Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages

Manuel Rigger[1]

1 **Johannes Kepler University Linz**
Altenberger Straße 69, 4040 Linz, Austria
`manuel.rigger@jku.at`

Abstract

Memory errors in C/C++ can allow an attacker to read sensitive data, corrupt the memory, or crash the executing process. The renowned top 25 of most dangerous software errors as published by the SANS Institute, as well as recent security disasters such as Heartbleed show how important it is to tackle memory safety for C/C++. We present Sulong, an efficient interpreter for LLVM-based languages that runs on the JVM. Sulong guarantees memory safety for C/C++ and other LLVM-based languages by using managed allocations and automatic memory management. Through dynamic compilation, Sulong will achieve peak performance close to state of the art compilers such as GCC or Clang, which do not produce memory-safe code. By efficiently implementing memory safety, Sulong strives to be a real-world solution for mitigating software security problems.

1998 ACM Subject Classification D.4.6 Security and Protection; D.3.4 Processors: Interpreters, Run-time environments

Keywords and phrases Memory Safety, LLVM, Java Virtual Machine, Dynamic Compilation

1 Introduction

Software written in unmanaged languages such as C and C++ is omnipresent. In these languages, invalid memory accesses, manual memory management errors, and undefined behavior can crash the process, or silently corrupt the memory or computations. The CWE/SANS Top 25 list [18] demonstrates that C/C++ security bugs are of high practical relevance. Among the 25 most dangerous software errors, buffer overflow is on the third place, directly after SQL and OS command injection. Memory errors are the most dangerous errors in C/C++ since such errors can be exploited to read and overwrite arbitrary memory [30]. Due to its importance, there are countless approaches that try to solve memory errors in C/C++. However, they are either not complete or not efficient enough to be used in real world applications [24].

In this paper we present Sulong, an ecosystem that strives to solve these issues. By executing LLVM IR, Sulong can execute all LLVM-based languages including C/C++. Sulong provides spatial memory safety by keeping all data as managed Java objects instead of using unmanaged memory. Thus, Sulong prevents spatial memory errors, which include out-of-bounds accesses to allocated memory (e.g., buffer overflows), dereferencing null pointers, and dereferencing "crafted pointers" such as obtained by casting integer values to addresses. By using managed allocations, Sulong also provides temporal memory safety and prevents the program from dereferencing memory which it already deallocated. If Sulong encounters spatial or temporal memory errors, it displays an error message and exits the program.

Since efficiency is a big concern, Sulong uses dynamic compilation to compile frequently executed LLVM IR functions to optimized machine code. In its final version, we expect

Sulong to execute code with the same peak performance as code that was generated by optimizing state of the art compilers, which produce unsafe code. By translating native libraries to LLVM IR, Sulong will also be able to support precompiled libraries. Thus, Sulong strives to be a real world solution for memory safety of C/C++ programs.

2 Problem Setting

According to an IEEE Spectrum ranking, the three most popular programming languages are Java, C, and C++ [1]. In contrast to Java, C and C++ do not enforce type safety, are not memory safe, and have many operations that can result in undefined behavior [26]. Due to the high prevalence of C/C++ and the large body of legacy code written in these languages, the unsafeness of these two languages is a threat to system security.

Memory errors are the most notorious type of errors that can occur in C/C++. The CWE/SANS list of the most dangerous software errors [18] ranks stack overflows as one of the most harmful errors. The lack of bounds checks, type checks, and manual memory management allow attackers to exploit stack overflows (and other memory errors) by injecting and executing arbitrary code or reading sensitive data [5]. Therefore, memory error exploits are omnipresent in exploit packs [25].

We distinguish between temporal and spatial memory errors: Spatial memory errors occur through out-of-bounds accesses, i.e., when a pointer is dereferenced that points outside of allocated memory, or is null. Spatial memory errors can manifest themselves as stack-based buffer overflows, heap-based buffer overflows, null pointer dereferences, and as format string vulnerabilities. Temporal memory errors happen when a dangling pointer, i.e., a pointer that points to memory that has already been freed, is again freed or dereferenced. Complete memory safety is only guaranteed when no temporal and spatial memory errors can occur [24].

2.1 Problem Statement

For decades, both academia and industry have been trying to come up with countless static and run-time countermeasures, as well as with hardware- and software-based approaches to prevent memory errors. As for software-based run-time countermeasures, approaches can roughly be classified into Address Space Layout Randomization (ASLR), canaries approaches, data execution prevention, data space randomization, and bounds checkers. Literature already extensively covers these approaches and provides an historical overview of memory errors and defense mechanisms [25], an investigation of the weaknesses of current memory defense mechanisms and a general model for memory attacks [24], and a survey of vulnerabilities and run-time countermeasures [30]. Despite the efforts, memory safety is still an unsolved issue in practice [25], since existing approaches have one or more weaknesses with respect to the following properties:

- *Complete Memory Safety*: The strength of a policy characterizes whether an approach provides full or partial memory safety [24]. We believe that a policy that implements complete memory safety is essential, since even near-to-complete memory safety approaches will still result in hard-to-debug errors, and since attackers have always found new ways of exploitation which had previously been deemed secure [25].
- *Precompiled Library Interoperability*: Sometimes, source code of shared libraries is closed source or the source code of a library is no longer available. Being still able to execute such libraries by guaranteeing binary compatibility can be a requirement for a memory

safety approach [24]. We believe that supporting such libraries is important, but executing them should not compromise complete memory safety.

- *Run-time Performance*: The performance overhead of a memory safety approach has to be kept minimal, since otherwise the approach will not be adapted in practice. An analysis showed that techniques that introduce overheads of more than roughly 10% do not tend to gain wide adoption [24].

2.2 State of the Art

Existing memory approaches do not fulfill all of the above requirements, namely complete memory safety, interoperability with precompiled libraries, and run-time performance. Most approaches adopted in practice have low overheads and support precompiled libraries but prevent only a subset of spatial memory errors [24]. For example, stack canaries [6, 5] and a non-executable stack can prevent stack overflows and the execution of data on the stack. There are also approaches for heap overflows, and approaches such as DEP/W^X that restrict executing heap data. Address Layout Space Randomization [19] makes it difficult to perform return-to-libc attacks, as long as no relevant pointers are exposed.

The only way to enforce complete spatial memory safety is to keep track of pointer bounds. Bounds checkers [30, 15, 24] implement such an approach, but have overheads of 67% [15] and more. Literature discriminates approaches that store bounds information as separate metadata or add them to pointers (fat-pointer approaches) or objects. Before accessing an object, these approaches look up the bounds information and check that an access is within bounds. These approaches only provide limited interoperability with unprotected precompiled libraries, since those libraries do not provide or maintain bounds information.

Complete memory safety approaches have overheads that vastly exceed the 10% required for wide adoption. A complete memory safety approach not only needs to keep track of pointers, but also needs to maintain allocation information. For example, SoftBound with CETS [15, 16] is a recent complete memory safety approach that combines bounds checking with checks if an object is still allocated. It has an average overhead of 116% [16].

3 Approach

In this paper we present Sulong, an ecosystem (see Figure 1) that executes LLVM-based languages on the JVM. Sulong provides complete memory safety, supports precompiled libraries, and will have a peak performance that is competitive with state of the art compilers. Sulong executes LLVM IR, which is part of the LLVM static compilation framework [14] and is produced by LLVM front ends (e.g. Clang for C/C++) that compile source languages to this IR. Executing LLVM IR allows Sulong to support many different languages such as C/C++, Fortran, and Objective-C. Instead of statically compiling the LLVM IR to machine code, Sulong interprets it using a Truffle [29] AST interpreter. To build executable ASTs, Sulong's parser front end parses LLVM IR files and instantiates the AST node classes that implement the LLVM IR operations. Sulong reaches excellent peak performance by dynamically compiling these ASTs to machine code.

3.1 Complete Memory Safety

To provide complete memory safety in Sulong, we want to use managed Java memory allocations instead of unmanaged native allocations. Thus, we must implement all memory allocation mechanisms (stack and heap) by allocating Java objects.

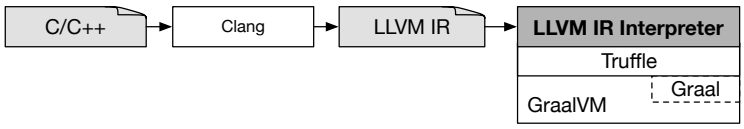

Figure 1 Sulong System Overview

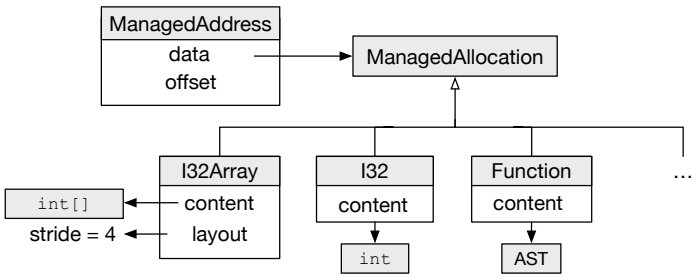

Figure 2 `ManagedAllocation`s and `ManagedAddresses` instead of unmanaged memory and machine addresses

For stack allocations, LLVM IR has a special instruction `alloca`. We simply implement the execution of this instruction by instantiating a Java object of the specified type. For heap allocations, LLVM IR does not provide an instruction. Instead, LLVM IR programs use external calls to allocation functions (e.g. `malloc` in C) implemented in a standard library. Usually, a standard library is dynamically linked to the executable. For Sulong, we compile the given standard library to LLVM IR and then link it with the program. Every function of a standard library can only call other standard library functions or use system calls. Since a standard library eventually has to use known system calls for memory allocations (such as `brk` or `mmap` of the Linux Kernel API) we substitute the calls to such kernel memory allocation functions with methods that allocate managed Java objects. Using this approach, we can provide memory safety for both the application and the standard library in any LLVM-based language. For memory safety on a finer granularity we additionally plan to substitute some allocation functions (e.g., malloc) with Java equivalents.

To guarantee spatial and temporal safety with Java objects we base our approach on the work of ManagedC [13]. For every stack and heap allocation Sulong returns an instance of class `ManagedAllocation` with different subclasses for primitives, functions, arrays, and other objects (see Figure 2). Every `ManagedAllocation` subclass contains a `content` field with a reference to the actual data, e.g., a Java integer array for a LLVM `I32` array. The different subclasses have additional fields to provide information for the access, e.g., arrays and structures have an additional layout table as explained below. When a programmer wants to reclaim memory, e.g., with a `free` call of the C standard library that would eventually invoke a system call, we instead set the `content` field to null. The Java garbage collector can then collect the object previously referenced by the `content` field. Even if the programmer forgets to deallocate an object, the garbage collector eventually collects it if there are no references to it. If, however, the program tries to free a `ManagedAllocation` twice, or dereference a freed object, Sulong recognizes null in the `content` field and reports an error. Refusing execution and exiting the program after such an error guarantees temporal memory safety.

To support pointers and pointer arithmetics, Sulong implements managed pointers via a `ManagedAddress` class. A `ManagedAddress` contains a reference to a `ManagedAllocation`,

as well as an offset. The offset is relative to the referenced object, i.e., it can specify the offset inside a compound object (array, struct, or vector). Pointer arithmetics use this `ManagedAddress` to perform pointer calculations. For example, a pointer increment will create a new `ManagedAddress` by adding the size of the referenced data type to the offset of the given managed pointer. When dereferencing a pointer, Sulong uses the reference to the `ManagedAllocation` and the given offset to dereference the object. For compound objects Sulong needs to map the offset to an object, e.g, for an I32 array it maps the offset 8 (twice the size of an int) to an index 2 of a Java integer array. `ManagedAllocation`s that represent compound objects thus provide layout information to map an offset to a member (and a type for structs) of the `content` field. If an access is out-of-bounds, the JVM automatically prevents the invalid access and thus guarantees spatial memory safety. Trivially, this approach can also prevent null pointer references which have a null value in the `ManagedAddress` field.

3.2 Eliminating Overheads by Dynamic Compilation

Our approach has potential performance overheads compared to the execution of unsafe code generated by static compilation, since we use managed allocations that require additional checks for accesses. For example, for an access to an array element in an LLVM IR program, the compiler has to insert guards to ensure that the `ManagedAllocation` and its `content` field are not `null`, that the `ManagedAllocation` type is of the expected subclass, and that the access is in-bounds. If such a guard fails the compiled code is discarded and execution is continued in the interpreter. While bounds checks alone proved to be prohibitively expensive in static compilation [25], dynamic compilers can optimize or even eliminate object allocations as well as type checks and bounds checks based on run-time feedback. Our dynamic compiler Graal benefits from profiling information collected by the underlying JVM which includes branch probabilities, type profiles, exception probabilities, and method invocation counters [22].

We are confident that managed allocations do not impede peak performance because previous work on Truffle/C and ManagedC [11, 13] demonstrated that using Java allocations instead of native memory when executing C code on Truffle does not impede peak performance on average. Using Java allocations is even faster in some cases, since the dynamic compiler can make better assumptions about aliasing than when using unmanaged native memory.

For Sulong, Graal can minimize the overhead of managed memory, since escape analysis in combination with scalar replacement [23] can often eliminate allocations and instead use local variables or allocate objects on the stack. For heap-allocated objects, typed-checked inlining [22] can speculatively inline calls by inserting a guard before the call site, which can reduce the overhead when calling methods of the `ManagedAllocation` subclasses. Array bounds check elimination [28, 27] can identify situations where bounds checks are redundant and can fully remove bounds checks that it can prove to never fail. If it cannot prove full redundancy, the optimization can often still move checks out of loops where they are likely to be executed less often than inside the loop. Graal can also apply code motion [3, 8] to move other loop-invariant code out of a loop. Finally, conditional elimination [22] can remove conditional expressions where conditions can be proven to be true or false, which can remove or simplify type checks within an if-statement that checks for the same type.

3.3 Support of Precompiled Libraries

As initially stated, lacking interoperability with precompiled libraries can prevent the adoption of a memory safety approach. Sulong provides complete memory safety as long as the complete program is available as LLVM IR. Thus, we also translate precompiled libraries to LLVM IR. We explained how we support the standard libraries by compiling them to LLVM IR and substituting the system calls for memory allocations with Java equivalents.

For other system calls (e.g., file descriptor operations), we can use the Graal native function interface [12] which allows us to directly invoke native functions from Java. Using the Graal Native Function Interface (Graal NFI) is generally unsafe, since invalid memory accesses could go unnoticed in native functions. However, we decided to trust system calls since they are implemented as part of the OS, and since the system call interface has been designed to be robust against usage errors.

For precompiled libraries where the source code is not available, using the Graal NFI would break complete memory safety since a precompiled library could contain an exploitable memory error, and since we cannot assume any bounds information for pointers passed to or returned by the library. Therefore, we want to follow a binary translation approach [21] to support precompiled libraries. A binary translation approach translates a source instruction set to a target instruction set which is LLVM IR in our case. Several tools exist that translate code of a source instruction set to LLVM IR including QEMU [2], MC-Semantics for X86 [7], and LLBT for ARM [20]. Projects such as Apple's Rosetta which translated PowerPC applications to Intel X86 binaries demonstrated that such an approach is feasible in practice. By translating the machine code of the precompiled library to LLVM IR, we can execute it and still provide memory safety for it. We will evaluate if such an approach can be used to convert large libraries to LLVM IR, for which we could again provide complete memory safety.

4 Evaluation Strategy

We plan to evaluate Sulong with respect to safety and performance and want to show that Sulong is a sound and efficient approach for getting rid of memory errors.

Our first hypothesis is that our approach is memory safe and detects memory errors during run-time. For an empirical evaluation we will use NIST's Juliet test suite for C/C++ which contains 61,387 synthetic test cases with examples for 118 Common Weakness Enumeration categories (CWEs) [9]. Juliet contains test cases for memory bug CWEs such as buffer overflows and underflows, uses after free, invalid frees and double frees. We expect to detect all memory errors that are exhibited during run-time. Apart from synthetic test cases we also want to evaluate memory safety by testing our approach on large C or C++ projects. Using the NIST National Vulnerability Database [17] we will identify programs with exploitable memory bugs, and will then execute these programs with an attack vector using Sulong. By demonstrating that Sulong can detect real world security bugs we will show that Sulong is a sound approach to safely execute real world programs.

Our second hypothesis is that our memory safety approach does not incur significant overheads and that is competitive with optimizing static compilers. To argue that the memory safety does not incur substantial overheads, we will provide a Sulong version, subsequently referred to as Sulong$_{UNSAFE}$, that uses the same memory model as native applications. Sulong$_{UNSAFE}$ will implement the native memory model by using the Java Unsafe API through which Sulong allocates, deallocates, and accesses raw memory instead of managed memory. We will then show that Sulong executes programs with the same

peak performance as Sulong$_{UNSAFE}$ and thus demonstrate, that there is no abstraction overhead when using Java memory allocations. To argue that Sulong and Sulong$_{UNSAFE}$ are competitive with the execution speed of code compiled by state of the art optimizing compilers, we will compare them with the performance of optimized code generated by Clang and GCC. By reaching a comparable performance we will show that Sulong is efficient enough to be used as a safer replacement for static compilers. In order to evaluate the performance of Sulong, we want to evaluate typical C/C++ benchmarks. *The Computer Language Benchmark Game* [10] provides small benchmark programs that typically consist of up to a few hundred lines of code. Additionally, we want to evaluate Sulong's performance on the SPEC benchmarks [4]. The SPEC benchmarks are suitable for evaluating memory safety approaches, since they are mainly CPU bound and suitable to measure the overhead of memory accesses [24].

5 Conclusion

This paper presented the design of Sulong, a memory safe execution environment for C, C++, and other LLVM-based languages which we currently develop. Sulong strives to provide complete memory safety, support precompiled libraries, and reach the same peak performance as code that was generated by state of the art static compilers which produce unsafe code. Sulong guarantees temporal and spatial memory safety by using managed Java allocations instead of unmanaged native allocations. By using a binary translation approach, Sulong translates precompiled libraries to LLVM IR which it can safely execute then. Sulong uses a dynamic compilation approach, through which it minimizes the overhead of bounds checks and type checks. We want to proof the effectiveness of our approach by evaluating the peak performance for benchmarks such as SPEC. We also want to demonstrate Sulong's security claim by applying it to the NIST Juliet suite and to case studies on real world programs.

References

1 Stephen Cass. The 2015 top ten programming languages. 2015.

2 Vitaly Chipounov and George Candea. Dynamically translating x86 to llvm using qemu. Technical report, École polytechnique fédérale de Lausanne, 2010.

3 Cliff Click. Global code motion/global value numbering. In *ACM SIGPLAN Notices*, volume 30, pages 246–257, 1995.

4 Standard Performance Evaluation Corporation. Cint2006 (integer component of spec cpu2006), 2006.

5 Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DISCEX'00. Proceedings*, volume 2, pages 119–129, 2000.

6 Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of ASIACCS 2015*, pages 555–566, 2015.

7 ARTEM DINABURG and ANDREW RUEF. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

8 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of VMIL '13*, pages 1–10, 2013.

9 Center for Assured Software. Juliet test suite v1.2 for c/c++. Technical report, National Security Agency, 2012.

10 The Computer Language Benchmarks Game. The computer language benchmarks game.

11 Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. Trufflec: Dynamic execution of c on a java virtual machine. PPPJ '14, pages 17–26, 2014.

12 Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for java. PPPJ '13, pages 35–44, 2013.

13 Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Memory-safe execution of c on a java vm. PLAS'15, pages 16–27, 2015.

14 Chris Lattner and Vikram Adve. The llvm instruction set and compilation strategy. *CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS*, 2002.

15 Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. PLDI '09, pages 245–258, 2009.

16 Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40, 2010.

17 NIST. National vulnerability database, 2016.

18 SANS. Cwe/sans top 25 most dangerous software errors, 2011.

19 Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the CCS '04*, pages 298–307, 2004.

20 Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. Llbt: An llvm-based static binary translator. CASES '12, pages 51–60, 2012.

21 Richard L Sites, Anton Chernoff, Matthew B Kirk, Maurice P Marks, and Scott G Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.

22 Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala*, page 9, 2013.

23 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. CGO '14, pages 165:165–165:174, 2014.

24 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dong Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62, 2013.

25 Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. RAID'12, pages 86–106, 2012.

26 Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of APSys '12*, page 9, 2012.

27 Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination for the java hotspot™ client compiler. In *Proceedings of PPPJ '07*, pages 125–133. ACM, 2007.

28 Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5):279–295, 2009.

29 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. Onward! 2013, pages 187–204, 2013.

30 Yves Younan, Wouter Joosen, and Frank Piessens. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv.*, 44(3):17:1–17:28, June 2012.