Sulong, and Thanks For All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model

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 machine. They insert additional checks in an ad-hoc fashion, which makes them prone to forgotten checks for corner-cases. To address this issue, we devised a novel approach to find bugs during the execution of a program. At the core of this approach lies an interpreter that is written in a high-level language that performs automatic checks (such as bounds checks, `NULL` checks, and type checks). By mapping C data structures to data structures of the high-level language, accesses are automatically checked and bugs are found. We implemented this approach and show that our tool (called *Safe Sulong*) can find bugs that have been overlooked by state-of-the-art tools, such as out-of-bounds accesses to the main function arguments. Additionally, we demonstrate that the overheads are low enough to make our tool practical, both during development and in production for safety-critical software projects.

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 thus abort the program. Errors induce *undefined behavior*; in practice they can corrupt memory, leak sensitive data, change the control flow, or crash the program. Sometimes, errors remain undetected since they can cause delayed failures or possibly do not exhibit any visible symptoms. Therefore, memory errors in C often result in hard-to-find bugs or enable attackers to exploit them.

To tackle this issue, industry and academia have come up with a plethora of static and dynamic tools to find bugs in C programs. Static tools perform analyses on the source code to detect errors of certain bug categories; they typically rely on necessarily incomplete heuristics and exhibit both false positives and false negatives [11, 19]. In contrast, dynamic tools insert additional checks either as part of the compilation process or at run time, and find errors during the execution of the program. Although they only find those errors that occur during a specific run of the program, they are expected to find all errors and do not exhibit false positives. Both static and dynamic bug finding tools have been widely successful and have detected many bugs in commonly used libraries.

In this paper we concentrate on dynamic bug finding tools and argue that state-of-the-art approaches such as LLVM's AddressSanitizer (ASan) [28] and Valgrind [24] miss real-world errors that programmers expect to be found. We argue that this stems from current approaches not abstracting from the machine's low-level execution model, but instead relying on a patchwork of additional checks at their core where a check can be easily forgotten. Furthermore, the checks are implemented using inexact techniques, which by design makes these tools miss errors. Dynamic bug finding tools are based on static compilers, or are employed after compilation. It is known that compiler optimizations at higher optimization levels interfere with bug finding tools; we show that compilers can also optimize away memory errors even when explicitly compiling without optimizations (i.e., with the `-O0` flag). Finally, interoperability with uninstrumented native code makes it difficult to determine when native code is executed or uninstrumented data structures are accessed, which gives users a false sense of security and may cause tool writers to overlook checks. As one example, out-of-bounds accesses to the `main()` function's arguments are not detected by current bug finding tools.

In this paper, we present a novel approach to find bugs during run time and to address these issues. Based on this approach, we implemented a tool (called *Safe Sulong*) that can detect out-of-bounds accesses, use-after-free errors, invalid-free errors, double-free errors, use-after-free errors, and `NULL` dereferences. Additionally, it can detect accesses to non-existent variadic arguments, which no other tool can currently detect in the general case.[1] Our approach abstracts from the machine's execution model through an execution environment for C that is written in a high-level language. By abstracting pointers and other C data structures and representing them with the data structures of the high-level language, we can rely on well-defined automatic checks of the high-level language to detect bugs in the C program. While we used Java for our implementation, the approach also works for other languages that check and disallow buffer overflows, `NULL` pointer dereferences, and use-after-free errors. 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 Java semantics, where run-time errors in the program have to cause run-time exceptions. We do not provide interoperability with pre-compiled native code, since 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 where no source code is available. We want to address this in future work by using binary translation to translate machine code to a low-level intermediate representation that we can execute.

Our evaluation shows that Safe Sulong has a higher warm-up cost than current approaches, but a peak performance that is better than other bug finding tools. We detected and fixed 62 errors with our tool, out of which 8 were not found by ASan and Valgrind. Overall, this paper provides the following contributions:

- We provide an analysis of memory errors over the last five years to identify which memory errors are important to find.
- We discuss limitations in state-of-the-art approaches and demonstrate them on real-world examples.
- We present an alternative approach for bug finding that abstracts from the machine.
- We implemented our approach and evaluated its start-up costs, warm-up costs, peak performance, and memory usage.

[1] Note that there are static approaches such as FormatGuard [6] which can statically detect missing variadiac arguments in functions that use format strings.

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 a key-word search on the Common Vulnerabilities and Exposures (CVE)[2] and the ExploitDB[3] databases. Unlike a previous study on memory errors (up to 2012) [33] we grouped the errors into different bug categories. Note that we only concentrated on memory errors (i.e., dereferencing invalid pointers) and thus did not consider memory leaks, reading from uninitialized memory, and other C errors.[4] Figure 2 and Figure 3 show the results for 2012 to 2017. Note that bug categories with a high number of vulnerabilities were also exploited more often.

Out-of-bounds accesses. The most dangerous bug category (as already previously shown [27, 7, 33]) refers to out-of-bounds accesses to objects, which is also known as spatial memory safety errors. Such bugs are not only still relevant today; they are on an all-time high. We refer to an out-of-bounds access as a buffer overflow when it attempts to access memory past the end of the object, and as buffer underflow when it accesses memory before the beginning of the object. Bug finding tools typically differ on whether they can detect out-of-bounds accesses to the stack, heap, global (static) data as well as whether they detect read and/or write accesses. For example, Valgrind can only find heap buffer overflows.

Use-after-free errors. The second most dangerous bug category is use-after-free (known as a 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 grouped invalid-free errors, double-free errors, and accesses to non-existent variadic arguments as "other errors". An invalid-free error is caused

[2] `https://cve.mitre.org/`

[3] `https://www.exploit-db.com/`

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

```c
int sum(int count, ...) {
    va_list argp;
    va_start(argp, count);
    int res = 0;
    for (int i = 0; i < count; i++)
        res += va_arg(argp, int);
    return res;
}

int main() {
    int count = 3;
    printf("%d\n", sum(count, 4, 5));
}
```

Figure 1: Example for an access to a non-existent variadic arguments

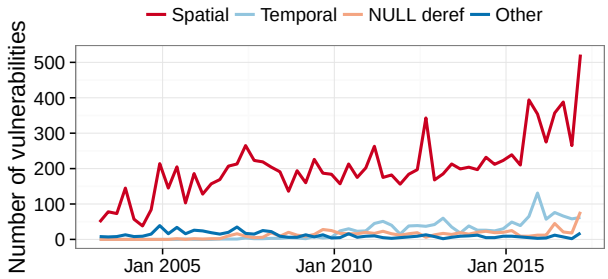

Figure 2: Number of reported vulnerabilities in the CVE database (2012-03 to 2017-03)

when a pointer to a stack object or to a global object is passed to `free()`, or when the passed pointer points into the middle of an object. Double-free errors occur when a heap object is freed twice. Accesses to non-existent variadic arguments happen when the number of passed variadic arguments is less than the function expects. Figure 1 shows an example where a `sum()` function tries to read three variadic arguments (based on the `count` argument), although only two are passed. One subclass of this error are format-string vulnerabilities, where the format string specifies how many arguments on the stack should be accessed.

From this initial study we conclude that a bug finding tool for memory safety errors needs at least be able to detect out-of-bounds accesses, use-after-free errors, and `NULL` dereferences.

2.2 State of the Art

Shadow memory. Most practical bug finding tools such as ASan [28], Mudflap [10], Valgrind [24], Dr. Memory [3], and Purify [15] base their bug finding capabilities on the concept of shadow memory. They maintain metadata for application memory in a separate memory area referred to as shadow memory. When a program allocates memory, the runtime of the shadow

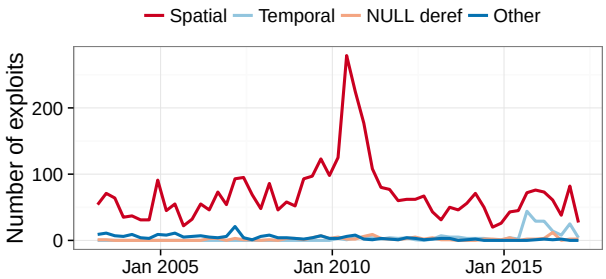

Figure 3: Number of available exploits in the ExploitDB (2012-03 to 2017-03)

memory tool marks the shadow memory area associated with the program memory as accessible, and a region around it as inaccessible (called a redzone). This metadata is used to verify certain actions; for example, read accesses validate that a memory cell is accessible. Some tools also provide additional shadow registers to track values in registers. Most shadow memory 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 during compile-time 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-instrumentation approach is LLVM's AddressSanitizer which initially detected out-of-bounds accesses, use-after-free errors, and `NULL` dereferences [28], 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. Mudflap [10] was used by the GCC project until GCC 4.9 when it was superseded by the AddressSanitizer. It was known to have several shortcomings such as reporting false positives and not detecting buffer overflows for neighboring objects in the memory.[5] Commercial tools include Purify [15], which is not strictly a compile-time approach, since it inserts code into object files, and Insure++.

Dynamic instrumentation. Dynamic instrumentation involves inserting checks on the binary level during the execution of the program. The advantages of dynamic instrumentation are that it works for any language that is compiled to machine code, that all code is checked,

[5] see `https://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging`

even if the source code is not available, and that it does not require recompilation [29]. The most widely used run-time instrumentation approach is Valgrind. Other dynamic instrumentation approaches include Dr. Memory [3], and Intel Inspector[6]. Note that binary instrumentation approaches cannot detect ouf-of-bounds accesses to the stack (unless the top of the stack is exceeded).

2.3 Limitations of Current Approaches

Our main focus is finding all memory safety errors in a C program. Under this considerations, current approaches have several limitations.

Problem 1: Lack of abstraction from the machine. Current bug finding tools do not abstract from the low-level execution model of the machine, since they aim to provide interoperability with existing machine code. We argue that such tools follow a "patchwork architecture" that results in a number of limitations for bug finding tools. They insert additional checks in the program, either as a separate phase in an an existing compiler (such as ASan) or to existing native code (such as Valgrind). They typically need to instrument all read and write operations, all allocations and deallocations, as well as all system calls [24]. A forgotten check cannot be easily found, since in many cases the program behaves as intended, with the only difference that specific bugs went undetected. Additionally, current tools are based on low-level details of the respective platform. As a consequence, bug finding capabilities cannot easily be ported to other platforms [15]. Finally, current approaches are implemented in low-level languages such as C/C++ with core parts written in assembler (for performance reasons). These languages provide few safety guarantees, which makes the bug finding tools prone to the same kind of errors that they want to find.

Problem 2: Compiler optimizations. Current bug finding tools are built on top of an optimizing compiler such as Clang or GCC. As previously noted [34], this is an issue for bug finding tools, since they implement C semantics that are different from the ones 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 [35].

On the one hand, 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 [28] where a series of loads is transformed to 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 on the system level; however, ASan classified it as a bug since the access would be out-of-bounds in C. While this issue has been fixed by disabling load widening [28], such compiler optimizations can still cause false positives in dynamic instrumentation bug finding tools (such as Valgrind [29]).

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

Figure 4: A C program with a potential out-of-bounds access is reduced to `return 0` by optimizing compilers.

The more serious problem is that compiler optimization can lead to missed errors, that is, *false negatives*. It is widely known that on high optimization levels (e.g., with the `-O2` flag), compilers optimize the code based on the fact that error semantics are undefined. It also has been shown that compilers can remove redundant null pointer checks, even at -O0 [35]. We have also found, that Clang can optimize away memory safety errors at -O0. For example, consider the (contrived) function in Figure 4. The function initializes elements of an array without further using it. 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 on the binary level, and current bug finding approaches are unable to find them.

Since the compiler can optimize away bugs (or cause false positives), many projects decide to disable optimizations for testing altogether (with the `-O0` flag) and accept performance degradations. However, as we will demonstrate, explicitly disabling optimizations does not stop compilers from optimizing away bugs.

Problem 3: Inexact approaches Shadow memory approaches have not been designed to detect all bugs of a certain category. First, they cannot detect all out-of-bounds accesses. When an access to an object runs out-of-bounds and lands inside a different object, the access is not detected as a bug. Also, if the redzone of a global variable is exceeded, the shadow memory check is ren-

[6] `https://software.intel.com/en-us/intel-inspector-xe`

dered useless. Second, shadow memory approaches cannot reliably detect use-after-free errors. When freeing an object, these approaches mark the object's 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 [28] and Purify [15] rely on heuristics to avoid that freed memory is quickly allocated again.

Problem 4: Finding invalid accesses in the libc.
A challenge for bug finding tools is how to support external libraries. Run-time instrumentation approaches support existing machine code by design. 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. We argue that such interoperability is not necessarily desired, since users (and even tool writers) might overlook errors in this precompiled code.

To achieve a higher coverage, compile-time instrumentation approaches recommend to create special instrumented builds for external libraries [28]. This is a challenge for the `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 comparisons [36, Section 6.1], which can produce out-of-bounds accesses similar to the load widening optimization. Current compile-time instrumentation tools disable instrumentation or checks for such functions, or replace them altogether.

As a pragmatic alternative, compile-time instrumentation approaches such as ASan and Mudflap provide so-called *interceptors* that wrap the system library functions and call them only after performing validity checks on the arguments. This approach is dangerous when users expect these interceptors to be comprehensive. As we will 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 [29].

Another challenge that is specific to `libc` implementations is that they heavily use non-standard C extensions such as function attributes, that are rarely used in user applications. Figure 5 shows an example that we found[7], where a function attribute causes Clang to optimize away an out-of-bounds access, even when compiling with `-O0`. The call to `isdigit()` is resolved to an access in a lookup table that is obtained by a call to `__ctype_b_loc()`. Since the argument is not in the range of an `unsigned char`, the call would cause an out-of-bounds access to the lookup table. However, Clang removes the call since it has an `__attribute__((__const__))` that specifies that the function has no side effects.

```c
#include <ctype.h>

int main() {
    isdigit(1000000);
}
```

Figure 5: This code fragment, which would cause an out-of-bounds access in the `libc`, is optimized away by Clang `-O0`.

3 Implementation

Safe Sulong was developed to address the four problems mentioned in Section 2.3. First, 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 machine and implements a simple execution model; it executes C code using an interpreter written in Java that relies on automatic checks of the language. The C interpreter uses an exact approach (to address **P3**), so no errors are missed. It also provides a `libc` that we specifically designed for finding bugs. We do not provide interoperability with native code, since this could undermine the bug finding capabilities (see **P3**). 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 Java semantics and cannot optimize away invalid accesses. In summary, our approach allows us to reliably find errors in C programs without sacrificing run-time performance.

3.1 System Overview

Figure 6 shows the architecture of Safe Sulong. It comprises the following components:

Bug finding libc. We argued that current `libc` implementations (which are optimized primarily for performance) are detrimental to bug finding tools. To address

[7] `http://lists.llvm.org/pipermail/llvm-dev/2017-March/111371.html`

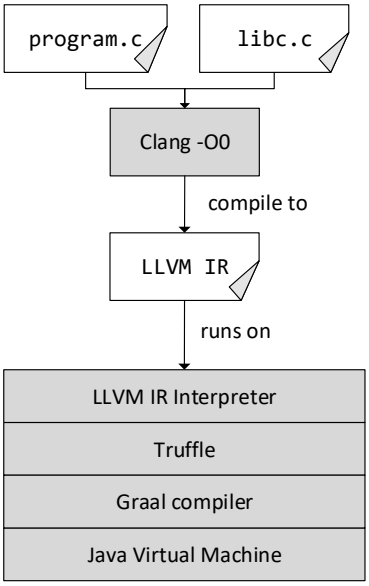

Figure 6: Overview of Safe Sulong

this issue, we implemented a `libc` that is tailored to finding bugs. Our `libc` is written in standard C and does not rely on any GNU extensions. It uses *introspection* to perform additional checks based on run-time information (which we will describe in a future paper). For example, for functions which expect a buffer and a buffer length (such as `gets_s()`), it uses bounds information to check whether the passed size does not exceed the actual bounds of the buffer. This allows the `libc` to print warnings for bugs in the program that are not triggered during execution. To implement the `libc`, Safe Sulong exposes functions that are implemented in Java and can be called similarly to 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.

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 [21], 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. In future work, we want to replace Clang with a non-optimizing frontend, to eliminate this risk completely (see Section 6). Since LLVM IR retains all C characteristics that are important for this paper, we will talk about C objects when we actually refer to LLVM IR 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 [37] 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 [40], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies aggressive optimistic optimizations based on assumptions that are later checked in the machine code. If an assumption no longer holds, the compiled code *deoptimizes* [17], that is, control is transferred back to the interpreter and the machine code of the AST is discarded.

LLVM IR Interpreter. The LLVM IR interpreter is the core of Safe Sulong; it executes both the user application as well as 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. Then, the interpreter starts executing the `main()` function's AST, which can invoke other ASTs. During execution, Graal compiles frequently executed functions to machine code.

JVM. The system runs on any JVM that implements the Java based JVM compiler interface (JVMCI [26]). JVMCI supports Graal and other compilers written in Java. 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. To achieve this, Safe Sulong does not provide interoperability with pre-existing native code.

3.2 Managed Objects and Type Safety

We base the execution model of Safe Sulong on abstraction from 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 by 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. Note that the interpreter could also have been implemented in another high-level language that provides these capabilities.

Figure 7 shows a simplified version of our class hierarchy, which is based on a previous Truffle implementation of C [14]. The base class for all objects is

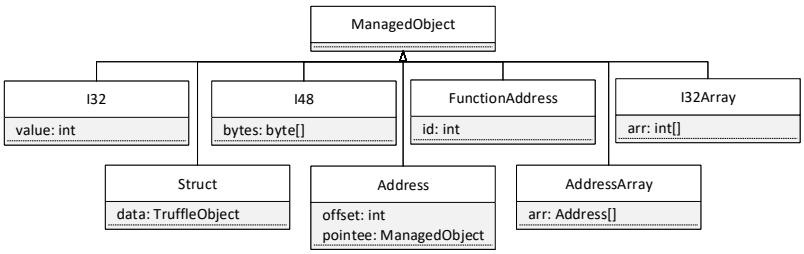

Figure 7: Simplified Class Hierachy of `ManagedObject`

`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 with a Java `byte` array. To represent function pointers, we use a function ID that we use 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 [16], and even enable speculative inlining [25]. For arrays, we use Java arrays. For structs, we use an array-based map-like data structure that is provided by the Truffle framework [38, 14]. To represent pointers, we implemented an `Address` class that contains a reference to its pointee and an integer field `offset` used for pointer arithmetics.

Figure 8 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` that holds a reference to a primitive Java `int` array. 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 obtained value `2` to index the Java array.

The presented type hierarchy guarantees type safety and restricts type punning (i.e., incompatible pointer casts), which has undefined behavior according to the C standard. We restrict type punning by detecting invalid casts when the casted pointer is used to read or write from the object. For example, in our architecture a pointer array can only hold `Addresses` and no integers; storing an integer would require converting it to an

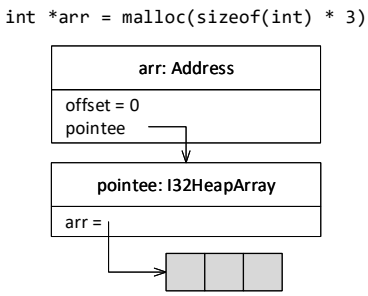

Figure 8: Example on pointer arithmetics and memory allocation

`Address` that could be stored in the array. While strict type safety is beneficial for improving the program quality and finding bugs, it can prevent real-world programs from execution; we found that many programs rely on type punning to store primitives of one type into a primitive or primitive array of another type. As a pragmatic solution, we relaxed the type safety rules to accomodate 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 into the array. In future work, we want to formalize these type rules and demonstrate their compliance with the C standard.

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 with the specified type in the function prologue. For heap objects (allocated by `malloc()`, `calloc()`, or `realloc()`) we do not yet know the type of the object that will be stored in it. Thus, we allocate the respective 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

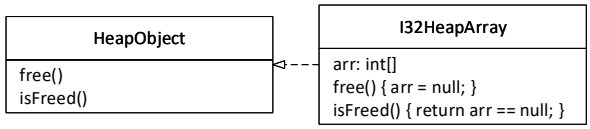

Figure 9: The `HeapObject` interfaces is used to free heap objects

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

Figure 10: Implementation of the free method

allocation mementos in V8 [5]). 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`, `I32DynamicArray`, and `I32StaticArray`. Each heap object implements the `HeapObject` interface which is used to free objects (see Figure 9). The `free()` method sets the object's data to `null`, so that the garbage collector can reclaim the memory. Having different classes 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 Java'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 stores 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 [39]. For structs, we currently perform explicit bounds checks where similarly to native approaches a check could be forgotten.

Use-after-free accesses. We map C objects that are allocated on the heap to Java objects that have a reference to the data object over 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 10, 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. For detecting 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. Next, the code verifies 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 the pointee is freed.

Type errors and non-existing variadic arguments. Figure 11 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 over the `count_varargs()` function; which can determine this number because the interpreter passes function arguments over a Java `Object` array that has a field for the array length. Also, we do not require the user to specify the types of the variadic arguments since we can get 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 by the user-specified type. We can detect an access to a non-existent variadic argument (and also type errors), since it would cause an out-of-bounds read of the malloced array. This allows our interpreter to detect the classic format string problems.

4 Evaluation

In our evaluation, we primarily want to demonstrate the effectiveness of Safe Sulong as a bug finding tool (Section 4.1). We also demonstrate the resource costs of our implementation to argue that our approach is efficient enough to be used in practice. Safe Sulong is based on a dynamic approach, which makes it difficult to directly

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

#define va_list struct varargs *

#define va_start(ap, count)
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 11: Implementation of variadic arguments

compare it with static approaches. First, it has different memory usage characteristics than other tools (Section 4.2). Second, its run-time performance varies during execution: at the beginning it is worse than other tools (Section 4.3) but it becomes faster when warmed up (Section 4.4). 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 evaluate this claim, we selected C projects from Github (see below) and executed them with Safe Sulong to find errors in them. We also want to demonstrate that state-of-the-art approaches fail to detect common real-world bugs that Safe Sulong can detect. To evaluate this, we executed each of the found erroneous 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.

We primarily selected small programs, reaching from 25 to 1248 (on average 220) lines of code (LOC)[8]. We observed that small open source projects were more likely to contain errors than larger projects because they were often personal "hobby projects" that have not been tested with bug finding tools. Occasionally, this enabled us to find bugs by simply executing the test suite of the project with Safe Sulong. 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 [13]. Furthermore, many small projects only relied on the C standard library and were otherwise self-contained, so that we did not have to compile additional dependencies. Finally, smaller projects were less likely to contain features that are not yet supported by Safe Sulong (see Section 6).

Buffer overflows	55
NULL dereferences	5
Use-after-free	1
Varargs	1

Table 1: Error distribution of the bugs found by Safe Sulong

Read	29	Underflow	8	Stack	31
				Heap	14
Write	26	Overflow	47	Global	8
				Main args	3

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

Safe Sulong is an effective bug finding tool. In total, we found and fixed 62 errors in 57 projects.[9] Table 1 shows the distribution of the bugs, which roughly follows the distribution of the vulnerability and exploits databases (see Section 2.1). As we expected, the majority of bugs were out-of-bounds accesses. They were caused by strings being not `NULL`-terminated, not allocating enough space for a string to hold the `NULL` terminator, missing checks, integer overflows, wrong hard-coded sizes, performing a check after an invalid access has already happened (see [35]), off-by-one errors in comparisons, mistakenly interpreting a negative signed number as a large unsigned number, 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 on stack objects, but we also identified several ones on heap objects, global objects, and to the `main()` function's arguments. A smaller number of bugs were caused by `NULL` dereferences that could also have been found without a bug finding tool. We only found 1 use-after-free error and 1 variadic argument error (where arguments did not match the format string).

ASan and Valgrind miss errors. We compiled the

[8] To calculate the LOC we used `cloc` which omits comments and newlines. Where it was possible to distinct subprojects or programs inside a project, we took the LOC of the subproject.

[9] Where we found multiple similar errors in a single program, we only counted them as one error if they were similar (e.g., both where buffer overflows caused by a missing NULL terminator).

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 in the omission of errors, we also compiled the programs with 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` linker flag for ASan.

Valgrind `-O0` and `-O3` found slightly more than half of the errors since Valgrind only reliably detects out-of-bounds accesses to the heap. It misses many of the out-of-bounds accesses to the stack and to globals. 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 some bugs can be found by compiling with either `-O0` or `-O3`, but not with both. ASan `-O0` detected 54 of the 62 errors that Safe Sulong found. Only 50 errors (a subset of those found with `-O0`) were also found with `-O3`, since in the other cases Clang optimized away bugs. From the 62 errors that Safe Sulong detected, 8 could neither be found by Valgrind nor by ASan with any optimization level (`-O0` and `-O3`).

Uninstrumented main arguments array (P4, P1). We argued that for tools that are based on low-level approaches it is not always obvious whether or not the analyzed programs contain uninstrumented native code or data. We found that neither ASan[10] nor Valgrind detect out-of-bounds accesses to the `main()` function's arguments, a bug that we found in three applications. Figure 12 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; this array is initialized irrespective of the `main()` function's signature [22]. A missing or wrong check might allow an attacker to exploit an out-of-bounds access to leak secrets contained in an environment variable. On our system, executing the program (without passing arguments) prints a key and value pair of an environment variable stored in `~/.bashrc`.

Missing interceptors (P1). ASan could not find two bugs due to missing or incomplete interceptors; Valgrind did not find them since 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 13). ASan failed to detect this bug since it lacked an interceptor for `strtok()`, which we consequently implemented.[11] We also found one error where the program passed an integer to `printf("%ld")`, where the format string specified a `long` (see Figure 14). Note that Clang detected the bug statically and printed a warning; however, ASan did not detect the error, because the interceptor for `printf()` only checks pointer arguments. Building and linking an instrumented version of the `libc` would have allowed ASan to detect these errors. However, we believe that most users use the libc in the standard precompiled form, where the errors are not found.

Backend compiler optimizations (P2). In the `isidigit()` example of Figure 5, we showed that even Clang `-O0` can optimize bugs away. We found another case where a bug was eliminated by the compiler when compiling with `-O0`, namely a global array out-of-bounds access, similar to the one shown in Figure 15. Clang statically detected the out-of-bounds access and printed a warning. However, Clang's front end did not yet optimize away the bug, so Safe Sulong was still able to detect it while executing the LLVM IR; only LLVM's back end optimized it 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. Arguable, a user could have found the bug through the compiler warning. However, as demonstrated by the `isdigit()` example, Clang -O0 sometimes optimizes bugs away without a warning.

```c
#include <stdio.h>

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

Figure 12: ASan does not detect out-of-bounds accesses to the main function.

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

Figure 13: The delimiter passed to `strtok()` is not `NULL`-terminated.

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

Figure 14: A wrong format specifier is used which causes an out-of-bounds read.

[10] See `https://github.com/google/sanitizers/issues/762`.

[11] See `https://github.com/google/sanitizers/issues/766` and `https://reviews.llvm.org/rL298650`

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

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

Figure 15: 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 16: A large number as user input causes a buffer overflow that can exceed ASan's redzone.

Overflowing the redzone (P3) As shown before, shadow memory approaches are inexact and cannot find all errors of a certain category. Safe Sulong found such a case in a program that reads a number and converts it to a string; Figure 16 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.

Missing variadic arguments In the projects that we evaluated, we only found a few implementations of variadic functions. However, we identified a missing argument to the variadic `printf()` `libc` function (see Figure 17). As in Figure 12, Clang detected the bug statically, since `printf()` is a well-known library function. However, the bug could also have occurred in an application-specific function, where Clang would not be able to detect it; similar format string vulnerabilities have been recently 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.

```
printf("%i not found in array.\r\n");
```

Figure 17: Missing argument to printf

4.2 Memory Costs

We evaluated the memory costs qualitatively, since quantitative measurements do not adequately account for the different characteristics of tools based on static compilers and on our dynamic compilation approach. In contrast to tools that are based on static compilers, Safe Sulong has additional memory costs since we run on top of a JVM which requires memory for internal data structures such as class metadata, memory-mapped files, code cache, and deoptimization metadata. JVMs use a managed heap, which is typically larger than the memory needed by the user application, since a larger heap typically improves run-time performance, because the garbage collector needs to run less frequently [2]. Additionally, the memory consumption of Java objects varies during the execution of the application, since compilation of methods can eliminate allocations or move them to the stack [31] (all Java objects are by default allocated on the heap). While static approaches produce an executable before program start, Safe Sulong parses the LLVM IR of the program and the libraries and converts them to Truffle ASTs at run time. After parsing, our in-memory representation of the program is larger than the size of executables, since the Truffle framework trades memory for peak performance by having AST nodes that use more memory than a naive bytecode interpreter or machine code. Allocations in the C programs have an additional constant overhead, since on most JVMs, Java objects contain an object header that typically requires 8 bytes. In contrast to that, the memory overhead of allocations for shadow memory approaches scales with the object size. While our approach is efficient for large objects, the constant overhead makes it inefficient for small ones. However, some of our data structures are generally larger than in native approaches since we store more data; for example, for pointers we store a reference to the pointee plus a field for the pointer offset, while a native pointer only stores the address of its pointee.

4.3 Start-up and Warm-up Costs

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

First, the LLVM IR interpreter has a noticeable start-up cost, which is the time from when the user starts the program until the program begins to run. We measured the start-up time using a `"Hello, World!"` program and `/usr/bin/time`. Figure 18 show the results of executing the program 100 times for each tool. Safe Sulong needs almost 700 ms to start up, in which the JVM initializes itself and starts Safe Sulong, which then has to parse the `libc` before calling the main function. Note that we could improve the start-up performance by lazily

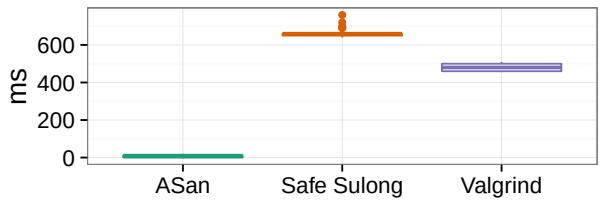

Figure 18: Start-up time in milliseconds

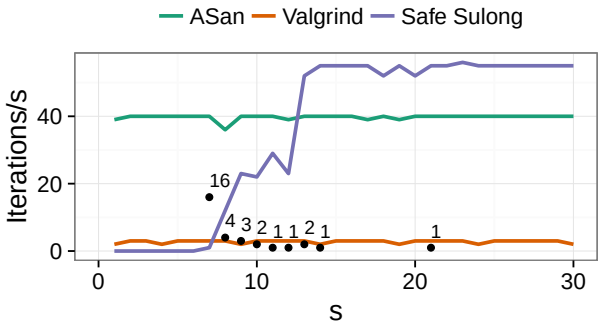

Figure 19: Warm-up time of the meteor benchmark. The x axis shows the time in seconds and the y axis the number of iterations the respective approach could run in the last second. The points illustrate the number of ASTs that Graal compiled up to that point.

parsing the libc, and by improving the performance of our parser. The start-up time of Safe Sulong on this program is worse than the start-up time 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 its peak performance. Figure 19 illustrates the warm-up times of ASan, Valgrind, and Safe Sulong on the meteor benchmark which is only 400 LOC large. 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 mostly attributed to the time that the program spends in the interpreter; only when the interpreter identifies hot functions, Graal compiles them to machine code. The curve shows a typical VM warm-up [1]. Only in second 7, Safe Sulong completed its first execution of the benchmark. During this time, Graal had compiled the 18 most important functions to machine code. Afterwards, it quickly became faster and executed more iterations per second than Valgrind (in second 7), and ASan (in second 13).

Note that the benchmark contains a call to `printf()` and to other `libc` functions, which Safe Sulong interprets and compiles to machine code during execution as well. Even after compilation, the program fails to immediately reach peak performance since we currently lack on-stack replacement, 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 [18, 12, 20]. However, our peak performance is higher than the one of existing tools as we demonstrate in Section 4.4. For ASan, we see that compile-time instrumentation approaches incur almost no warm-up costs, since checks are inserted during compilation and the run-time is initialized during start-up. Run-time approaches can either insert checks during start-up, or insert them on-demand while executing the program. Valgrind inserts them while executing the program; still, 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 [41, 40]. Applying this approach to Safe Sulong would allow us to obtain a standalone tool which no longer requires a JVM, has a smaller memory footprint, and lower warm-up costs, since the parser and other components would already be compiled when starting the program.

4.4 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, which contains smaller benchmarks (66-453 LOC) to compare the performance of different programming languages.[12] When we executed this suite's fastaredux benchmark with Safe Sulong, we discovered that a loop ran out-of-bounds, since summed-up probabilities did not add up to the value 1 to a rounding error. We reported and fixed the bug[13] and used the fixed version in our evaluation. Additionally, we included the popular whetstone benchmark.[14]

As baselines, we measured the performance of executables compiled by Clang with disabled optimizations (-O0) and enabled optimizations (-O3). Since our concern was to find as many errors as possible, we compiled the benchmarks using Clang -O0 for all bug finding

[12] `http://benchmarksgame.alioth.debian.org/`

[13] `https://alioth.debian.org/tracker/?func=detail&atid=413122&aid=315503&group_id=100815`

[14] `http://www.netlib.org/benchmark/whetstone.c`

tools, although Safe Sulong would also profit from compiler optimizations. Besides measuring 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 should therefore just demonstrate that programs under Safe Sulong achieve a peak performance that is good enough 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 20 shows box plots for the peak performance relative to Clang -O0 (lower is better). We excluded Valgrind from the plots, since it runs `10`× to `58`× slower than Clang -O0 on 5 benchmarks. Its slowdown is lowest on `spectralnorm`, `fasta`, and `fannkuchredux` with a slowdown of `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 is due to the fact that binarytrees is allocation-intensive, which suggests that current bug finding approaches cannot deal well with allocation-intensive benchmarks. On this benchmark, Safe Sulong is only 1.7× slower than Clang -O0. In almost all benchmarks Safe Sulong is faster than `ASan -O0`; they are only on-par on `fastaredux`. Safe Sulong is mostly faster than `Clang -O0`, except on the `fastaredux` and `nbody` benchmarks. On `fannkuchredux` and `mandelbrot` Safe Sulong is even on-par with Clang -O3. Safe Sulong exhibits the worst performance on `fastaredux`, where it is `2.5`× slower than Clang -O0. In future work, we plan to further reduce Safe Sulong's overhead.

5 Limitations

Native interoperability. Interoperability with precompiled binaries is a double-edged sword. It is necessary to execute closed-source libraries, but it results in overlooked bugs as we have demonstrated in our findings. What sets Safe Sulong apart from state-of-the-art shadow memory approaches (which are inexact by design) is that our approach aims to find all errors of a category. To maintain this property, Safe Sulong does not provide a native function interface. Safe Sulong's biggest drawback is that this makes it inapplicable to programs that require this interoperability. We want to address this issue in future work by using *binary translation* to convert binaries to LLVM IR that can be executed by Safe Sulong. There are already translators that can convert binary code to LLVM IR; MC-Semantics [9], REVAMB [8], QEMU [4] support x86, and LLBT [30] supports the translation of ARM code.

Warmup time. As discussed in Section 4.3, 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. To get close to 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. As a long-term solution to reduce warm-up time, the Graal project is experimenting with ahead-of-time compilation of the interpreter and the JIT compiler. Note that the JIT compilation approach allows Safe Sulong to have a 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. Following, we also require a program to not violate the type rules of the C standard (known as type punning). Since type violations are still quite 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 [23]. Currently, Safe Sulong lacks support for many of such patterns; for example, it lacks support 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 example, we could allow users to store integers in the `offset` field of `Address`. However, note that Safe Sulong will never be able to support all non-standard C code, that compiles when using static compilers.

6 Future Work

Completeness. Safe Sulong is a prototype and is currently not able to execute all C programs. While our interpreter can execute most LLVM IR instructions, it also needs to provide implementations for the system libraries (most importantly the libc). We implemented most libc functions in C, but also needed to expose certain functionality in our Java runtime (similar to OS

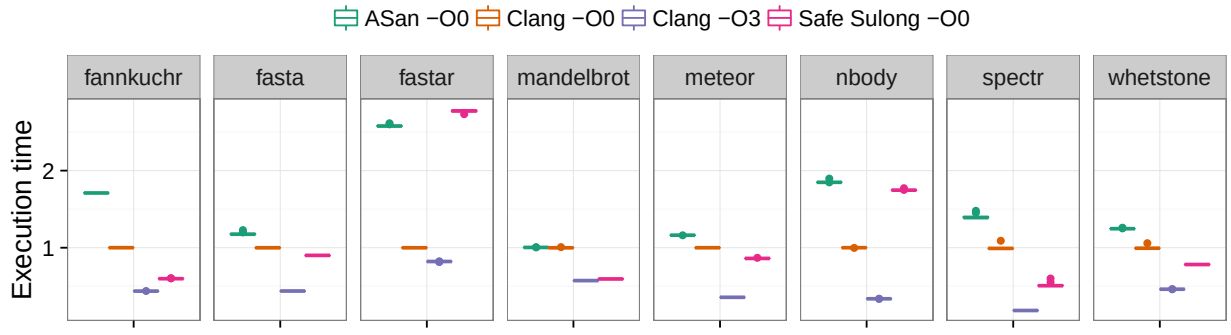

Figure 20: Execution times relative to Clang -O0 (peak performance, lower is better)

system calls). 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. While we have not yet stumbled on any essential problems, much engineering effort is required to implement the missing functionality. Note that we already support a small subset of x86 inline assembly by constructing ASTs for assembly operations.

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 not needed anymore, 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 manually freed to print an error in case it has not. This approach is comparable with Purify, which uses a user-callable conservative garbage collector [15].

Detection of uninitialized memory reads. Valgrind, Purify, Dr. Memory, and MSan [32] (but not ASan) detect reads of uninitialized memory. Currently, we zero-initialize all data structures and do not detect such bugs. We have already successfully investigated combining our approach with a shadow memory approach to yield an *exact shadow memory* tool. Our idea is to store shadow memory inside each object, so that it cannot be confused to belong to another object.

Replace Clang as a front end. As we have demonstrated, Clang (and other C compilers) can optimize away code with undefined behavior even with disabled optimizations. 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 want to implement a C front end that does not perform any optimizations based on undefined behavior.

7 Conclusion

In this paper, we presented a novel bug finding tool for C programs that is based on abstraction from 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 the performance of Clang `-O0`, and even Clang `-O3` in some cases.

References

[1] Barrett, E., Bolz, C. F., Killick, R., Knight, V., Mount, S., and Tratt, L. Virtual machine warmup blows hot and cold. *ICOOOLPS 2016* (2016).

[2] Blackburn, S. M., Cheng, P., and McKinley, K. S. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (2004), 25–36.

[3] Bruening, D., and Zhao, Q. Practical memory checking with dr. memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2011), IEEE Computer Society, pp. 213–223.

[4] Chipounov, V., and Candea, G. Dynamically translating x86 to llvm using qemu. Tech. rep., École polytechnique fédérale de Lausanne, 2010.

[5] Clifford, D., Payer, H., Stanton, M., and Titzer, B. L. Memento mori: Dynamic allocation-site-based optimizations. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 105–117.

[6] Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., and Lokier, J. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (2001), vol. 91, Washington, DC.

[7] COWAN, C., WAGLE, F., PU, C., BEATTIE, S., AND WALPOLE, J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings* (2000), vol. 2, IEEE, pp. 119–129.

[8] DI FEDERICO, A., AND AGOSTA, G. 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* (2016), ACM, p. 17.

[9] DINABURG, A., AND RUEF, A. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada* (2014).

[10] EIGLER, F. C. Mudflap: Pointer use checking for c/c+. *Proceedings of the First Annual GCC Developers Summit* (2003), 57–70.

[11] EVANS, D., AND LAROCHELLE, D. Improving security using extensible lightweight static analysis. *IEEE software 19*, 1 (2002), 42–51.

[12] FINK, S. J., AND QIAN, F. 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* (2003), CGO '03, pp. 241–252.

[13] GODEFROID, P., LEVIN, M. Y., MOLNAR, D. A., ET AL. Automated whitebox fuzz testing. In *NDSS* (2008), vol. 8, pp. 151–166.

[14] GRIMMER, M., SCHATZ, R., SEATON, C., WÜRTHINGER, T., AND MÖSSENBÖCK, H. Memory-safe execution of c on a java vm. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security* (2015), ACM, pp. 16–27.

[15] HASTINGS, R., AND JOYCE, B. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference* (1991), Citeseer.

[16] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91* (1991), pp. 21–38.

[17] HÖLZLE, U., CHAMBERS, C., AND UNGAR, D. Debugging optimized code with dynamic deoptimization. In *ACM Sigplan Notices* (1992), vol. 27, pp. 32–43.

[18] HÖLZLE, U., AND UNGAR, D. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM SIGPLAN Notices* (1994), vol. 29, ACM, pp. 326–336.

[19] HOLZMANN, G. J. Static source code checking for user-defined properties. In *Proc. IDPT* (2002), vol. 2.

[20] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO) 5*, 1 (2008), 7.

[21] LATTNER, C., AND ADVE, V. Llvm: a compilation framework for lifelong program analysis transformation. In *CGO 2004* (March 2004), pp. 75–86.

[22] MATZ, M., HUBICKA, J., JAEGER, A., AND MITCHELL, M. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0 99* (2013).

[23] MEMARIAN, K., MATTHIESEN, J., LINGARD, J., NIENHUIS, K., CHISNALL, D., WATSON, R. N., AND SEWELL, P. Into the depths of c: elaborating the de facto standards. In *PLDI 2016* (2016), pp. 1–15.

[24] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices* (2007), vol. 42, ACM, pp. 89–100.

[25] RIGGER, M., GRIMMER, M., WIMMER, C., WÜRTHINGER, T., AND MÖSSENBÖCK, H. 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* (New York, NY, USA, 2016), VMIL 2016, ACM, pp. 6–15.

[26] ROSE, J. Jep 243: Java-level jvm compiler interface, 2014.

[27] SANS. Cwe/sans top 25 most dangerous software errors, 2011.

[28] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference* (2012), pp. 309–318.

[29] SEWARD, J., AND NETHERCOTE, N. Using valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track* (2005), pp. 17–30.

[30] SHEN, B.-Y., CHEN, J.-Y., HSU, W.-C., AND YANG, W. Llbt: an llvm-based static binary translator. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems* (2012), ACM, pp. 51–60.

[31] STADLER, L., WÜRTHINGER, T., AND MÖSSENBÖCK, H. Partial escape analysis and scalar replacement for java. In *Proceedings of CGO '14* (2014), pp. 165–174.

[32] STEPANOV, E., AND SEREBRYANY, K. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on* (2015), IEEE, pp. 46–55.

[33] VAN DER VEEN, V., DUTT SHARMA, N., CAVALLARO, L., AND BOS, H. Memory errors: The past, the present, and the future. In *Proceedings of RAID'12* (2012), pp. 86–106.

[34] WANG, X., CHEN, H., CHEUNG, A., JIA, Z., ZELDOVICH, N., AND KAASHOEK, M. F. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems* (2012), ACM, p. 9.

[35] WANG, X., ZELDOVICH, N., KAASHOEK, M. F., AND SOLAR-LEZAMA, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 260–275.

[36] WARREN, H. S. *Hacker's delight*. Pearson Education, 2013.

[37] WIMMER, C., AND WÜRTHINGER, T. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (2012), SPLASH '12, pp. 13–14.

[38] WÖSS, A., WIRTH, C., BONETTA, D., SEATON, C., HUMER, C., AND MÖSSENBÖCK, H. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools* (2014), ACM, pp. 133–144.

[39] WÜRTHINGER, T., WIMMER, C., AND MÖSSENBÖCK, H. 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* (2007), ACM, pp. 125–133.

[40] WÜRTHINGER, T., WIMMER, C., WÖSS, A., STADLER, L., DUBOSCQ, G., HUMER, C., RICHARDS, G., SIMON, D., AND WOLCZKO, M. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2013), Onward! 2013, ACM, pp. 187–204.

[41] WÜRTHINGER, T., WIMMER, CHRISTIAN HUMER, C., WÖSS, A., STADLER, L., SEATON, C., DUBOSCQ, G., SIMON, D., AND GRIMMER, M. Practical partial evaluation for high-performance dynamic language runtimes. In *PLDI (Tentatively accepted)* (2017).