Sulong, and Thanks For All the Bugs

Finding Errors in C Programs by Abstracting from the Native Execution Model

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

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

René Mayrhofer
Johannes Kepler University Linz
Austria
rene.mayrhofer@jku.at

Matthias Grimmer
Oracle Labs
Austria
matthias.grimmer@oracle.com

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

Abstract

In C, memory errors, such as buffer overflows, are among the most dangerous software errors; as we show, they are still on the rise. Current dynamic bug-finding tools that try to detect such errors are based on the low-level execution model of the underlying machine. They insert additional checks in an ad-hoc fashion, which makes them prone to omitting checks for corner cases. To address this, we devised a novel approach to finding bugs during the execution of a program. At the core of this approach is an interpreter written in a high-level language that performs automatic checks (such as bounds, `NULL`, and type checks). By mapping data structures in C to those of the high-level language, accesses are automatically checked and bugs discovered. We have implemented this approach and show that our tool (called *Safe Sulong*) can find bugs that state-of-the-art tools overlook, such as out-of-bounds accesses to the main function arguments.

CCS Concepts • **Software and its engineering** → **Dynamic compilers**; **Runtime environments**; *Interpreters*; *Software testing and debugging*; • **Security and privacy** → *Virtualization and security*;

Keywords Sulong; memory errors; bug detection; C

ACM Reference Format:
Manuel Rigger, Roland Schatz, René 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 *ASPLOS '18: Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3173162.3173174

1 Introduction

C programs are plagued by bugs. In particular, memory errors such as buffer overflows, `NULL` pointer dereferences, and use-after-free errors cause critical bugs. Unlike higher-level languages, the C standard does not define any checks that could detect such erroneous accesses and then abort the program. If not prevented in the application logic, errors induce *undefined behavior*; in practice, they can corrupt memory, leak sensitive data, change the control flow, or crash the program. In some cases, errors remain undetected because they can cause delayed failures or do not exhibit any visible symptoms. Memory errors in C therefore often result in hard-to-find bugs or enable exploitation by attackers.

To tackle this issue, industry and academia have come up with a plethora of static and dynamic tools for finding bugs in C programs [62, 63, 73]. Static tools perform analyses of source code to detect errors of specific types; they typically rely on necessarily incomplete heuristics and give rise to both false positives and false negatives [11, 17, 27]. In contrast, dynamic tools insert additional checks either as part of the compilation process or at run time, and find errors during program execution. Although they only find errors that occur during a specific run of the program, they are expected to find all errors and not to produce false positives. Both static and dynamic bug-finding tools have been widely successful and have detected numerous bugs in commonly used libraries.

In this paper, we concentrate on dynamic bug-finding tools and demonstrate that state-of-the-art approaches such as LLVM's AddressSanitizer (ASan) [55] and Valgrind [42] miss real-world errors that programmers would expect to be found. We argue that this is due to current approaches not abstracting from the underlying machine's low-level execution model; the lack of source information makes it difficult to find all bugs, and a check can easily be forgotten.

Furthermore, the checks are implemented using inexact techniques, which inherently causes these tools to miss errors. Dynamic bug-finding tools are either based on static compilers or employed after compilation. It is known that compiler optimizations at higher optimization levels interfere with bug-finding tools [64]; we show that compilers can also optimize away memory errors even when explicitly compiling without optimizations (i.e., with the `-O0` flag). Finally, bug-finding tools that support interoperability with native code usually provide restricted bug-finding coverage, which gives users a false sense of security. For example, both ASan and Valgrind cannot detect out-of-bounds accesses to the `main()` function's arguments.

In this paper, we present a novel approach to finding bugs at run time and to addressing these issues. We implemented this approach in a tool called *Safe Sulong*, which can detect out-of-bounds accesses, use-after-free errors, invalid free errors, double free errors, `NULL` dereferences, and accesses to non-existent variadic arguments. Our approach abstracts from the underlying machine's execution model, using an execution environment for C that is written in a high-level language. By abstracting pointers and other C data structures and representing them in the high-level language, we can rely on well-defined automatic checks of the high-level language to detect bugs in a C program. While we used Java for our implementation, the approach also works for other languages that check and disallow buffer overflows and `NULL` pointer dereferences. Our approach is exact (i.e., non-heuristic) and can find all errors of a specific category. To reach native speeds, it uses a dynamic compiler that compiles frequently executed functions to machine code. This compiler does not optimize away bugs, since it optimizes code based on safe semantics in the sense of Felleisen & Krishnamurthi [18], where run-time errors in the program must cause run-time exceptions. We do not provide interoperability with pre-compiled native code, because it would undermine our bug-finding capabilities. We assume that all C code (including libraries) is executed with our tool, which makes our approach impractical for programs that use libraries for which no source code is available.

In our evaluation we tried to find bugs in small open-source projects. We detected and fixed 68 errors, 8 of which were not found by ASan and Valgrind. We argue that these bugs are due to the lack of abstraction of current-bug finding tools. Additionally, we conducted a preliminary performance evaluation; our prototype lacks functionality to execute large benchmarks such as the ones of SPEC [23] and browsers. The evaluation demonstrates that Safe Sulong has a higher warm-up cost than current approaches, but a peak performance that is better than of other bug-finding tools.

Overall, this paper provides the following contributions:

- We present an alternative approach to bug-finding that abstracts from the underlying machine.
- We implemented our approach and evaluated its start-up costs, warm-up costs, and peak performance.

2 Background

2.1 Errors in C

To determine which memory errors are relevant in practice and should therefore be detected by bug-finding tools, we performed keyword searches of the Common Vulnerabilities and Exposures (CVE)[1] and the ExploitDB[2] databases. Unlike a previous study of memory errors (up to 2012) [63], we grouped the errors into different bug categories. Note that we concentrated only on memory errors (i.e., dereferencing invalid pointers) and thus did not consider memory leaks, reading from uninitialized memory, and other C errors.[3] Figures 1 and 2 show the results for the period from 2012 to 2017. Note that bug categories with a high number of vulnerabilities were also exploited more often.

Out-of-bounds accesses. The most common and dangerous bug category (as previously shown [9, 54, 63]) consists of out-of-bounds accesses to objects, which are also known as spatial memory safety errors. Not only do such bugs continue to be relevant, they are currently on an all-time high. We define an out-of-bounds access as a buffer overflow when it attempts to access memory past the end of an object, and as a buffer underflow when it accesses memory before the beginning of an object.[4] Bug-finding tools typically differ in whether they can detect out-of-bounds accesses to the stack, heap, or global (static) data and whether they detect read and/or write accesses. For example, Valgrind can only find heap buffer out-of-bounds accesses.

Use-after-free errors. The second-most common bug category comprises use-after-free errors (known as temporal memory errors), where an object allocated by `malloc()`, `calloc()`, or `realloc()` is freed, but then accessed again. Such an access is also known as an invalid access to a *stale* or *dangling pointer.*

NULL dereferences. The third-most important bug category is a `NULL` dereference. Note that this error can be detected during normal execution of a program, where dereferencing a `NULL` pointer results in a trap on most architectures.

Other errors. Since the remaining memory errors are less common, we classified invalid free errors, double free errors, and accesses to non-existent variadic arguments as "other errors". An invalid free error is caused when a pointer to a stack object or to a global object is passed to `free()`, or when the pointer passed points into the middle of an object. Double free errors occur when a heap object is freed twice. Accesses

[1] https://cve.mitre.org/

[2] https://www.exploit-db.com/

[3] We are currently adding support for finding such bugs in Safe Sulong (see Section 6) and will describe them in a future paper.

[4] We do not consider out-of-bound accesses in sub objects (e.g., from one array field member to another), as they are deliberately used in `memcpy`-like patterns.

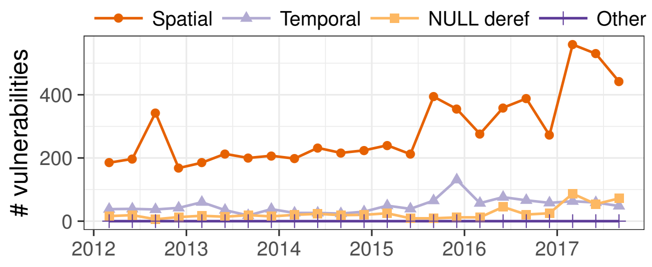

Figure 1. Number of reported vulnerabilities in the CVE database (2012-03 to 2017-09).

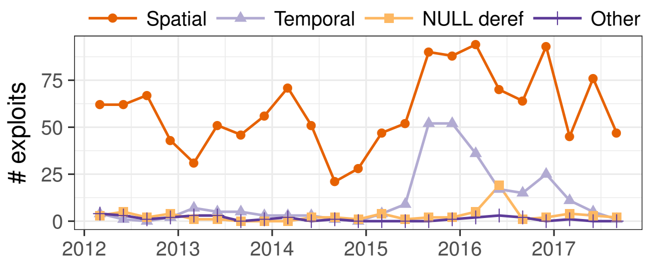

Figure 2. Number of available exploits in the ExploitDB (2012-03 to 2017-09).

to non-existent variadic arguments happen when the number of passed variadic arguments is smaller than that expected by the function. One subclass of this error type are format-string vulnerabilities, where the format string specifies how many arguments on the stack should be accessed.

2.2 State of the Art

Shadow memory. Most practical bug-finding tools such as ASan [55], Mudflap [16], Valgrind [42], Dr. Memory [4], SoftBound+CETS [39, 40], and Purify [22] base their bug-finding capabilities on the concept of shadow memory: They maintain metadata about application memory in a separate memory area referred to as shadow memory, which is used to verify specific actions; for example, read accesses validate that a memory cell is accessible (i.e., allocated memory). Shadow-memory tools are typically combined with red-zone approaches: when a program allocates memory, the runtime of the tool marks the shadow-memory area associated with the program memory as accessible and a region around it as inaccessible (called a *redzone*). Most shadow-memory-based bug-finding tools use this technique to detect out-of-bounds accesses, use-after-free errors, double free errors, invalid free errors and `NULL` dereferences. Some tools also detect reads of uninitialized memory, use-after-scope errors, and memory leaks. We further discriminate between shadow-memory tools based on whether the instrumentation is added at compile or run time.

Compile-time instrumentation. Compile-time instrumentation involves inserting code for tracking allocations and inserting additional checks when (or before) the program is compiled. The most widely used compile-time instrumentation approach is LLVM's AddressSanitizer, which initially detected out-of-bounds accesses, use-after-free errors, and `NULL` dereferences [55] and has been extended to detect invalid free, double free, and use-after-scope (including use-after-return as a special case) errors as well as memory leaks. Another state-of-the-art tool that is less used in practice is SoftBound+CETS [39, 40], a bounds checker with a temporal memory safety tool. Mudflap [16] was used by the GCC project until GCC 4.9, when it was superseded by AddressSanitizer. It was known to have several shortcomings, such as reporting false positives and not detecting buffer overflows for neighboring objects in the memory [67]. Commercial tools include Purify [22], which is not strictly a compile-time approach, since it inserts code into object files, and Insure++ [44].

Dynamic instrumentation. Dynamic instrumentation involves inserting checks at the binary level during program execution. The advantages of dynamic instrumentation are that it works for any language that is compiled to machine code, that all code is checked even if the source code and meta information (such as debug information) are not available, and that it does not require recompilation [56]. The most widely used run-time instrumentation approach is Valgrind. Other dynamic instrumentation approaches include Dr. Memory [4] and Intel Inspector [28]. Note that binary instrumentation approaches cannot detect out-of-bounds accesses to the stack (unless the top of the stack is exceeded).

Other approaches. A plethora of other approaches tackle memory errors [62, 63, 73]. For example, *Polymorphic C* [59], *Cyclone* [29], and *CCured* [41] are well-known approaches that provide guarantees against memory errors. However, they require modification of the source code, and are thus not widely used. Canary-based approaches [35] inserts special values (called canaries) next to allocated memory to detect overflows, and inside freed objects to detect writes to freed memory. However, canary approaches can detect only invalid writes, as long as the canary value is not reset again.

2.3 Limitations of Current Approaches

Our main focus is on finding all memory errors in a C program. Current approaches to this have several limitations. We will provide examples for these limitations in our evaluation (see Section 4.1).

Problem 1 (P1): Lack of abstraction from the machine. Current bug-finding tools do not abstract from the low-level execution model of the underlying machine: instead of defining errors at the source level, they define them on the machine level. They insert additional checks either as part of a separate phase in an existing compiler (e.g., ASan or SoftBound) or directly into existing native code (e.g., Valgrind).

On this level, the loss of source information makes it conceptually challenging (or impossible) to find all bugs that existed on the source level. Additionally, some approaches need to instrument all read and write operations, all allocations and deallocations, and all system calls [42]. The additional complexity when compared to source-level approaches makes it easy to overlook bugs. A forgotten check cannot easily be found, since in many cases the program behaves as intended, with the only exception that specific errors have gone undetected.

Problem 2 (P2): Compiler optimizations. Current bug-finding tools are built on top of an optimizing compiler such as Clang or GCC. As previously noted [64], this is an issue for bug-finding tools, since they implement C semantics that differ from those of the compiler's optimizer. For example, while bug-finding tools report errors for invalid accesses and abort the program, compilers assume undefined semantics for errors and sometimes optimize them away. It has been shown that compilers are increasingly taking advantage of undefined semantics to optimize code, which leads to more vulnerabilities [14, 65].

Compiler optimizations can lead to *false positives*. For example, a false positive that was found in an ASan-instrumented Firefox build was caused by load-widening [55] where a series of loads is transformed into a single load of several memory values at once while potentially exceeding the bounds of an object. Due to platform-specific alignment requirements, such an optimization can be correct at the system level; however, ASan classified it as a bug because the access would be out of bounds in C. While this issue has been fixed by disabling load-widening [55], such compiler optimizations can still cause false positives in dynamic-instrumentation bug-finding tools (such as Valgrind [56]).

A more serious problem is that compiler optimization can lead to missed errors, that is, *false negatives*. It is widely known that at high optimization levels (e.g., with the `-O2` flag), compilers optimize the code based on the fact that error semantics are undefined [33, 45]. For example, consider the (contrived) function in Figure 3. The function initializes elements of an array without using it further. The array accesses have no visible side effects, so the compiler optimizes the function to immediately return 0. The compiler can exploit the fact that an out-of-bounds access (when $length \geq 10$) has undefined error semantics. Consequently, out-of-bounds accesses that would have occurred in the original code might stay undetected at the binary level, and current bug-finding approaches are unable to find them. It has also been shown that compilers can remove redundant null-pointer checks, even at `-O0` [65]. Further, we have found that Clang can optimize away memory safety errors at `-O0`.

Since the compiler can optimize away bugs (or cause false positives), many projects decide to disable optimizations altogether (with the `-O0` flag) when testing and accept performance degradations. However, as we will demonstrate,

```c
int test(size_t length) {
    int arr[10] = {0};
    for (size_t i = 0; i < length; i++) {
        arr[i] = i;
    }
    return 0;
}
```

Figure 3. A C program with a potential out-of-bounds access is reduced to `return 0` by optimizing compilers.

explicitly disabling optimizations does not stop compilers from optimizing away bugs.

Problem 3 (P3): Inexact approaches. Tools based on shadow-memory red-zone approaches typically cannot detect all bugs of a particular category. First, they cannot detect all out-of-bounds accesses. An access to an object running out of bounds and landing inside a different object is not detected as a bug, because the check does not access the redzones next to the original object. Second, shadow-memory approaches cannot reliably detect use-after-free errors. When freeing an object, these approaches mark its shadow memory as unallocated. If the block is quickly reallocated, subsequent uses of the dangling pointer stay undetected, since the memory is again marked as valid. ASan [55] and Purify [22] rely on heuristics to avoid rapid reallocation of freed memory. Note that SoftBound+CETS [39, 40] is not susceptible to such false negatives.

Problem 4 (P4): Finding invalid accesses in libc. Supporting external libraries is a challenge for bug-finding tools. Run-time instrumentation approaches inherently support existing machine code. In contrast, compile-time instrumentation approaches that support native interoperability require heuristics or special treatment of native functions in order to maintain a correct state of the shadow memory.

To achieve higher coverage, compile-time instrumentation approaches recommend creating special instrumented builds for external libraries [55]. This is a challenge in relation to libc, where most production-quality implementations contain non-standard C code (or hand-written assembly) that causes most bug-finding tools (both run-time and compile-time instrumentation approaches) to report errors. Examples are optimized versions of `strlen()` that compute the length of a string by word-wise comparison [66], which can—like the load-widening optimization—lead to out-of-bounds accesses. Current compile-time instrumentation tools disable instrumentation or checks for such functions, or replace them altogether.

The pragmatic alternative that compile-time instrumentation approaches such as ASan and Mudflap provide are so-called *interceptors* that wrap the system library functions and call them only after performing validity checks on the arguments.

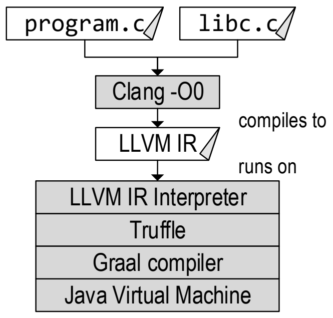

Figure 4. Overview of Safe Sulong.

This approach is dangerous when users expect these interceptors to be comprehensive. As we show in Section 4.1, we found bugs in real-world programs that were not detected by ASan due to a missing interceptor. Valgrind and Dr. Memory also provide replacements for these functions, which, however, do not work when these calls have already been inlined at compile time. Thus, Valgrind detects magic constants that point towards a `strlen()` implementation and disables checks for that code block [56].

3 Implementation

We developed Safe Sulong to address the four problems mentioned in Section 2.3. We designed our tool with a focus on bug-finding capabilities. Unlike state-of-the-art approaches that plug into compilers or into native code (see **P1**), Safe Sulong abstracts from the underlying machine and implements a simple execution model; it executes C programs using an interpreter written in Java that relies on automatic checks of the language. The interpreter uses an exact approach (to address **P3**), so no errors are missed. We do not provide interoperability with native code, since this could undermine the bug-finding capabilities (see **P3**). However, we provide our own libc that is written in standard C and is optimized for safety instead of performance (see **P4**). Unlike state-of-the-art approaches that rely on compilers that exploit undefined behavior for compiler optimizations (see **P2**), we use a dynamic compiler that optimizes the code based on safe semantics and cannot optimize away invalid accesses. In summary, our approach allows us to find errors in C programs reliably while still reaching a good peak performance.

3.1 System Overview

Figure 4 shows the architecture of Safe Sulong. It comprises the following components:

Libc. We argued that current libc implementations (which are optimized primarily for performance) are detrimental to bug-finding tools. To address this issue, we implemented a libc that is written in standard C and does not rely on any GNU extensions. It performs additional checks based on run-time information [52]. To implement libc, Safe Sulong exposes functions that are implemented in Java and serve the same purpose as system calls. For example, when printing a pointer value using `printf("%p")`, the `printf()` implementation calls a function implemented in Java to retrieve a textual representation of the pointer. Currently, we support 126 common libc functions, which is sufficient to execute a large body of programs. However, we still lack support for threads and synchronization, interprocess communication, many low-level operations (`mmap()`, `mprotect()`, `setjmp()` and `longjmp()`), and less commonly used functions. As part of future work, we intend to support running an existing libc that chooses standards-conformance and safety over performance (e.g., the `musl libc`) on Safe Sulong. This will require us, for example, to add support for the safe execution of inline assembly that libcs use to implement functionality such as system calls, atomics, and busy-waiting processor hints.

Clang and LLVM IR. Safe Sulong executes LLVM Intermediate Representation (IR), which represents C functions in a simpler, lower-level format. LLVM is a flexible compilation infrastructure [34], and we use LLVM's front end Clang to compile the source code (our libc and the user application) to the IR. Note that we do not enable any of Clang's optimizations to lower the risk that bugs are optimized away. As part of future work, we will replace Clang with a non-optimizing front end, to eliminate this risk completely (see Section 6). Since LLVM IR retains all C characteristics that are important in our context, for simplicity we hereafter refer to LLVM IR objects as C objects. By executing LLVM IR, Safe Sulong could execute languages other than C that can be compiled to this IR, including C++ and Fortran.

Truffle. We used Truffle [68] to implement our LLVM IR interpreter. Truffle is a language implementation framework written in Java. To implement a language, a programmer writes an Abstract Syntax Tree (AST) interpreter in which each operation is implemented as an executable node. Nodes can have children that parent nodes can execute to compute their results.

Graal. Truffle uses Graal [72], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies optimistic optimizations based on assumptions that are later checked in the machine code [15, 60, 61]. If an assumption no longer holds, the compiled code *deoptimizes* [25], that is, control is transferred back to the interpreter, and the machine code of the AST is discarded. Graal optimizes based on safe semantics and cannot introduce false positives or false negatives with respect to the bugs that Safe Sulong finds.

LLVM IR Interpreter. The LLVM IR interpreter (approx. 60k lines of Java code) is at the core of Safe Sulong; it executes both the user application and the enhanced libc. It performs checks while executing the LLVM IR and aborts execution with an error when it detects a bug. First, a front end parses the LLVM IR and constructs a Truffle AST for each

LLVM IR function. The interpreter then starts executing the `main()` function's AST, which can invoke other ASTs. During execution, Graal compiles frequently executed functions to machine code.

JVM. Safe Sulong's interpreter can run on any JVM, since it is written in Java. However, to reach native speeds, it requires a JVM that implements the Java-based JVM compiler interface (JVMCI [53]). JVMCI will be included in OpenJDK 9 by default and enables Graal as a compiler. Note that our tool is platform-independent and provides the same bug-finding capabilities on all platforms. Additionally, Safe Sulong running on a Windows JVM can execute code that was written for libc under Linux.

3.2 Managed Objects and Type Safety

We base the execution model of Safe Sulong on an abstraction of the underlying machine. Our basic idea is to implement our interpreter in Java (i.e., in a high-level language) and represent C data structures as Java data structures. Since Java provides well-specified automatic bounds and type checks, the interpreter automatically checks and detects invalid accesses, such as out-of-bounds accesses, use-after-free errors, and `NULL` pointer dereferences. As C programs sometimes deliberately contain patterns that violate the C standard [6, 38], we relaxed our type rules (see below). Note that the interpreter could also have been implemented in another high-level language that provides these capabilities.

Figure 5 shows a simplified version of our class hierarchy, which is based on a previous Truffle implementation of C [21]. The base class for all objects is `ManagedObject`, from which subclasses for all primitives, pointers, functions, arrays, and structs inherit. To represent primitive types, we implemented classes that wrap a Java primitive. For example, to represent an LLVM IR `I32` object (which corresponds to a C `int` on AMD64), we use a Java `int`, since both have the same bit width. For some data types, no equivalent Java primitive exists; for example, Clang produces LLVM IR code that can contain integers with uncommon bit widths such as `I48`. We implemented such types using a Java `byte` array. To represent function pointers, we use a function ID to look up the AST for a function at a function call site. Note that we use inline caches to make function pointer calls efficient [24], and even enable speculative inlining [51]. For arrays, we use Java arrays. For structs, we employ an array-based map-like data structure that is provided by the Truffle framework [21, 69], and contains `ManagedObjects`. To represent pointers, we implemented an `Address` class that contains a reference to its pointee and an integer field `offset` used for pointer arithmetic.

Figure 6 shows an example where `malloc()` allocates an `int` array with three elements. Our interpreter maps this allocation to an `Address` that points to an `I32HeapArray` which holds a reference to a primitive Java `int` array (Section 3.3 explains on how we determine the allocation type). The `offset` in `Address` is initially `0`; when pointer arithmetics compute an address in the middle of an object, the `offset` is updated. For example, execution of the expression `arr[2]` first sets the `offset` to 8, which is computed by multiplying the size of `arr`'s type by 2. When the interpreter executes the load, it takes the offset from `Address`, divides it by 4 (since the dereferenced object is an `int` array), and uses the value 2 obtained to index the Java array.

In contrast to approaches that represent C objects as raw bytes that are stored in a large array [36] (e.g., LLJVM[5]), the presented type hierarchy guarantees type safety and restricts invalid pointer casts [10] when the cast pointer is used to read or write from the object. For example, in our architecture an integer array can only hold integer values and no `Addresses`; storing an `Address` would require converting it to an `int` that could be stored in the array. While strict type safety is beneficial to improve program quality and finding bugs, it can prevent real-world programs from executing [6, 31, 38]; for example, we found that many programs rely on invalid type casts to deliberately violate C's type rules. To provide a pragmatic solution, we relaxed the type safety rules to accommodate common patterns that we observed in real-world programs. For example, when the program stores a `double` in a `long` array, we simply take the bit representation of the `double`, convert it to a `long`, and store it in the array. As part of future work, we plan to further investigate the trade-offs between executing real-world programs and finding bugs.

3.3 Memory allocation

Every allocated object is either a stack object, a heap object, or a global object, that is, automatic, dynamic, or static memory, respectively. We know the type for stack allocations, and can thus directly allocate memory of the specified type in the function prologue. For heap objects (allocated by `malloc()`, `calloc()`, or `realloc()`) we do not know the type of object that will be stored in it. Thus, we allocate the corresponding Java object only on the first cast, read, or write access (i.e., when the type of the object becomes known) and propagate the type back to the allocation site (similar to allocation mementos in V8 [7]). The next time the allocation function is called, we directly allocate an object of the observed type. For global objects, the parser allocates objects at the start of the program.

We have subclasses of each data structure for each storage location. For example, an `I32Array` has the subclasses `I32AutomaticArray`, `I32HeapArray`, and `I32StaticArray`. Each heap object implements the `HeapObject` interface, which is used to free objects (see Figure 7). The `free()` method sets an object's data to `null` so that the garbage collector can reclaim the memory. Having different classes

[5]https://github.com/davidar/lljvm

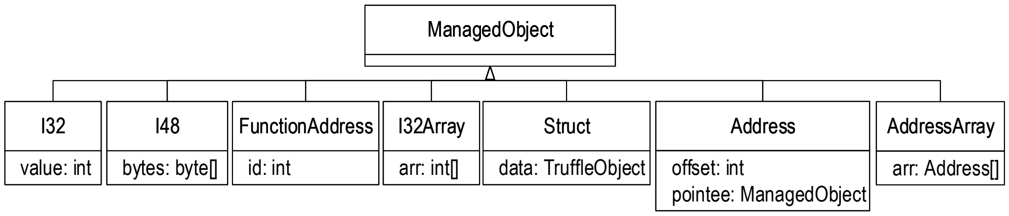

Figure 5. Simplified Class Hierachy of `ManagedObject`.

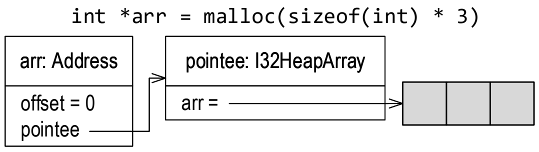

Figure 6. Example of pointer arithmetics and memory allocation (`I32HeapArray` is a subclass of `I32Array`).

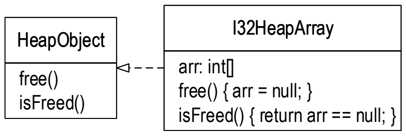

Figure 7. The `HeapObject` interface is used to free heap objects.

```
HeapObject obj =
  (HeapObject) pointer.pointee;
if (pointer.offset != 0) {
  throw new InvalidFreeException();
}
if (obj.isFreed()) {
  throw new DoubleFreeException();
}
obj.free();
```

Figure 8. Implementation of the free method.

for different storage locations also allows us to print meaningful error messages, since we can include the memory type of an object that is illegally accessed or freed.

3.4 Finding bugs

We implemented our bug-finding capabilities by relying on the JVM's automatic checks. In contrast to C, the Java language semantics require that illegal loads, stores, and casts result in an exception. Thus, a JVM cannot simply optimize invalid accesses away.

Out-of-bounds accesses. We translate load and store accesses to arrays in C to array accesses in Java. When the JVM executes the load, it first checks whether the index is in bounds; an out-of-bounds index access results in a Java `ArrayIndexOutOfBoundsException`. Note that such checks reduce the performance of Java programs. To address this, Java compilers such as Graal eliminate checks when they can prove that the index will always be in bounds [71]. As structs do not exist in Java, we represent them using a custom data structure [21, 69], for which we have to perform explicit bounds checks.

Use-after-free accesses. We map C objects that are allocated on the heap to Java objects that have references to the data objects via the `data` field. If an object is freed, the reference to its data is set to `null`. A subsequent access will result in a `NullPointerException`, since Java checks and prevents dereferences of `null`.

Double free errors. As shown in Figure 8, we explicitly check for double free errors in the AST node of the `free()` function using the `isFreed()` method specified by `HeapObject`. This method is implemented by checking whether the data field has already been set to `null`.

Invalid free errors. To detect invalid free errors, Safe Sulong first casts the object to be freed to the `HeapObject` interface. If the object was not allocated on the heap, a `ClassCastException` is thrown, since Java checks every type cast. Therefore, invalid free errors with a wrong pointee are detected. The code verifies next that the pointer offset is zero, that is, an exception is thrown if the pointer does not point to the start of the pointee. Only if the checks succeed is the pointee freed.

Variadic argument errors. Figure 9 shows how we implemented variadic arguments. A call to `va_start()` sets up the processing of variadic arguments by allocating space for a struct that holds a counter and an array of pointers to the variadic arguments. `va_start()` can also initialize this array, since the interpreter exposes the number of variadic arguments via the `count_varargs()` function, which can determine this number because the interpreter passes function arguments as a Java `Object` array that has a field for the array length. We do not require the user to specify the types of the variadic arguments, since we can obtain pointers to them via the `get_vararg()` function. When the user accesses a variadic argument via `va_arg()`, the current variadic argument index is used to access the pointer array. The result is then dereferenced using the user-specified type, which results in a type error for type violations. We can also detect an access to a non-existent variadic argument, as it

```
struct varargs {
  int counter;
  void **args;
};

#define va_list struct varargs *

#define va_start(ap, last)
ap = (va_list)malloc(sizeof(struct varargs));
ap->args = (void **)
    malloc(sizeof(void *) * count_varargs());

for (ap->counter = count_varargs() - 1;
     ap->counter != -1;
     ap->counter--) {
  ap->args[ap->counter] =
    get_vararg(ap->counter);
}

ap->counter = 0;

#define va_arg(ap, type) *((type *)
  (ap->args[ap->counter++]))
```

Figure 9. Implementation of variadic arguments.

would cause an out-of-bounds read of the malloced array. This allows our interpreter to detect the classic format-string problems [8].

4 Evaluation

In our evaluation, we primarily seek to demonstrate the effectiveness of Safe Sulong as a bug-finding tool (Section 4.1). We also show the resource costs of our implementation to argue that our approach is efficient enough to be used in practice. Safe Sulong's run-time performance varies during execution: at the beginning it is poorer than that of other tools (Section 4.2) but it improves when warmed up (Section 4.3). We performed all measurements on a quad-core Intel Core i7-6700HQ CPU at 2.60GHz on Ubuntu version 14.04 (with kernel 4.3.0-040300rc3-generic) with 16 GB of memory.

4.1 Effectiveness

We claimed that Safe Sulong is an effective bug-finding tool. To test this claim, we selected C projects from Github (see below) and executed them with Safe Sulong to find errors in them. We also sought to demonstrate that state-of-the-art approaches fail to detect common real-world bugs that Safe Sulong can detect. To this end, we executed each of the known faulty programs under the same conditions with ASan and Valgrind, that is, with the most popular compile-time and run-time instrumentation approaches, to check whether they could also find the error. Note that SoftBound+CETS [39, 40] was another candidate in the evaluation; however, it is not actively maintained and the version of SoftBound+CETS that is based on a recent LLVM version is still experimental.[6] Consequently, we restricted our evaluation to ASan and Valgrind. Although SoftBound+CETS was formally proven to find all memory-safety violations, we still expect false negatives because it detects memory errors that exist on the machine-level and not on the source-level as it applies LLVM's optimizations both before and after its passes [39, 40]; bugs that exist on the source-level could be removed by the compiler (**P2**).

We primarily selected small programs, ranging from 25 to 4792 (on average 289) lines of code (LOC)[7] because we observed that they were more likely to contain errors than larger projects, as they were often personal "hobby projects" that had not been tested with bug-finding tools. In some cases, this enabled us to find bugs simply by using Safe Sulong to execute the test suite of the project. When the project lacked a test suite, we executed the program, providing both expected input and corner cases. Finding bugs in larger programs would have required us to use automated testing strategies such as fuzzing [20]. Additionally, small programs were unlikely to use library functions that were not yet supported by Safe Sulong (see Section 6). Finally, many small projects relied only on the C standard library and were otherwise self-contained, so we did not have to compile additional dependencies.

In total, we found 68 errors in 63 projects and provided bug fixes for them, many of which were accepted by the project maintainers. Table 1 shows the distribution of the bugs, which roughly follows the distribution of the vulnerability and exploits databases (see Section 2.1). As expected, the majority of bugs were out-of-bounds accesses caused by strings not being `NUL`-terminated, not allocating enough space for a string to hold the `NUL` terminator, missing checks, integer overflows, incorrect hard-coded sizes, performing a check after an invalid access has already happened (see [65]), off-by-one errors in comparisons, and other errors. Table 2 shows that the out-of-bounds accesses included both reads and writes (with almost equal distribution) as well as buffer underflows and overflows. Most out-of-bounds accesses occurred to stack objects, but we also identified several to heap objects, global objects, and to the `main()` function's arguments. A smaller number of bugs were `NULL` dereferences that could also have been found without a bug-finding tool. We found only 1 use-after-free error and 1 variadic argument error (where arguments did not match the format string).

We compiled the programs with Clang using no optimizations (`-O0`), since we aimed to find as many errors as possible. In order to show that compiling with optimizations results

[6] See https://github.com/santoshn/SoftBoundCETS-3.9.

[7] To calculate the LOC, we used `cloc`, which omits comments and empty lines.

Buffer overflows	61
NULL dereferences	5
Use-after-free	1
Varargs	1

Table 1. Error distribution of the detected bugs.

Read	32
Write	29

Underflow	8
Overflow	53

Stack	32
Heap	17
Global	9
Main args	3

Table 2. Distribution of out-of-bounds accesses according to reads/writes, overflows/underflows, and memory kind.

in the failure to detect specific errors, we also compiled the programs at optimization level -O3 for ASan and Valgrind. We used standard options to execute Valgrind, but after finding out that ASan does not check zero-initialized global data by default, we had to enable the -fno-common compiler flag for ASan.

Valgrind -O0 and -O3 found slightly more than half of the errors because Valgrind reliably detects only out-of-bounds accesses to the heap and misses many of the out-of-bounds accesses to the stack and to global variables. Note that Valgrind detects reads of uninitialized values, so it could arguably be used to indirectly identify out-of-bounds reads to the stack (14 out of 31 stack accesses). However, we found that this feature is not reliable, and that compiling with either -O0 or -O3 reveals different but overlapping sets of bugs.

ASan -O0 detected 60 of the 68 errors that Safe Sulong found. Only 56 errors (a subset of those found with -O0) were also found with -O3, since in the other cases Clang optimized away bugs. From the 68 errors that Safe Sulong detected, 8 could neither be found by Valgrind nor by ASan at either optimization level (-O0 and -O3).

1. Uninstrumented main arguments array (P4, P1). We argued that to tools that are based on low-level approaches it is not always obvious whether the programs analyzed contain uninstrumented native code or data. We found that neither ASan [49] nor Valgrind detects out-of-bounds accesses to the main() function's arguments—a bug that we discovered in three applications. Figure 10 shows an example: the buffer for argv is created before the program (and libc) is invoked and is therefore not instrumented. Note that the main() function can have an additional argument for a pointer to an array of environment variables (declared as int main(int argc, char *argv[], char *envp[])); this array is initialized irrespective of the main() function's signature [37]. A missing or incorrect check might allow an attacker to exploit an out-of-bounds access to leak secrets contained in an environment variable.

```
#include <stdio.h>

int main(int argc, char** argv) {
    printf("%d %s\n", argc, argv[5]);
}
```

Figure 10. ASan does not detect out-of-bounds accesses to the main function.

```
const char t[2] = " \n";
token = strtok(buf, t);
```

Figure 11. The delimiter passed to strtok() is not NUL-terminated.

```
int counter;
// ...
printf("counter: %ld\n", counter);
```

Figure 12. A wrong format specifier is used, which causes an out-of-bounds read.

2. Missing interceptors (P1). Two bugs could not be found by ASan due to missing or incomplete interceptors; Valgrind did not find them, because the out-of-bounds accesses did not occur in heap-allocated objects. The first bug was caused by an unterminated string that the program passed to the strtok() libc function (see Figure 11). ASan failed to detect this bug, as it lacked an interceptor for strtok(), which we consequently implemented [47, 48]. We also found one error in which the program passed an integer to printf("%ld"), where the format string specified a long (see Figure 12). Note that Clang detected the bug statically and printed a warning; however, ASan did not detect the error, because the interceptor for printf() checks only pointer arguments.

3. Backend compiler optimizations (P2). In one case, a bug was eliminated by the compiler when compiling with -O0, namely a global array out-of-bounds access similar to that shown in Figure 13. Clang statically detected the out-of-bounds access and printed a warning. However, Clang's front end had not yet optimized away the bug, so Safe Sulong was able to detect it while executing the LLVM IR; not until LLVM's back end was it optimized away. Thus, ASan was unable to detect the bug. Valgrind would not have detected the bug in either case, since the array was not allocated on the heap. Arguably, a user could have found the bug given the compiler warning. However, we found cases were Clang -O0 optimized bugs away even without emitting a warning [50].

4. Overflowing the redzone (P3). As previously shown, shadow-memory red-zone approaches are inexact and cannot find all errors of a particular category. Safe Sulong found such a case in a program that reads a number and converts

```
int count[7] = {0, 0, 0, 0, 0, 0, 0};

int main(int argc, char** args) {
    return count[7];
}
```

Figure 13. The out-of-bounds error in this program is optimized away, even with optimizations disabled (`-O0` flag).

```
const char * strings[] = {"zero","one","two","↩
     three","four","five","six" /* ... */ };

void convert(FILE *input, FILE *output) {
    int number;
    fscanf(input, "%d", &number);
    // ...
    fprintf(output, "%s\n", strings[number]);
}
```

Figure 14. A large number as user input causes a buffer overflow that can exceed ASan's redzone.

it to a string; Figure 14 shows a simplified version of the program. In this example, the user input is used to index a global array; if the input number is too large, it causes a buffer overflow. ASan can only find the buffer overflow if the index is close to the object, that is, if it does not exceed the redzone; for our random inputs the access exceeded the redzone and the program either printed `(null)` or crashed. Valgrind could not find the error, since `strings` is a global buffer.

5. Missing variadic arguments (P1). In the projects we evaluated, we found only a few implementations of variadic functions. However, we identified a missing argument to the variadic `printf()` libc function. As in Figure 10, Clang detected the bug statically, since `printf()` is a well-known libc function. However, the bug could also have occurred in an application-specific function, where Clang would not have been able to detect it; similar format string vulnerabilities have recently been identified in libxml2 (CVE-2016-4448), in Dropbear SSH (CVE-2016-7406), and PHP (CVE-2016-4071). ASan and Valgrind cannot detect such errors at run time.

Discussion. Safe Sulong detected bugs that Valgrind and ASan missed. On the one hand, they missed bugs due to fundamental limitations of their approaches that cannot be addressed without significant enhancements. These stem from the limitations of shadow memory and redzones (4) as well as from their reliance on compilers that optimize based on unsafe semantics (3). On the other hand, they missed bugs due to implementation issues that could be addressed by implementation enhancements or fixes. The uninstrumented `main()` arguments (1) could be addressed by adding the missing instrumentation. Variadic arguments (5) could

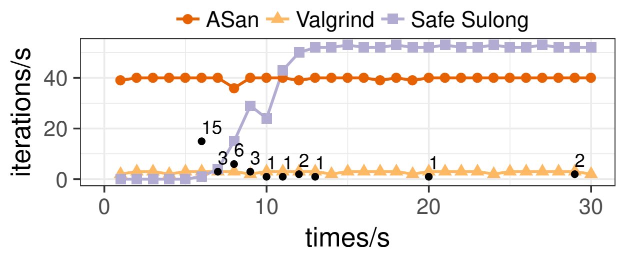

Figure 15. Warm-up time on the meteor benchmark. The x-axis shows the time in seconds and the y-axis the number of iterations an approach could run in the last second. The dots indicate the numbers of ASTs that Graal compiled up to that point.

also be instrumented, passing the argument types supplied at the call site and verifying them in the callee [3]. Missing or incomprehensive interceptors (2) could be addressed by comprehensive implementations or by instrumenting a libc implementation.

We argue that also implementation issues highlight the conceptual advantage of Safe Sulong's abstraction from the native execution model; in Safe Sulong, all accesses are checked automatically, so instrumentation for corner cases cannot be forgotten. Note that state-of-the-art static checkers can detect some of the bugs that were missed by Valgrind and ASan but detected by Safe Sulong; however, they suffer from false positives and other issues [30]. More precise tools such as SoftBound+CETS do not suffer from the limitations of shadow memory and redzones (4); however, SoftBound+CETS applies an unsound optimization where instrumentation for copied memory is omitted [39], which could result in missed bugs.

4.2 Start-up and Warm-up Costs

Safe Sulong uses a dynamic compilation approach and has therefore some additional run-time performance costs.

First, the LLVM IR interpreter has a noticeable start-up cost, which is the time between the user starting the program and the program beginning to run. We measured the start-up time by executing a `"Hello, World!"` program 100 times for each tool and measuring the execution time using `/usr/bin/time`. Safe Sulong needs slightly more than 600 ms to start up, during which the JVM initializes and starts Safe Sulong, which must then parse libc before calling the main function. Note that we could improve the start-up performance by lazily parsing libc and by improving the performance of our parser, but this was not the focus of our research. The start-up time of Safe Sulong for this program is longer than that of Valgrind, which needs around 500 ms to instrument and execute the program. With less than 10 ms, ASan starts up the fastest.

Second, the LLVM IR interpreter has a high warm-up cost, which is the time after start-up until the application reaches peak performance. Figure 15 illustrates the warm-up times of ASan, Valgrind, and Safe Sulong on the meteor benchmark. To approximate how Safe Sulong would behave for larger programs (which Safe Sulong currently fails to execute), we continuously executed the benchmark and plotted how many iterations per second the respective approach could execute over time. Safe Sulong's warm-up costs can be attributed mostly to the time the program spends in the interpreter; not until the interpreter has identified hot functions does Graal compile them to machine code. The curve shows a typical VM warm-up [2]. Not until second 6 did Safe Sulong complete its first execution of the benchmark. During this time, Graal had compiled the 15 most important functions to machine code. Afterwards, it soon sped up and executed more iterations per second than Valgrind (in second 7), and ASan (in second 11).

Note that the benchmark contains a call to `printf()` and to other libc functions, which Safe Sulong also interprets and compiles to machine code during execution. Even after compilation, the program fails to immediately reach peak performance, since we currently lack on-stack replacement [1, 19], which is used by production VMs to reduce the warm-up costs by switching from an interpreted method to a compiled method while executing in a loop [19, 26, 32]. However, our peak performance is higher than that of other tools as we demonstrate in Section 4.3. For ASan, we see that compile-time instrumentation approaches incur almost no warm-up costs, because checks are inserted during compilation and the runtime is initialized during start-up. Run-time approaches can insert checks either during start-up or on demand while executing the program. Valgrind inserts them while executing the program; nonetheless, the warm-up costs are not visible and likely overshadowed by the execution time needed for one iteration.

To address start-up and warm-up costs, the Graal project currently explores ahead-of-time compilation for the interpreter and the compiler [70, 72]. Applying this approach to Safe Sulong would allow us to create a standalone tool which would no longer require a JVM, and would have a smaller memory footprint and lower warm-up costs, since the parser and other components would have been compiled before starting the program.

4.3 Peak Performance

In this section we evaluate the peak performance that Safe Sulong can reach on long-running programs. Safe Sulong is a prototype and currently cannot execute large programs such as the SPEC benchmarks. Thus, we decided to evaluate benchmarks from the Computer Language Benchmark game [58], which contains smaller benchmarks for comparing the performance of different programming languages. When we executed this suite's fastaredux benchmark with Safe Sulong, we discovered that a loop ran out of bounds because, due to a rounding error, probabilities did not add up to the value `1.00`. We reported and fixed the bug [46], and used the fixed version in our evaluation. Additionally, we included the whetstone benchmark [43].

We measured the performance of executables compiled by Clang with disabled optimizations (`-O0`) and enabled optimizations (`-O3`) as baselines. Since our aim was to find as many errors as possible, we compiled the benchmarks using `Clang -O0` for all bug-finding tools, although Safe Sulong would also profit from compiler optimizations. In addition to assessing the performance of Safe Sulong, we also measured the performance of executables compiled by Clang 3.9 using ASan based on LLVM version 3.9 and Valgrind version 3.12. A direct comparison of run-time performance between different tools is not fair, since they provide different features. Our measurements are therefore intended to demonstrate that the peak performance of programs under Safe Sulong is sufficient to make our approach viable in practice. To approximate the performance of larger programs, we had to account for the adaptive compilation techniques of Truffle and Graal by setting up a harness that warmed up the benchmarks. By executing 50 in-process warm-up iterations, we ensured that every benchmark reached a steady state. We executed each benchmark 10 times and used the last iteration of each run as a sample for computing the peak performance. We also used the same benchmark harness for the other tools, even though their warm-up costs are minimal.

Figure 16 shows box plots for the peak performance relative to that of `Clang -O0`. We excluded Valgrind from the plots because it was `10`× to `58`× slower than `Clang -O0` on 5 benchmarks. Its slowdown was lowest on `spectralnorm`, `fasta`, and `fannkuchredux` (`2.3`, `3.6` and `5.1`, respectively). We did not plot the results for the `binarytrees` benchmark, since ASan was 14× slower and Valgrind 58× slower than `Clang -O0`. This slowdown was due to `binarytrees` being allocation-intensive, which suggests that current bug-finding approaches cannot deal well with allocation-intensive benchmarks. On this benchmark, Safe Sulong was only 1.7× slower than `Clang -O0`. In almost all benchmarks, Safe Sulong was faster than `ASan -O0`; they were on a par only on `fastaredux`. Safe Sulong was faster than `Clang -O0`, except on the `fastaredux` and `nbody` benchmarks. On `fannkuchredux` and `mandelbrot`, Safe Sulong was even on a par with `Clang -O3`. Safe Sulong exhibited the poorest performance on `fastaredux`, where it was `2.5`× slower than `Clang -O0`. As part of future work, we plan to further reduce Safe Sulong's overhead.

5 Limitations

Native interoperability. Interoperability with pre-compiled binaries is a double-edged sword. It is necessary to execute closed-source libraries [62] and convenient for users, but it results in overlooked bugs, as our findings have demonstrated. What sets Safe Sulong apart from

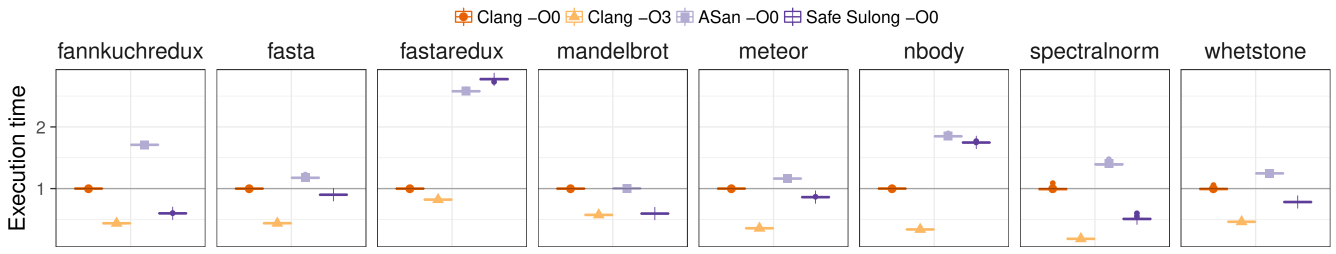

Figure 16. Execution times relative to `Clang -O0` (peak performance, lower is better).

state-of-the-art shadow-memory red-zone approaches (which are inherently inexact) is that our approach is exact and aims to find all errors of a category. To maintain this property, Safe Sulong does not provide a native function interface. This is also the source of Safe Sulong's most significant drawback, as this renders it unusable with programs that require this interoperability. We want to address this issue in the future by using *binary translation* to convert binaries to LLVM IR, which can be executed by Safe Sulong. Translators that can convert binary code to LLVM IR already exist; MC-Semantics [13], REVAMB [12], QEMU [5] support x86, and LLBT [57] support the translation of ARM code.

Warmup time. As discussed in Section 4.2, Safe Sulong needs significantly more time to execute small programs due to warm-up time. Similar to state-of-the-art JVMs, we start by running the program in an interpreter and only compile frequently executed functions to machine code. For approaching the warm-up time of current JVMs, we still lack on-stack replacement, which would allow us to switch to a compiled version of a function while executing in its loop. The Graal project is experimenting with ahead-of-time compilation of the interpreter and the JIT compiler to provide a long-term solution that reduces warm-up time. Note that the JIT compilation approach allows Safe Sulong to reach better peak performance than other bug-finding tools, which could make it applicable to long-running server applications in production.

Programs that rely on non-standard C. Safe Sulong cannot execute all programs that occur "in the wild". We assume that a programmer wants to eliminate undefined behavior from the execution. Consequently, we also require that a program does not violate the type rules of the C standard (e.g., the strict-aliasing rule [10, 31]). Since type violations continue to be relatively common, we relaxed some of the type rules to accommodate real-world code. Additionally, a previous survey discussed certain non-standard-compliant C patterns that are commonly assumed to work [38]. Currently, Safe Sulong lacks support for many such patterns, for example, for tagged pointers where pointers are converted to integers, values stored in spare bits, and converted back to an address. We could implement further relaxations to support such patterns; for instance, we could allow users to store integers in the `offset` field of `Address`.

6 Future Work

Completeness. Safe Sulong can execute most LLVM IR instructions, but is still a prototype that fails to execute larger programs such as the SPEC benchmarks. Such programs require system libraries (most importantly libc) which rely on system calls, inline assembly, compiler builtins, and linker features. As part of future work, we will extend Sulong to support these features, and replace our custom, incomplete libc by an existing, complete one such as musl libc.

Detection of memory leaks. Many bug-finding tools such as Dr. Memory, Valgrind, Purify, and ASan offer support for memory-leak detection. Our approach is based on an exact garbage collector, which reclaims memory when it is no longer needed, irrespective of whether it has been freed or not. We plan to add support for detecting objects that have not been freed by having a background thread that is notified when the garbage collector collects an object (using Java's `PhantomReferences`). When this thread receives a notification, we can check whether the object has been freed manually to print an error in case it has not.

Replace Clang as a front end. As we have demonstrated, Clang (and other C compilers) can optimize away code with undefined behavior even with optimizations disabled. We cannot exclude the possibility that Clang optimized away other bugs that could then no longer be found by ASan, Valgrind, and Safe Sulong. To address this issue, we intend to implement a C front end that does not perform any optimizations.

7 Conclusion

In this paper, we have presented a novel bug-finding tool for C programs that is based on abstraction of the underlying machine. We implemented our approach in a tool called Safe Sulong, which discovered several errors in open-source projects that current bug-finding tools could not find. By using dynamic compilation, Safe Sulong reaches a peak performance that is comparable to that of Clang `-O0`, and even that of Clang `-O3` in some cases.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We thank Stephen Kell for his insightful comments, and Ingrid Abfalter for proofreading and editorial assistance. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for their support and contributions. The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

A Found Bugs

We uploaded a list of found bugs to http://ssw.jku.at/General/Staff/ManuelRigger/ASPLOS18-SafeSulong-Bugs.csv.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000. Adaptive Optimization in the JalapeñO JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 47–65. https://doi.org/10.1145/353171.353175

[2] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133876

[3] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. 2017. Venerable Variadic Vulnerabilities Vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 186–198.

[4] 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 (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 213–223.

[5] Vitaly Chipounov and George Candea. 2010. *Dynamically Translating x86 to LLVM using QEMU.* Technical Report. École polytechnique fédérale de Lausanne.

[6] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. 2015. Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 117–130. https://doi.org/10.1145/2694344.2694367

[7] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 International Symposium on Memory Management (ISMM '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2754169.2754181

[8] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic Protection from Printf Format String Vulnerabilities. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, 15–15.

[9] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, Vol. 2. IEEE, 119–129.

[10] Pascal Cuoq, Loïc Runarvot, and Alexander Cherepanov. 2017. Detecting Strict Aliasing Violations in the Wild. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 14–33.

[11] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 144–157.

[12] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 17.

[13] Artem Dinaburg and Andrew Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada.*

[14] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optimization. In *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE, 73–87.

[15] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings VMIL '13*. 1–10.

[16] Frank Ch Eigler. 2003. Mudflap: Pointer Use Checking for C/C+. *Proceedings of the First Annual GCC Developers' Summit* (2003), 57–70.

[17] David Evans and David Larochelle. 2002. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Softw.* 19, 1 (Jan. 2002), 42–51. https://doi.org/10.1109/52.976940

[18] Matthias Felleisen and Shriram Krishnamurthi. 1999. *Safety in Programming Languages.* Technical Report. Rice University.

[19] Stephen J. Fink and Feng Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-stack Replacement. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. 241–252.

[20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated Whitebox Fuzz Testing.. In *NDSS*, Vol. 8. 151–166.

[21] Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-safe Execution of C on a Java VM. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security (PLAS'15)*. ACM, New York, NY, USA, 16–27. https://doi.org/10.1145/2786558.2786565

[22] Reed Hastings and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference.* Citeseer.

[23] John L Henning. 2000. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer* 33, 7 (2000), 28–35.

[24] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38.

[25] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[26] Urs Hölzle and David Ungar. 1994. Optimizing Dynamically-dispatched Calls with Run-time Type Feedback. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 326–336. https://doi.org/10.1145/178243.178478

[27] Gerard J Holzmann. 2002. Static source code checking for user-defined properties. In *Proc. IDPT*, Vol. 2.

[28] Intel. [n. d.]. Intel Inspector 2017. ([n. d.]). https://software.intel.com/en-us/intel-inspector-xe

[29] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In

Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02). USENIX Association, Berkeley, CA, USA, 275–288.

[30] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, Piscataway, NJ, USA, 672–681.

[31] Stephen Kell. 2016. Dynamically Diagnosing Type Errors in Unsafe Code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016).* ACM, New York, NY, USA, 800–819. https://doi.org/10.1145/2983990.2983998

[32] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot&Trade; Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[33] Chris Lattner. 2011. What Every C Programmer Should Know About Undefined Behavior. (2011). http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[34] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA.* 75–88. https://doi.org/10.1109/CGO.2004.1281665

[35] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* ACM, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[36] J. Martin and H. A. Muller. 2001. Strategies for migration from C to Java. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering.* 200–209. https://doi.org/10.1109/.2001.914988

[37] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[38] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the depths of C: elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016.* 1–15. https://doi.org/10.1145/2908080.2908081

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09).* ACM, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

[40] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010.* 31–40. https://doi.org/10.1145/1806651.1806657

[41] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526. https://doi.org/10.1145/1065887.1065892

[42] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).* ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[43] Painter Engineering. [n. d.]. Whetstone benchmark. ([n. d.]). http://www.netlib.org/benchmark/whetstone.c.

[44] Parasoft. [n. d.]. Insure++. https://www.parasoft.com/product/insure/.

[45] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. (2010). https://blog.regehr.org/archives/213.

[46] Manuel Rigger. 2016. Fix for fasta-redux C gcc #2 program. (2016). https://alioth.debian.org/tracker/?func=detail&atid=413122&aid=315503&group_id=100815.

[47] Manuel Rigger. 2017. Add an interceptor for strtok/rL298650. (2017). https://reviews.llvm.org/rL298650.

[48] Manuel Rigger. 2017. Asan does not detect ouf-of-bounds read in strtok. (2017). https://github.com/google/sanitizers/issues/766.

[49] Manuel Rigger. 2017. Asan does not detect out-of-bounds accesses to argv #762. (2017). https://github.com/google/sanitizers/issues/762.

[50] Manuel Rigger. 2017. Clang -O0 performs optimizations that undermine dynamic bug-finding tools (LLVM Developers Mailing List). (2017). http://lists.llvm.org/pipermail/llvm-dev/2017-March/111371.html.

[51] 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

[52] Manuel Rigger, Roland Schatz, Rene Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. 2018. Introspection for C and its Applications to Library Robustness. *The Art, Science, and Engineering of Programming* 2 (2018).

[53] John Rose. 2014. JEP 243: Java-Level JVM Compiler Interface. (2014). http://openjdk.java.net/jeps/243.

[54] SANS. 2011. CWE/SANS TOP 25 Most Dangerous Software Errors. (2011). https://www.sans.org/top25-software-errors/.

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

[56] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference, General Track.* 17–30.

[57] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems.* ACM, 51–60.

[58] shootouts. [n. d.]. The Computer Language Benchmarks Game. ([n. d.]). http://benchmarksgame.alioth.debian.org/.

[59] Geoffrey Smith and Dennis M. Volpano. 1998. A Sound Polymorphic Type System for a Dialect of C. *Sci. Comput. Program.* 32, 1-3 (1998), 49–72. https://doi.org/10.1016/S0167-6423(97)00030-0

[60] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013.* 9:1–9:8. https://doi.org/10.1145/2489837.2489846

[61] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014.* 165. https://doi.org/10.1145/2544137.2544157

[62] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* 48–62. https://doi.org/10.1109/SP.2013.13

[63] 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

[64] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: what happened to my code?. In *Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012*. 9. https://doi.org/10.1145/2349896.2349905

[65] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. 260–275. https://doi.org/10.1145/2517349.2522728

[66] Henry S Warren. 2013. *Hacker's delight.* Pearson Education.

[67] GCC Wiki. 2014. Mudflap Pointer Debugging. (2014). https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging].

[68] 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

[69] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An object storage model for the truffle language implementation framework. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*. 133–144. https://doi.org/10.1145/2647508.2647517

[70] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[71] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array bounds check elimination for the Java HotSpot client compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ 2007, Lisboa, Portugal, September 5-7, 2007*. 125–133. https://doi.org/10.1145/1294325.1294343

[72] 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

[73] 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