulong’s interpreter can be configured to use different strategies for handling allocations and accessing memory in the user program; depending on its strategy, we refer to Sulong as Native Sulong, Safe Sulong, or Managed Sulong.

Native Sulong. Native Sulong prioritizes compatibility with native libraries over safety [10]. It supports passing objects allocated by the LLVM IR interpreter in the user program (e.g., an allocation by malloc in C) to machine-code functions. Native Sulong achieves this by allocating unmanaged memory for all user-program objects,

∗The title is a reference to Douglas Adams’ fourth book So Long, and Thanks for All the Fish of the Hitchhiker’s Guide to the Galaxy.

†Available at https://github.com/graalvm/sulong and shipped as part of the GraalVM at http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html.
The main drawback of Safe Sulong is that in addition to the applications or methods in Java’s standard library. However, C/C++ projects contain elements such as inline assembly and compiler builtins, which resemble external calls after being compiled to LLVM IR. The call target, either an inline assembly snippet or a builtin, needs to be implemented by Sulong’s interpreter.

Managed Sulong. We are currently working on combining the advantages of Safe Sulong and Native Sulong, providing memory safety were possible, while retaining the ability to use native libraries. To this end, we are implementing Managed Sulong, which can handle both managed and native allocations. Additionally, we are experimenting with executing machine code on the Truffle platform, using either an x86 Truffle interpreter or by lifting machine code to LLVM IR, which could then be executed in a safe way.

3 EXECUTING LIBC ON SULONG

Most LLVM IR instructions can be mapped to similar Java operations or methods in Java’s standard library. However, C/C++ projects contain elements such as inline assembly and compiler builtins, which resemble external calls after being compiled to LLVM IR. The call target, either an inline assembly snippet or a compiler builtin, needs to be implemented by Sulong’s interpreter.

and by using raw pointers to memory as references. However, a drawback is that buffer overflows and other memory errors can corrupt memory content or crash the JVM.

Safe Sulong. Due to the seriousness of buffer overflows—they belong to the most dangerous software errors overall [1]—we have implemented Safe Sulong, which prioritizes safety over native compatibility [8, 12]. It allocates Java objects for allocations in the user program and detects buffer overflows through automatic checks of the underlying JVM, which then aborts execution. It can also detect other errors such as use-after-free errors, double-free errors, and accesses to non-existent variadic arguments and has found errors in open-source programs that state-of-the-art bug-finding tools failed to detect [12]. Additionally, we have shown that exposing run-time information tracked by Safe Sulong, such as object bounds or types, helps programmers to improve the robustness of their libraries [13]. The main drawback of Safe Sulong is that in addition to the application itself it also requires all libraries used by the application to be available as LLVM IR as well because it cannot pass Java objects to native libraries.

Acknowledgments

We want to thank Oracle Labs, which partly funded the authors from Johannes Kepler University Linz. We thank all Sulong, Truffle, and Graal contributors, the members of Oracle Labs, and the members of the Institute for System Software at the Johannes Kepler University Linz for their contributions. Finally, we thank Stefan Marr, Stephen Kell, Bram Adams, and David Leopoldseder for their assistance in analyzing the usage of inline assembly and/or GCC compiler builtins to help their implementation in Sulong.

References


https://www.musl-libc.org/


