Sandboxed Execution of C and Other Unsafe Languages on the Java Virtual Machine

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

ABSTRACT

Sulong is a system that tackles buffer overflows and other low-level errors in languages like C by automatically checking them and aborting execution if an error is detected. Supporting unstandardized elements such as inline assembly and compiler builtins is a challenge, which we have addressed by investigating their usage in a large number of open-source programs. Finally, we have devised an introspection mechanism, for which Sulong exposes metadata such as bounds, which library writers can use to increase the robustness of their libraries.

KEYWORDS

memory safety, low-level errors, Sulong

1 INTRODUCTION

C does not enforce type safety, which can result in critical low-level errors such as buffer overflows or use-after-free errors. Although there are numerous approaches to tackle these errors [12, 13, 16], they still frequently occur. My research tackles this issue by providing a safe execution environment for executing languages like C. The core idea of my approach is that the language semantics of unsafe languages can be mapped to Java semantics, allowing such programs to be executed in a sandboxed environment on the Java Virtual Machine (JVM). My research on and contributions to this topic are divided into three parts:

(1) the implementation of this safe execution environment (called Safe Sulong) [4, 8, 9];
(2) empirical studies on unstandardized constructs in C code (such as inline assembly and compiler builtins) to prioritize their implementation in Sulong and other systems [6],
(3) and the design of an introspection interface for library writers to enhance the robustness of their libraries [7].

2 SAFE EXECUTION OF C ON THE JVM

Safe Sulong maps C data structures to Java. For example, a C struct is represented as a Java object that has a field for each struct element. A pointer is represented as a Java object that contains a reference to its pointee as well as an offset field that is used for pointer arithmetic (see Figure 1). We implemented Safe Sulong as a layered system (see Figure 2), in which we compile the program and the libraries to LLVM IR, which is easier to process than C code, and which we obtain by using LLVM's C front end Clang [2]. Executing LLVM IR allows us to handle also other languages including C++ and Fortran. Safe Sulong's core is an LLVM IR interpreter based on the Truffle language implementation framework [5, 14]. Efficiency is a key requirement for users [12], which we achieve by using the highly-optimizing Graal just-in-time compiler [15] to produce machine code during execution, which executes similarly fast as code generated by static compilers for C.

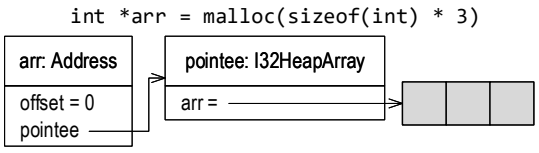

Figure 1: Example of pointer arithmetics and memory allocation (I32HeapArray is a subclass of I32Array).

Our approach does not require instrumentation to find bugs; instead, it relies on the underlying JVM. JVMs perform bounds checks to detect out-of-bounds accesses as well as type and null-pointer checks to guarantee type safety. Additionally, JVMs provide automatic memory management, so that programs cannot corrupt memory. These checks and mechanisms are automatic and well-specified. Relying on them when accessing the abstracted C data structures is more robust than instrumentation-based checks [1, 3, 11] on a conceptual level, because checks cannot simply be missed. Indeed, we found bugs in open-source software that were overlooked by other approaches [9].

3 EMPIRICAL STUDIES ON C PROGRAMS

C code (and LLVM IR) consists of unstandardized elements such as inline assembly and compiler builtins. Implementing them in Sulong, by expressing their semantics in Java, is a large implementation effort, which we prioritized by empirically analyzing the usage of x86-64 inline assembly [6]

*Advisor: Hanspeter Mössenböck, hanspeter.moessenboeck@jku.at, Johannes Kepler University Linz, Austria

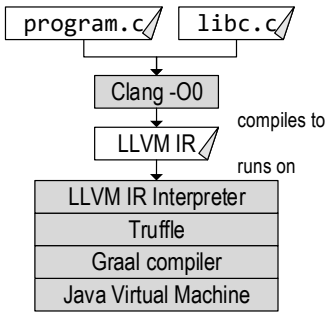

Figure 2: Overview of Safe Sulong.

and GCC compiler builtins. We downloaded a large number of GitHub projects and processed them to answer research questions relevant for tool developers (e.g., how many x86-64 inline assembly instructions need to be implemented to support most C projects?). Our findings suggest that both inline assembly and compiler builtins are widespread, but that most projects rely only on a small common subset (see Table 1 and Table 2). Thus, a large number of projects can be supported by implementing a few selected builtins. We believe that our findings are useful also for other tools that process C code.

4 EXPOSING METADATA TO LIBRARIES

Sulong's Java data structures have metadata attached such as bounds information or object types. We devised an introspection interface where Sulong exposes this metadata to library writers who can use it to increase the robustness of libraries [7]. For example, the `size_right()` function expects a pointer and returns the amount of allocated bytes right to the pointed address. In the spirit of failure-oblivious computing [10] this metadata can be used, for example, to make libc functions that process strings robust against missing `NUL` terminators by explicitly checking for the end of the buffer:

```
size_t strlen(const char *str) {
  size_t len = 0;
  while (size_right(str) > 0 && *str != '\0') {
    len++; str++;
  }
  return len;
}
```

We implemented parts of the introspection interface also in other bug-finding tools such as LLVM's AddressSanitizer, SoftBound, and Intel MPX-based bounds instrumentation.

5 CONCLUSION

My research is about safely executing programs written in languages like C. To this end, I have been working on Safe Sulong, which provides a sandboxed execution environment on the JVM. As part of this effort, I have been analyzing unstandardized elements in C (such as inline assembly) and devised an introspection interface that library writers can use to increase the robustness of their libraries.

Table 1: The three most frequent x86-64 inline assembly instructions in C projects

instruction	% projects
rdtsc	27.4%
cpuid	25.4%
mov	24.9%

Table 2: The three most frequent GCC builtins in C projects

instruction	% projects
__builtin_expect	48.2%
__builtin_clz	29.3%
__builtin_bswap32	26.2%

REFERENCES

[1] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 213–223.

[2] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO 2004.* 75–86.

[3] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.

[4] Manuel Rigger. 2016. Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. In *ECOOP 2016 Doctoral Symposium.*

[5] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016).* ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[6] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. [n. d.]. An Analysis of x86-64 Inline Assembly in C Programs. In *Virtual Execution Environments (VEE 2018).* https://doi.org/10.1145/3186411.3186418

[7] Manuel Rigger, Rene Mayrhofer, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Introspection for C and its Applications to Library Robustness. *Programming Journal* 2, 2 (2018), 4. https://doi.org/10.22152/programming-journal.org/2018/2/4

[8] Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine: Or: How I Learned to Stop Worrying and Run the Code. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017).* ACM, New York, NY, USA, 35–47. https://doi.org/10.1145/3132190.3132204

[9] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18).* Williamsburg, VA, USA. https://doi.org/10.1145/3173162.3173174

[10] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. 2004. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings*

of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04). USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251254.1251275

[11] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker.. In *USENIX Annual Technical Conference*. 309–318.

[12] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of SP '13*. 48–62.

[13] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses (RAID'12)*. 86–106. https://doi.org/10.1007/978-3-642-33338-5_5

[14] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. 13–14. https://doi.org/10.1145/2384716.2384723

[15] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[16] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime Countermeasures for Code Injection Attacks Against C and C++ Programs. *ACM Comput. Surv.* 44, 3, Article 17 (June 2012), 28 pages. https://doi.org/10.1145/2187671.2187679