Lenient Execution of C on a Java Virtual Machine

or: How I Learned to Stop Worrying and Run the Code

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

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

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

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

ABSTRACT

Most C programs do not conform strictly to the C standard, and often show *undefined behaviors*, for instance, in the case of signed integer overflow. When compiled by non-optimizing compilers, such programs often behave as the programmer intended. However, optimizing compilers may exploit undefined semantics to achieve more aggressive optimizations, possibly breaking the code in the process. Analysis tools can help to find and fix such issues. Alternatively, a C dialect could be defined in which clear semantics are specified for frequently occurring program patterns with otherwise undefined behaviors. In this paper, we present *Lenient C*, a C dialect that specifies semantics for behaviors left open for interpretation in the standard. Specifying additional semantics enables programmers to make safe use of otherwise undefined patterns. We demonstrate how we implemented the dialect in *Safe Sulong*, a C interpreter with a dynamic compiler that runs on the JVM.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Imperative languages**; **Interpreters**; *Translator writing systems and compiler generators*;

KEYWORDS

C, Undefined Behavior, Sulong

ACM Reference format:
Manuel Rigger, Roland Schatz, Matthias Grimmer, and Hanspeter Mössenböck. 2017. Lenient Execution of C on a Java Virtual Machine. In *Proceedings of ManLang 2017, Prague, Czech Republic, September 27–29, 2017,* 13 pages.
https://doi.org/10.1145/3132190.3132204

C is a language that leaves many semantic details open. For example, it does not define what should happen in the case of an out-of-bounds access to an array, when a signed integer overflow occurs, or when a type rule is violated. In such cases, not only does the invalid operation yield an undefined result, but — according to the C standard — the whole program is rendered invalid. As compilers become more powerful, an increasing number of programs break because *undefined behavior* allows more aggressive optimization and may lead to machine code that does not behave as expected. Consequently, programs that rely on undefined behavior may introduce bugs that are hard to find, can result in security vulnerabilities, or remain as time bombs in the code that explode after compiler updates [34, 51, 52].

While bug-finding tools help programmers to find and eliminate undefined behavior in C programs, the majority of C programs will still contain at least some non-portable code. This includes unspecified and implementation-defined patterns, which do not render the whole program invalid, but can cause unexpected results. Specifying a more lenient C dialect that better suits programmers' needs and addresses common programming mistakes has been suggested to remedy this [2, 8, 13]. Such a dialect would extend the C standard and assign semantics to otherwise non-portable behavior; it would make C safe in the sense of Felleisen & Krishnamurthi [15]. We devised *Lenient C*, a C dialect which, for example,

- assumes allocated memory to be initialized,
- assumes automatic memory management,
- allows dereferencing pointers using an incorrect type,
- defines corner cases of arithmetic operators,
- and allows pointers to different objects to be compared.

Every C program is also a Lenient C program. However, although Lenient C programs are source-compatible with C programs, they are not guaranteed to work correctly when compiled by C compilers.

We implemented Lenient C in *Safe Sulong* [37], an interpreter with a dynamic compiler that executes C code on the JVM. Per default, Safe Sulong aborts execution when it detects undefined behavior. As part of this work, we added an option to assume the Lenient C dialect when executing a program to support execution of incompliant C programs. Implementing Lenient C in Safe Sulong allowed us to validate the approach without having to change a static compiler. Although a managed runtime is not a typical environment for running C, it is a good experimentation platform because such runtimes typically execute memory-safe high-level languages that provide many features that we also want for C, for example, automatic garbage collection and zero-initialized memory. In this context, Lenient C is a dialect that is suited to execution on the JVM, .NET, or a VM written in RPython [39]. If Lenient C turns out to be useful in managed runtimes, a subset of its rules might also be incorporated into static compilers.

We assume that implementations of Lenient C in managed runtimes represent C objects (primitives, structs, arrays, etc.) using

ManLang 2017, September 27–29, 2017, Prague, Czech Republic
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of ManLang 2017, September 27–29, 2017*, https://doi.org/10.1145/3132190.3132204.

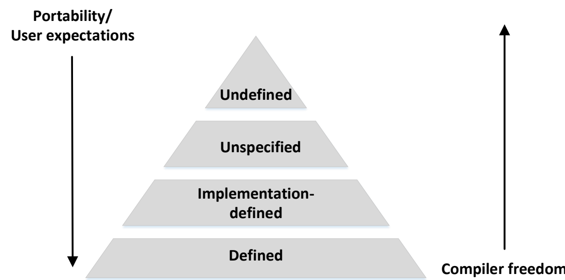

Figure 1: The Pyramid of "Undefinedness"

an object hierarchy, and that pointers to other objects are implemented using managed references. This approach enables use of a GC, which would not be possible if a large byte array were used to represent C allocations [25]. In terms of language semantics, we focused on implementing operations in a way most programmers would expect. Undefined corner cases in arithmetic operations behave similarly to Java's arithmetic operations, which also resemble the behavior of AMD64. This paper makes the following contributions:

- a relaxed C dialect called Lenient C that gives semantics to undefined behavior and is suitable for execution on a JVM and other managed runtimes;
- an implementation of this dialect in Safe Sulong — an interpreter written in Java;
- a comparison of Lenient C with the Friendly C proposal and the anti-patterns listed in the *SEI CERT C Coding Standard*.

1 BACKGROUND

1.1 Looseness in the C Standard

The main focus of C is on performance, so the C standard defines only the language's core functionality while leaving many corner cases undefined (to different degrees, see below). For example, unlike higher-level-languages, such as Java and C#, C does not require local variables to be initialized, and reading from uninitialized variables can yield undefined behavior [43].[1] Avoiding storage initialization results in speed-ups of a few percent [27]. As another example, 32-bit shifts are implemented differently across CPUs; the shift amount is truncated to 5 bits and 6 bits on X86 and PowerPC architectures, respectively [22]. In C, shifting an integer by a shift amount greater than the bit width of the integer type is undefined, which allows the CPU's shift instructions to be used directly on both platforms.

The C standard provides different degrees of looseness, as illustrated by the pyramid of "undefinedness" in Figure 1. Programmers usually want their programs to be *strictly conforming*; that is, they only rely on *defined* semantics. Strictly-conforming programs exhibit identical behavior across platforms and compilers (C11 4 §5). Layers above "defined" incrementally provide freedom to compilers, which limits program portability and results in compiled code that often does not behave as the user expected [51, 52]. *Implementation-defined* behavior allows free implementation of a specific behavior that needs to be documented. Examples of implementation-defined behavior are casts between pointers that underlie different alignment requirements across platforms. *Unspecified* behavior, unlike implementation-defined behavior, does not require the behavior to be documented. Since it is allowed to vary per instance, unspecified behavior typically includes cases in which compilers do not enforce a specific behavior. An example is using an unspecified value, which can, for example, be produced by reading padding bytes of a struct (C11 6.2.6.1 §6). Another example is the order of argument evaluation in function calls (C11 6.5.2.2 §10). *Undefined* behavior provides the weakest guarantees; the compiler is not bound to implement any specific behavior. A single occurrence of undefined behavior renders the whole program invalid. The tacit agreement between compiler writers seems to be that no meaningful code needs to be produced for undefined behavior, and that compiler optimizations can ignore it to produce efficient code [13]. Consequently, the current consensus among researchers and industry is that C programs should avoid undefined behavior in all instances, and a plethora of tools detect undefined behavior so that the programmer can eliminate it [e.g., 4, 5, 14, 17, 20, 31, 44, 46, 50]. Examples of undefined behavior are `NULL` dereferences, out-of-bounds accesses, integer overflows, and overflows in the shift amount.

1.2 Problems with Undefined Behavior

While implementing Safe Sulong, we found that most C programs exhibit undefined behavior and other portability issues. This is consistent with previous findings. For example, six out of nine SPEC CINT 2006 benchmarks induce undefined behavior in integer operations alone [11].

It is not surprising that the majority of C programs is not portable. On the surface, the limited number of language constructs makes it easy to approach the language; its proximity to the underlying machine allows examining and understanding how it is compiled. However, C's semantics are intricate; the informative Annex J on portability issues alone comprises more than twenty pages. As stated by Ertl, "[p]rogrammers are usually not language lawyers" [13] and rarely have a thorough understanding of the C standard. This is even true for experts, as confirmed by the *Cerberus* survey, which showed that C experts rely, for example, on being able to compare pointers to different objects using relational operators, which is clearly forbidden by the C standard [26].

Furthermore, much effort is required to write C code that cannot induce undefined behavior. For example, Figure 2 shows an addition that cannot overflow (which would induce undefined behavior). Such safe code is awkward to program and defeats C's original goal of defining its semantics such that efficient code can be produced across platforms.

In general, code that induces undefined behavior cannot always be detected at compile time. For example, adding two numbers is defined as long as no integer overflows happen. It is also seldom a problem when the program is compiled with optimizations turned off (e.g., with flag `-O0`). However, it is widely known that compilers perform optimizations at higher optimization levels that

[1] Note that reading an uninitialized variable produces an indeterminate value, which — depending on the type — can be a trap representation or an unspecified value.

```
signed int sum(signed int si_a, signed int si_b) {
  if (((si_b > 0) && (si_a > (INT_MAX - si_b))) ||
      ((si_b < 0) && (si_a < (INT_MIN - si_b)))) {
    /* Handle error */
  } else {
    return si_a + si_b;
  }
}
```

Figure 2: Avoiding overflows in addition [42]

cause programs to behave incorrectly if they induce undefined behavior [22, 34]. For instance, about 40% of the Debian packages contain unstable code that compilers optimize away at higher optimization levels, often changing the semantics because compilers exploit incorrect checks or undefined behavior in the proximity of checks [52]. This is worrisome, since optimizing away checks that the user deliberately inserted is likely to create vulnerabilities in the code [2, 12, 52]. Finally, code can be seen as a *time bomb* [34]. Increasingly powerful compiler optimization can cause programs to break with compiler updates; if code that induces undefined behavior does not break now, it might do so in the future [22].

1.3 Calls for a Lenient C

One strategy to tackle portability issues is to detect them and to fix the relevant code. To this end, a host of static and dynamic tools enable programmers to detect memory problems [4, 31, 44, 46], integer overflows [5, 50], type problems [20], and other portability issues in programs [14, 17]. Another approach is to educate programmers and inform them about common portability pitfalls in C. The most comprehensive guide to avoiding portability issues is the *SEI CERT C Coding Standard* [42], which documents best practices for C.

Due to the portability issues described, a more lenient dialect of C has been called for (see below). Rather than consider common patterns that are in conflict with the C standard as portability problems, it would explicitly support them by assigning semantics to them in a way that programmers would expect from the current C standard. Most code that would execute correctly at `-O0`, even if it induces undefined behavior, would also correctly execute with this dialect. For unrecoverable errors, it would require implementations to trap (i.e., abort the program). This dialect would be source-compatible with standard C: every program that compiles according to the C standard would also compile with this dialect. Consequently, such an effort would be different from safer, C-like languages such as *Polymorphic C* [45], *Cyclone* [18], and *CCured* [30], which require porting or annotating C programs.

Three notable proposals for such a safer C dialect can be found. Bernstein called for a "boring C compiler" [2] that would prioritize predictability over performance and could be used for cryptographic libraries. Such a compiler would commit to a specified behavior for undefined, implementation-defined, and unspecified semantics. The proposal did not contain concrete suggestions, except that uninitialized variables should be initialized to zero. A second proposal, a "Friendly Dialect of C", was formulated by Cuoq, Flatt, and Regehr [8]. The *Friendly C* dialect is similar to C, except that it replaces occurrences of undefined behavior in the standard either with defined behavior or with unspecified results (which do not render the whole program or execution invalid). Friendly C specifies 14 changes to the language, addressing some of the most important issues, but was meant to trigger discussion and not to cover all the deficiencies of the language comprehensively. Eventually, Regehr [35] abandoned the proposal and concluded that too many variations to a Friendly C standard would be possible to have experts reach a consensus. Instead, he proposed that reaching a consensus should be skipped in favor of developing a friendly C dialect, which could be adopted by more compilers if used by a broader community. A third proposal, for a "C*" dialect in which operations at the language level directly correspond to machine level operations, was outlined by Ertl [13]. Ertl observed that the C standard gave leeway to implementations to efficiently map C constructs to the hardware. However, he noted that compiler maintainers diverge from this philosophy and implement optimizations that go against the programmer's intent, by deriving facts from undefined behavior that enable more aggressive optimizations. Ertl believes that C programmers unknowingly write programs that target the C* dialect because they are not sufficiently familiar with the C rules. According to him, the effort needed to convert C* to C programs, however, would have a poor cost-benefit ratio, given that programmers could hand-tune the C code.

2 LENIENT C

We present *Lenient C*, a C dialect that assigns semantics to behavior in the C11 standard that is otherwise undefined, unspecified, or implementation-defined. Table 1 presents the rules that supersede those of the C11 standard and specify Lenient C.

A previous study categorized undefined behavior according to whether it involved the core language, preprocessing phases, or library functions [17]. We restrict Lenient C to the core language, and will consider extensions to it as part of future work, memory management functions being the only exception. We were primarily interested in undefined behavior that compilers cannot statically detect in all instances. Consequently, we disregarded problematic idioms such as writing-through consts [14], where an object with a const-qualified type is modified by casting it to a non-const-qualified type (C11 6.7.3 §6). We believe that increased research into compiler warnings and errors enables elimination of such bugs [47]. We also excluded undefined behavior caused by multiple modifications between sequence points [21] – which guarantee that all previous side effects have been performed – which includes expressions such as `i++ * i++` (C11 6.5 §2).

We created this dialect while working on the execution of C code on the JVM, using *Safe Sulong*, a C interpreter with a dynamic compiler. Per default, Safe Sulong aborts execution when it detects undefined behavior. However, we found that most programs induce undefined behavior or exhibit other portability issues. As an alternative to fixing such programs, we added an option to execute programs assuming the less strict Lenient C dialect.

Lenient C was inspired by Friendly C; additionally, we sought to support many anti-patterns that are described in the *SEI CERT C Coding Standard*, as they reflect non-portable idioms on which programmers rely. While the dialect can be implemented by static compilers, Lenient C programs are best suited to execution in a

ID	Lenient C	SEI CERT	Friendly C
General			
G1	Writes to global variables, traps, I/O, and program termination are considered to be side effects.		6, 13
G2	Externally visible side effects must not be reordered or removed.		6, 13
G3	Signed numbers must be represented in two's complement.	INT16-C	
G4	Variable-length arrays that are initialized to a length smaller or equal to zero must produce a trap.	ARR32-C	
Memory Management			
M1	Dead dynamic memory must eventually be reclaimed, even if it is not manually freed.	MEM31-C	
M2	Objects can be used as long as they are referenced by pointers.	MEM30-C	1
M3	Calling free() on invalid pointers must have no effect.	MEM34-C	
Accesses to Memory			
A1	Reading uninitialized memory must behave as if the memory were initialized with zeroes.	EXP33-C	8
A2	Reading struct padding must behave as if it were initialized with zeroes.	EXP42-C	
A3	Dereferences of `NULL` pointers must abort the program.	EXP34-C	4
A4	Out-of-bounds accesses must cause the program to abort.	MEM35-C	4
A5	A pointer must be dereferenceable using any type.	EXP39-C	10
Pointer Arithmetic			
P1	Computing pointers that do not point to an object must be permitted.	ARR30-C, ARR38-C, ARR39-C	9
P2	Overflows on pointers must have wraparound semantics.		9
P3	Comparisons of pointers to different objects must give consistent results based on an ordering of objects.	ARR36-C	
P4	Pointer arithmetic must work not only for pointers to arrays, but also for pointers to any type.	ARR37-C	
Conversions			
C1	Arbitrary pointer casts must be permitted while maintaining a valid pointer to the object.	MEM36-C, EXP36-C	10, 13
C2	Converting a pointer to an integer must produce an integer that, when compared with another pointer-derived integer, yields the same result as if the comparison operation were performed on the pointers.	INT36-C, ARR39-C	
C3	When casting a floating-point number to a floating point-number, or when casting between integers and floating-point numbers, a value not representable in the target type yields an unspecified value.	FLP34-C	
Functions			
F1	Non-void functions that do not return a result implicitly return zero.	MSC37-C	14
F3	A function call must trap when the actual number of arguments does not match the number of arguments in the function declaration.	EXP37-C, DCL40-C	
Integer Operations			
I1	Signed integer overflow must have wraparound semantics.	INT32-C	2
I2	The second argument of left- and right-shifts must be reduced to the value modulo the size of the type and must be treated as an unsigned value.	INT34-C	3
I3	Signed right-shift must maintain the signedness of the value; that is, it must implement an arithmetic shift.		
I4	If the second operand of a modulo or division operation is 0, the operation must trap.	INT33-C	5
I5	Like signed right-shifts, bit operations on signed integers must produce the same bit representation as if the value were cast to an unsigned integer.		7

Table 1: The rules of Lenient C compared to those of the *SEI CERT C Coding Standard* and Friendly C

managed environment. In other words, Lenient C makes some assumptions that hold for managed runtimes such as the JVM or .NET, but typically not for static compilers, such as LLVM and GCC, that compile C code to an executable. For example, Lenient C assumes automatic memory management. Although garbage collectors (GCs) exist that can be compiled into applications [3, 33], they are not commonly used. Nonetheless, we believe that many of Lenient C's rules might also inspire their implementation in static compilers.

In the following sections, we describe how we implemented the Lenient C dialect in Safe Sulong, and expand on its design decisions. Section 3 describes an object hierarchy suitable for implementing Lenient C in object-oriented languages. Section 4 addresses memory management and expands on Lenient C's memory management

and memory error handling requirements. Section 5 examines operations on pointers, and Section 6 discusses how Lenient C envisages the implementation of arithmetic operations.

3 REPRESENTING C OBJECTS

All our C objects are represented using classes that inherit from a `ManagedObject` base class.[2] We found such an approach to be more idiomatic than using a single array of bytes to represent memory. Subclasses comprise integer, floating point, struct, union, array, pointer, and function pointer types. For example, we represent the C `float` type as a `Float` subclass. To denote values, we use Haskell-style type constructors. A `float` value `3.0` is thus expressed as `Float(3.0)`.

The `ManagedObject` class specifies methods for reading from and writing to objects that the subclasses need to implement. The read operation is expressed as a method `object.read(type, offset)` that reads a specific type from an object at a given offset. Note that the `offset` is always measured in bytes. For example, reading a float at byte offset 4 from an object is expressed as `object.read(Float, 4)`. The write method is expressed as a method `object.write(type, offset, value)`. The C standard requires that every object can

- be treated as a sequence of `unsigned chars`, so every subclass must implement at least one method that can read and write the `char` type (C11 6.2.6.1 §4),
- and can be read using the type of an object, so, for example, an `int` object must implement read and write methods for `int`.

Additionally, we allow objects to be treated using other types, by concatenating their byte representations (see Section 5.4).

3.1 Integer and Floating-Point Types

Safe Sulong represents primitive types as Java wrapper classes. In subsequent examples, we assume an LP64 model in which an `int` has 32 bits, a `long` 64 bits, and a pointer 64 bits. However, our architecture also works for other 64-bit and 32-bit models; we will point out differences that influence the implementation at the corresponding places in the text.

For the C types `bool`, `char`, `short`, `int`, and `long` we use wrapped Java primitive types. For example, an `int` in C corresponds to a 32-bit integer in LP64, and we map it to a Java class `I32` that holds a Java `int`. Note that we do not need separate types for signed and unsigned integers; for example, we represent both `int` and `unsigned int` using a Java `int`. However, we need to provide both signed and unsigned operations (see Section 6).

We also represent `float` and `double` types using wrapped Java equivalents. C has a `long double` data type that is represented as an 80-bit floating-point type on AMD64. Since this data type does not exist in Java, we provide a custom implementation that emulates the behavior of 80-bit floats.[3]

[2] Safe Sulong interprets LLVM IR [23], which is a RISC-like intermediate representation, and not C code. LLVM IR also contains other integer types (e.g., I33 and I48) that we map to a wrapped `byte` array.

[3] Emulating 80-bit floats is inefficient and error-prone. As part of future work, we plan to provide a more efficient implementation of this type. However, we found that only few C programs rely on `long doubles`.

```
int main() {
    int val, arr[3];
    int *ptr1 = &val;        // (val, 0)
    int *ptr2 = &arr[2];     // (arr, 8)
    int *ptr4 = 0;           // (NULL, 0)
}
```

Figure 3: Various pointer tuples

3.2 Pointers and Function Pointers

We implement pointers using a class `Address`. `Address` has two fields: a `ManagedObject` field `pointee` that refers to its pointee, and an integer `offset` that denotes the offset within the object. The offset must be large enough to hold an integer with the same bit width as a pointer; assuming LP64, it is 64 bits wide. We denote a pointer-tuple by `(pointee, offset)`. The idea of representing pointers as a tuple is not new; for example, formal C models [19, 24] and also previous implementations of C on the JVM used such a representation [10, 16]. Figure 3 shows tuples for three different pointers. `ptr1` points to the start of an `int`; the offset is `0`. `ptr2` points to the second element of an integer array; the offset is 8 (`2 * sizeof(int)`). `ptr3` is a `NULL` pointer, which is obtainable by an integer constant `0`. The C standard specifies that `NULL` is guaranteed to be unequal to any pointer that points to a function or object (C11 6.3.2.3). We implement the `NULL` constant by an `Address` that has a `null` pointee and an `offset` of 0. Note that Section 5.1 gives a detailed account of pointer arithmetic.

To represent function pointers, we use a class that comprises a wrapped `long` that represents a *function ID*. For every parsed function, a unique ID starting from 1 is assigned. An ID of 0 represents a `NULL` function pointer. For calls, this ID is used to locate the executable representation of the function. Note that forgotten return statements in non-void functions induce undefined behavior (C11 6.9.1 §12). To address this, Safe Sulong implicitly returns a zero value of the return type when control reaches the end of the function. Note that another error class is when a function call supplies a wrong number of arguments, for which Lenient C requires the function call to trap.

3.3 Arrays

We represent C arrays by means of Java classes that wrap Java arrays. Primitive C arrays are represented by primitive Java arrays. For example, the type `int[]` is represented as a Java `int` array. We represent other C arrays using Java arrays that have a `ManagedObject` subtype as their element type. For example, we represent C pointer arrays as Java `Address` arrays. In our type hierarchy, arrays and structs are nested objects, which the `read` and `write` operations must take into account. Consequently, a given `offset` value must be decomposed to select the array element and then the offset within that element. For example, to read a byte from an `I32Array`, the `I8` read operation computes the value as the right-most byte taken from `values[offset / 4] >> (8 * (offset % 4))`. The division selects the array member, and the modulo the byte inside the integer.

```
struct {
    int a;
    long b;
} t;

char* val = (char*) &t;
val[9] = 1;
```

Figure 4: Writing a char into a struct member

3.4 Structs and Unions

Java lacks a struct type. We represent structs using a map [55] that contains `ManagedObjects`. A struct member can be accessed using an operation `getfield(offset)` that returns a tuple `(object, offset′)`.[4] The `object` denotes the member stored at the byte position `offset`, and `offset′` denotes the offset relative to the start of this member. Figure 4 shows an example. Note that the struct has a size of 16 bytes, where the stored `int` takes up 4 bytes, the padding values after the `int` 4 bytes, and the `long` 8 bytes. To write a byte at offset 9, Safe Sulong first selects the member using `getfield(9)`; the tuple returned is `(I64(0), 1)`. It then writes the value to the selected member object using `object.write(I8, 1)`. The read operation looks similar.

Safe Sulong also takes into account padding bytes, whose values are not specified by the standard (C11 6.2.6.1 §6). It initializes such bytes with a sequence of `I8(0)`. This allows programmers, for example, to compare structs using byte-by-byte comparison. Note that an alternative way of representing structs would be to use classes that represent struct members by fields. In a source-to-source transformation approach, these classes could be generated when compiling the program [25]. In interpreters, this would be more complicated because they would have to generate Java bytecode at run time.

In this object hierarchy, unions are structs with only one field. Unions allow programmers to view a memory region under different types. When reading a value from a union using a type that is different from the type that was last used when storing to this union, the standard requires that the union be represented in the new type (C11 6.5.2.3 §3). To account for this, we allocate a union with a sub-type that reflects the most general member type: when aliasing primitive values and pointers, we select `Addresses` or arrays of `Addresses`, since integers and floating-point numbers can be stored in the `offset` field of an address. As an alternative to using a single type, a map operation `writefield(offset, object)` could be introduced that replaces an existing object at the given offset to store a member with a different type. Such an approach would resemble *tagged unions*, which are, for example, used by precise GCs for C [33].

4 MEMORY MANAGEMENT

Two of our main concerns are how to implement memory management for C and how to handle memory errors. Allocating stack objects and global objects is straightforward, since their type is known. We map such allocations to one of the types presented in Section 3. Variable-length array declarations that have a negative or zero size induce undefined behavior (C11 6.7.6.2 §5). We trap in such cases, which corresponds to Java's default behavior when the size is negative (we still have to explicitly check for zero). 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 operation (i.e., when the type of the object becomes known).[5] Another approach to addressing untyped heap allocations would be to determine a type using static analysis [20].

4.1 Uninitialized Memory

There is no clear guidance on how a program should behave when it reads from uninitialized storage, an action which can induce undefined behavior [43]. There are two contradictory use cases, one of which we must support in our lenient execution model.

The first use case is that some programs purposefully read from uninitialized memory to create entropy. The entropy originates from previously allocated memory; uninitialized stack reads can read previous activation frames, while uninitialized heap reads can read malloced and freed heap memory. This pattern is problematic, and commonly used bug-finding tools such as Valgrind [31] and MSan [46] report it as a program error. Another issue is that reads to uninitialized memory make applications prone to information-leak attacks [27]. While allowing a program to read stale values could be dangerous, initializing all data structures with random values (to create entropy) would be overkill.

The second scenario is that programmers read uninitialized storage by accident. When executing programs with Safe Sulong, we found a number of programs that forgot to initialize memory or assumed that it was zero-initialized. Those programs worked correctly when uninitialized reads returned zero, which was suggested by Bernstein [2]. Zero-initialization is also supported by SafeInit [27], a protection system for C/C++ programs. Like SafeInit, we decided to support the second scenario, as it does not obviously jeopardize system security. Our implementation initializes all values to zero (recursively for nested objects); primitives are initialized to zero values, while pointers are initialized to `NULL`. Note that this approach is close to Java's default behavior, which initializes fields with an `Object` type to zero values if they are not initialized explicitly.

4.2 Memory Leaks and Dangling Pointers

C requires programmers to manually manage heap memory: memory allocated by `malloc()` must be freed using `free()`. Forgetting to free an object causes a memory leak, which can impact performance and can lead to the application running out of memory. Since Safe Sulong runs on a JVM, the JVM's GC reclaims objects after they are no longer needed. Note that automatic memory management cannot easily be implemented for static compilers; hence, it is also not covered by Friendly C.

[4] Read operations typically always access the same struct field through its name in the source code. Consequently, we mitigate the map-lookup costs by caching the lookup result. Note that an index for a struct member is the same across struct objects, so reading the same struct field from different objects is also efficient.

[5] For efficiency, we propagate the type back to the allocation site, similarly to allocation mementos in V8 [7]. Subsequent calls to the allocation function directly allocate an object of the observed type.

A *dangling pointer* is a pointer whose pointee has exceeded its lifetime. Accessing such a pointer induces undefined behavior. There are two situations in which a dangling pointer can be created:

- A heap object is freed using `free()` (C11 7.22.3.3).
- A C object with automatic storage duration (i.e., a stack variable) exceeds its lifetime (C11 6.2.4 §2).

There is no use case for accessing dangling pointers; they are caused by errors in manual memory management. In our type hierarchy, we could detect such errors by setting automatic objects to `null` after leaving a function scope, and by letting `free()` calls set the data of a pointee to `null` [37]. However, since we strive for lenient execution, we do not set them to `null` and retain references to objects whose lifetimes are exceeded. Consequently, programs can access dangling pointers as if they were still alive. This is useful to execute programs which, for example, access a dangling pointer shortly after a `free()` call under the assumption that the memory has not yet been reallocated. Only when the program loses all references to a pointee does the GC reclaim the pointee's memory.

Not aborting execution on use-after-free errors is Lenient C's most controversial design decision, as many tools strive to find such errors [29, 49, 58]. Safe Sulong can be used to find such errors, when executing programs using its default mode (instead of assuming the Lenient C dialect).

4.3 Buffer Overflows and NULL Dereferences

Besides use-after-free errors and invalid free errors, also buffer overflows and `NULL` dereferences are of concern, as they induce undefined behavior. For buffer overflows, an out-of-bounds read could produce a predefined zero value. This would work well when a non-delimited string was passed to a function operating on it; when reading zero, the function would assume that it had reached the end of the string. However, we also found that some programs with out-of-bounds reads did not terminate when producing a zero value upon out-of-bounds reads. For example, the fasta-redux benchmark ran out of bounds while adding up floating-point values. Due to a rounding error, the number did not add up to 1.00, and the program only terminated when reading positive garbage values [36]. In general, this approach is known as failure-oblivious computing [40], which ignores out-of-bound writes and produces a sequence of predefined values to accommodate various scenarios. As there is no value sequence that works for all programs, we decided to trap on buffer overflows. This also corresponds to Java's default semantics. Since we represent C arrays and structs using Java arrays, Java automatically performs bounds checks on accesses. On most architectures, `NULL` dereferences produce traps and usually cause unrecoverable program errors. Consequently, Lenient C also traps on `NULL` dereferences.

5 POINTER OPERATIONS

Pointers and pointer arithmetic are the main difference between C and other higher-level languages such as Java and C#, which use managed references instead. Consequently, this section explains how Safe Sulong implements operations that involve pointers.

```
int main() {
    int arr[3] = {1, 2, 3};
    ptrdiff_t diff1 = &arr[3] - &arr[0];
    size_t diff2 = (size_t) &arr[3] - (size_t) &arr[0];
    printf("%td %ld\n", diff1, diff2); // prints 3 12
}
```

Figure 5: Computing the pointer difference

5.1 Pointer Arithmetic

Addition or subtraction of integers. The standard defines additions and subtractions where one operand is a pointer `P` and the other an integer `N` (C11 6.5.6). Such an operation yields a pointer with the same type as `P` which points `N` elements forward or backward, depending on whether the operation is an addition or subtraction. For example, `arr + 5` computes an address by taking the address of `arr` and incrementing it by five elements. In our hierarchy, such an address computation creates a new pointer based on the old pointee and an updated offset. We compute the pointer as a new tuple `(pointee, oldPointer.offset + sizeof(type) * N)`. For example, if `arr` was an `int`, we would compute the offset by `sizeof(int) * N`. Note that the standard defines these operators only for pointers to arrays (C11 6.5.6 §8), while Lenient C allows pointer arithmetic for pointers to any type.

Subtraction of two pointers. The standard defines that subtracting two pointers yields the difference of the subscripts of the two array elements (C11 6.5.6 §9). Figure 5 shows a code snippet that subtracts two pointers, where one points to the start and one to the end of an array; note that the standard requires a common pointee (or a pointer one beyond the last array element). We implement pointer subtraction by subtracting the two integer representations of the pointers (see Section 5.3). Note that it would be sufficient to subtract the two pointer offsets; however, this could lead to unexpected results for different `pointees` (which is undefined behavior) with the same `offset` since the difference would suggest a common pointee.

Pointer overflow. The C standard allows pointers to point only to an object or to one element after it (C11 6.5.6 §8). The latter is useful when iterating over an array in a loop using a pointer. Lenient C abolishes these restrictions: in Safe Sulong a pointer is, through the `offset` field, handled like an integer and is, for example, allowed to overflow. However, we prohibit dereferencing out-of-bounds pointers (see Section 4.3).

Pointer comparisons. Two pointers `a` and `b` can be compared using the same comparison operators as integers and floating-point numbers.

Implementing the equality operators (`==` and `!=`) is straightforward. For example, to determine equality for two pointers, we check whether they refer to the same pointee and have the same pointer offset. In Java, we implement the pointee comparison using `a.pointee == b.pointee`, which checks for object equality. If the expression yields `true`, we also compare the offset using `a.offset == b.offset`.

Implementing the relation operators (<, >, <=, and >=) is more difficult. The C standard defines these operators only for pointers to the same object or its subobjects (for structs and arrays); comparing two different objects yields undefined behavior (C11 6.5.8 §5). To

```c
void *memmove(void *dest, void const *src, size_t n) {
    char *dp = dest;
    char const *sp = src;
    if (dp < sp) {
        while (n-- > 0)  *dp++ = *sp++;
    } else {
        dp += n; sp += n;
        while (n-- > 0)  *--dp = *--sp;
    }
    return dest;
}
```

Figure 6: Non-portable implementation of `memmove` (adapted from [48])

implement standard-compliant behavior, comparing the pointer offset would be sufficient; for example, to implement <, we could compare `a.offset < b.offset`. However, we found that programs often compare pointers to different objects. For example, Figure 6 shows a naive implementation of `memmove` that potentially compares two pointers to different objects, which is undefined behavior. For such patterns, comparing only the pointer offsets would give unexpected results, since it does not order objects. Instead, we establish an ordering using the integer representations of the pointers (see Section 5.3).

5.2 Pointer-to-Pointer Casts

In general, casts between pointers are implementation-defined (C11 6.3.2.3 §7). At the platform level, they are undefined if the converted pointer is not correctly aligned for the referenced type. Safe Sulong's abstracted architecture does not require any pointer alignments, so we support casts between different pointer types, as required by Lenient C. Since in our architecture, pointer-to-pointer casts do not change the underlying object representation, we can simply achieve the desired behavior by not performing any action.

5.3 Conversions between Pointers and Integers

We found that many applications assume pointers to be regular integer types. Consequently, some programs arbitrarily convert pointers to integers, perform computations on the integers, convert them back and dereference them. Additionally, programmers sometimes craft pointers from integers that are not obviously related. For example, the Cerberus survey showed that programmers rely on being able to compute the difference between two pointers, and using the pointer difference to refer from one object to another [26]. Another example are compressed oops in the Hotspot VM, where on 64-bit architectures addresses are compacted to 32 bits [41]. Finally, some popular C applications store information in unused bits of an address [6].

Such patterns are implementation-defined and discouraged (C11 6.3.2.3 §5); for example, they often cause vulnerabilities when upgrading to a platform on which data types have a different bit width [56]. Approaches that represent C memory as an array can easily support them, but they cannot rely on the GC to reclaim dead C objects. When programmers can construct pointers arbitrarily, a GC cannot securely reclaim *any* objects. Consequently, GCs for C must compromise. For example, the *Boehm GC* assumes all values to be pointers that, if treated as pointers, refer to a valid memory region. The *Magpie* GC assumes only those values to be reachable that have a pointer type [33]. Given the tradeoffs, we present two strategies for converting integers to pointers: the first prohibits converting integers to pointers, and the second must — like the Boehm GC and Magpie — rely on heuristics for garbage collection.

The first strategy converts an address to a 64-bit integer value by concatening the 32-bit hash of the pointee with the `offset` (`(long) System.identityHashCode(pointee) << 32 | offset`). Once an address has been converted to an integer, it loses its reference to its `pointee`. When converted back to a pointer, we assign the integer value to `offset`, and `NULL` to the `pointee`. The pointer can no longer be dereferenced. This can be a problem if a pointer is copied byte-wise (e.g., in functions similar to `memmove` or `memcpy`), since only its integer representation is copied. If two pointers referring to the same object are converted to integers, the ordering is maintained if the `offset` does not exceed 32 bits. For pointers with a `NULL` `pointee` we use the 64-bit pointer offset as an integer representation to maintain the order relation between pointers that were converted back and forth to integers. Note that this approach is unsound for pointers to different `pointees`, because it can yield identical or overlapping values for different pointees. This representation allows the relation operators to be total and transitive. However, it violates antisymmetry; that is, two pointers can have the same integer representation when they refer to different objects. That is, `(long) p == (long) q` for two pointers `p` and `q` can yield `true`, even if the pointers refer to different objects. Nevertheless, we have not yet found a program that relies on the antisymmetry property; programs typically use the equality operators on pointers to determine equality.

The second strategy is to assign a unique ID to every object when it is converted to an integer. The first strategy could also use unique IDs if an application requires antisymmetry. This ID is to be incremented by the size of the object. To support dereferencable pointers that were obtained by integers, we store "escaping" objects (i.e., objects whose pointers are converted to integers) in a tree data structure that associates the range of addressable bytes with an object. When an integer is converted to a pointer, the conversion operation looks up the object in this tree. Using the integer representation of the first strategy here would be dangerous, since an application could gain access to another object if they share the same hash code. Note that escaped objects stored in the tree would never be collected by the GC. To address this, the GC is allowed to collect such pointers when the application runs low on memory (by using a `SoftReference`). An alternative strategy would involve using a least-recently-used technique [32] to keep only those mappings alive that are used by the application. The drawback is that object graphs could be collected even though the application still wants to use them, specifically when the integer value is the only reference to the object graph, and when the application runs low on memory.

5.4 Reading from Memory

Two pointers can alias, which means that they can point to the same memory location. A frequent source of errors is that compilers assume that pointers cannot alias when programmers intend them to do [9]. The best known aliasing restriction is the strict-aliasing

```
int func(int *a, long *b) {
    *a = 5;
    *b = 8;
    return *a;
}
```

Figure 7: Example demonstrating strict aliasing

rule: the C11 standard specifies that two pointers of different types (if neither is a char pointer) cannot alias (C11 6.5 §7). Figure 7 shows an example that can yield unexpected results for programmers that are not familiar with this rule. Note that, without optimization, passing two identical pointers will likely yield a value of 8, since `a` and `b` alias. However, C's type rules do not allow them to alias, and when compiler optimizations are enabled, the return value is typically optimized to always be 5. Consequently, large projects often disable strict aliasing through the `-fno-strict-aliasing` compiler flag in GCC and LLVM [9, 26]. In Lenient C, we explicitly allow two pointers of different types to alias. Moreover, we allow that an incompatible type can be used to read from — or write a value to — the pointee. In Safe Sulong, storing a value or reading a value maps to a call of the `read` or `write` operation on the pointee.

6 ARITHMETIC OPERATIONS

C programmers often do not anticipate the semantics of corner cases in arithmetic operations. Many approaches try to find program errors related to arithmetic operations, especially integer-based errors [5, 11]. Our goal is to define the semantics of integer operations as programmers would currently expect from the C standard. To this end, Lenient C largely follows the way how corner cases are handled in Java, which also corresponds closely to the AMD64 operations. Note that unsigned operations can be implemented with operations on signed types. For example, we implement unsigned division on integers using `Integer.divideUnsigned` provided by the Java Class Library. Below, we explain how we address the corner cases in the arithmetic operations.

Data Model. C does not commit to a specific data model (e.g., LP64) that assigns sizes to all data types, and neither does Lenient C. However, in contrast to the C standard, we assume that signed integers are represented in two's complement, as is the case in most programming languages and hardware architectures. Consequently, we can assign useful semantics to implementation-defined corner cases in arithmetic operations. We define that right-shifting a negative value (of a signed type), which is implementation-defined (C11 6.5.7 §5), behaves like an arithmetic shift; that is, the sign bit of the value is extended to preserve the signedness of the number.

Signed integer overflow. While integer overflow is defined for unsigned types, it is undefined for signed integers. Many signed operations can overflow (+, -, *, /, %, and $\ll$ (C11 6.5.7 §4)), specifically when the result of the operation cannot be represented in the data type of the operation. Programs commonly rely on both signed and unsigned integer overflow, for example, for hashing, overflow checking, bit manipulation, and random-number generation [11]. Since in two's complement the range of representable positive and negative numbers is asymmetric, overflows can also occur for division and modulo. On architectures that support two's complement, integer overflow typically wraps around, as most programmers would expect. GCC and Clang provide the `-fwrapv` flag, which enforces this behavior. For example, the SPEC 2000 `197.parser` benchmark requires this flag, since today's compilers would otherwise optimize the code in a way that lets the benchmark go into an infinite loop [11]. Safe Sulong provides wraparound semantics per default, which we implemented using the standard Java arithmetic operators.

Division by zero. If the second operand of a division or modulo operation is zero, the result is undefined (C11 6.5.5 §5). In most languages and on most architectures, division by zero traps.[6] Since it is unclear what value a division by zero should return, Lenient C always traps in such cases, which also corresponds to Java's default behavior.

Invalid shift amount. If the shift amount of a left- or right-shift is negative or greater or equal to the width of the shifted operand, the result is undefined. As initially demonstrated, architectures handle negative shift amounts and excessively large shift amounts differently. We decided to implement the semantics of Java, which also correspond to those of AMD64, where the shift amount is truncated to 5 bits.

Integer and floating-point conversions. Converting between floating-point numbers and integers or converting between floating-point numbers with different types can yield undefined behavior if the value is not representable by the destination type. Examples include converting `NaN` to an integer, converting a large `double` value to a `float`, and converting a large `long` value to a `float`. To implement casts efficiently across platforms, execution yields an unspecified value in such cases.

7 EVALUATION

We evaluated our Lenient C dialect by comparing it with the Friendly C standard and the *SEI CERT C Coding Standard* (see Table 1). Additionally, we implemented the dialect in Safe Sulong.

Comparison with Friendly C. Of the 14 features that the Friendly C standard proposes, Lenient C explicitly addresses 12, for which it mostly requires stricter guarantees. Friendly C aims to be implemented by a static compiler and makes tradeoffs that enable its efficient implementation across platforms (see below). Lenient C prioritizes consistent behavior and safety over speed, and allows implementers less leeway. Lenient C requires freed objects to stay alive, which meets Friendly C's requirement that a pointer's value should not change when its lifetime is exceeded (1). It requires trapping upon out-of-bounds accesses and `NULL` pointer dereferences (4), whereas Friendly C also allows an unspecified value. Friendly C demands more lenient treatment for signed-integer overflows (2), invalid shift amounts (3), division-related overflows (5), and unsigned left-shifts (7). Lenient C addresses these demands and leaves less leeway for a compatible implementation; for example, Friendly C specifies an unspecified result for shift operations with an invalid shift amount, while Lenient C requires the shift value to be masked. Like Lenient C, Friendly C requires that externally visible side effects not be reordered (6), and that a compiler should not be granted additional optimization opportunities when inferring that a pointer is invalid (13). Additionally, both Lenient C and

[6] In MySQL, however, division by zero yields a `NULL` result.

Friendly C abolish the strict aliasing rule (10). While Friendly C specifies reads from uninitialized storage to yield an unspecified value (8), Lenient C requires such reads to return 0. Both Friendly C and Lenient C allow out-of-bounds pointers and arbitrary computations on pointers (9). With respect to functions, Friendly C requires that, when control reaches the end of a non-void function, an unspecified value be returned if the return statement is missing (14); Lenient C requires 0 to be returned.

Both Friendly C and Lenient C are not comprehensive. Of the two points that Lenient C does not address, one (11) is related to data races. We will consider extending the Lenient C standard to address multithreading issues as part of future work. The other point (12) proposes `memmove` semantics for `memcpy`. Using `memcpy` with overlapping arguments is a common error, so Safe Sulong implements `memcpy` using `memmove`. However, we deferred the discussion of lenient semantics for standard library functions. Lenient C has additional guarantees compared to Friendly C. It demands additional lenience on memory management errors (M1, M2, M3) and requires struct padding to be initialized to 0 (A2). It also establishes an ordering on objects that should hold when pointers are converted to integers (P3, C2). Lenient C allows arbitrary pointer casts (C1) and pointer arithmetic on pointers to non-array objects (P4). Additionally, it specifies semantics when types or number of arguments in a function call do not match the function declaration (F2, F3). Lenient C requires signed numbers to be represented in two's complement (G3), an invalid size in variable-length arrays to trap (G4), and otherwise undefined casts to produce an unspecified value (C3).

Comparison with the *SEI CERT C Coding Standard*. The *SEI CERT C Coding Standard* is a forward-looking set of best practices for the C11 language. It comprises 14 chapters with individual rules, each of which describes a best practice along with anti-patterns. Our goal in Lenient C is not to rely on programmers following these practices; instead, we assume that they have anti-patterns in their code which they assume to work correctly. Thus, for our evaluation we examined whether Lenient C addresses such anti-patterns. The *SEI CERT C Coding Standard* recommendations are comprehensive, and we excluded a number of chapters because they do not fall into the scope of our work. Specifically, we excluded the chapters on the preprocessor (PRE), library functions (FIO, ENV, SIG, ERR), and concurrency problems (CON).

The chapter regarding declarations and initialization (DCL) contains several rules of interest to Lenient C. It requires that variables be declared with appropriate storage durations (DCL30-C); Lenient C keeps referenced objects alive and thus accepts inappropriate storage durations. The chapter requires that no incompatible declarations of the same object or function be made (DCL40-C), which Lenient C partly addresses by trapping when a function is called with a wrong number of arguments.

The chapter regarding expressions (EXP) consists of rules with different concerns: it discusses invalid read operations, non-portable pointer casts, and errors in calling functions. Lenient C allows programs to read uninitialized memory (EXP33-C) and compare padding values (EXP42-C), which it requires to be initialized with zeroes. When `NULL` pointers are dereferenced, Lenient C specifies that the implementation must trap (EXP34-C). It enables arbitrary pointer casts (EXP36-C) and reading pointers using an incompatible type (EXP39-C). Lenient C requires trapping when a function is called with a wrong number of arguments (EXP37-C). The rule also addresses wrong types in arguments, for which a callee in Safe Sulong performs automatic conversions.

The chapter on integers (INT) warns against wrong integer conversions (INT31-C), using types with an incorrect precision (i.e., bit width, INT35-C) and unsigned integer wrapping (which is defined behavior, INT30-C); these rules are of little concern to Lenient C. Lenient C specifies wrapping semantics for signed overflow (INT32-C), traps on division or remainder operations with zero as second operand (INT33-C), and requires the shift amount to be masked (INT34-C). One rule is concerned with conversions between pointers and integers (INT36-C) and details anti-patterns using crafted pointers, which are implementation-defined. Lenient C does not specify the semantics of casts between pointers and integers. Safe Sulong provides two different standard-compliant strategies, of which only the second (that stores escaped objects in a map) addresses user expectations in this context. As part of future work, we plan to investigate both strategies using a case study of user programs.

The chapter on floating-point numbers (FLP) is concerned mainly with issues that are valid for floating-point numbers in general, so they are of little interest in our evaluation; however, we defined that otherwise undefined casts between integers and floating-point numbers yield unspecified values (C3).

We addressed all anti-patterns of the array chapter (ARR), which primarily discusses pointer arithmetic. Lenient C supports pointer arithmetic on non-array types (ARR37-C, ARR39-C), creating out-of-bounds pointers (ARR30-C, ARR38-C, ARR39-C), but traps when dereferencing an out-of-bounds pointer (ARR30-C). It requires trapping for non-positive variable-length array sizes (ARR32-C). Additionally, Lenient C supports subtracting and comparing pointers to different objects (ARR39-C).

The characters and strings chapter (STR) discusses issues which are statically detectable or which concern the usage of library functions.

All rules of the memory management chapter (MEM), except the `realloc` alignment requirement, are of interest to us, and Lenient C addresses each of them. Lenient C allows dangling pointers (MEM30-C) to be accessed as if they were still alive and ignores invalid frees (MEM34-C). It assumes a GC that reclaims memory that is no longer needed (MEM31-C). Upon out-of-bounds accesses, Lenient C requires implementations to trap (MEM33-C, MEM35-C).

The miscellaneous chapter (MISC) mostly discusses library functions; however, MSC37-C states that control should never reach the end of a non-void function, in which case Lenient C specifies a zero value to be returned.

Implementation in Safe Sulong. We implemented Lenient C in Safe Sulong, a system for executing LLVM-based languages on the JVM. It does not directly execute C code, but LLVM IR, which is the RISC-like intermediate format of the LLVM framework [23]. We implemented Safe Sulong on top of the Truffle language implementation framework [53], which uses the Graal compiler [57] to compile frequently-executed functions to machine code. Graal optimizes the code based on Java semantics and thus preserves side effects such as `NULL` dereferences, out-of-bounds accesses, and arithmetic errors. Safe Sulong is based on Native Sulong [38], but it

represents C objects on the managed Java heap instead of allocating them in native memory. Its peak performance — reached after a warm-up phase where Graal compiles frequently executed functions to machine code — is currently around half that of executables compiled by Clang `-O3` on small benchmarks. As part of future work we intend to thoroughly evaluate Safe Sulong's performance and lower its overhead.

8 RELATED WORK

ManagedC. We previously worked on a Truffle implementation for C called *ManagedC* [16]. ManagedC aimed to detect out-of-bounds accesses and use-after-free errors, but otherwise assumed strictly conforming C programs. Note that the implementation of Lenient C in Safe Sulong is based on ManagedC, in particular in its representation of pointers. However, while ManagedC had a relaxed mode which allowed some illegal type casts, it left open which C dialect it supported and how other portability issues (e.g., subtracting pointers to different objects) were addressed. Further, Lenient C's main goal is not to find errors in C programs, but to tolerate them whenever possible. Unlike ManagedC, we also tolerate use-after-free errors.

Dialects of C. Several C-like languages have been proposed, for example, *Polymorphic C* [45], *Cyclone* [18], and *CCured* [30]. These dialects add type safety and/or detection of memory errors to C-like languages, but are not source-compatible with C. They do not touch on other aspects of non-portable behavior.

Pointer-to-Integer casts. Kang et. al presented an approach to using pointer-to-integer casts in formal memory models [19]. Most formalizations rely on logical memory models (e.g., CompCert [24]), in which pointers are represented as pairs of an allocation block and an offset within that block, similar to our pointer pairs. They extended this approach such that a pointer has two representations: one in the concrete and one in the logical model. Per default, all allocation blocks are allocated as logical blocks; only when a pointer is cast to an integer is the logical pointer realized as an integer. This approach is similar to ours, where we convert pointers that are cast to an integer to a concrete representation that takes into account the hash code and `offset`.

C to Java converters. Several systems exist for executing C on the JVM, by converting C programs either to Java or to Java bytecode [10, 25]. C-to-Java systems are typically used to migrate legacy code and thus focus on producing readable code at the cost of correctness (e.g., by not supporting unsigned types [25]). Most of them do not support non-portable patterns such as casting pointers to integers. Only Demaine's approach touched on lenient execution [10]; for example, he stated that pointer comparisons between different objects could be established by ordering of the heap. These approaches use an object hierarchy similar to ours, which makes them suitable for implementing Lenient C.

CHERI. CHERI [54] is a RISC-based instruction set architecture that provides hardware support for memory safety through unforgeable fat-pointers (called capabilities). As with Safe Sulong, the CHERI authors found that it was straightforward to support well-behaved C programs, but that it was difficult to compile and run those with non-portable behavior. They studied problematic patterns (portable, undefined, and implementation-defined idioms) including removing const qualifiers, pointer arithmetic idioms, storing bits in an address and storing pointers in integer variables [6]. They found many instances of these patterns, and adapted their execution model to better support such idioms.

9 CONCLUSIONS AND FUTURE WORK

We found that implementing the Lenient C dialect is helpful when executing C programs that can be found "in the wild", as it removes the need to "fix" them to use only standard-compliant C. This dialect is best suited for execution on a managed runtime. However, we hope that some of the rules will be incorporated into static compilers to alleviate the problem of compiler optimizations conflicting with user assumptions (thus breaking their code). The inspiration to create this dialect came while executing non-portable programs with Safe Sulong. However, Safe Sulong is a prototype and cannot execute large programs, mainly due to unimplemented standard library functions. Consequently, we have validated Lenient C informally on programs of up to 5000 lines of code. We are currently adding support for running a complete, well-behaved libc (such as the musl libc [28]) on top of Safe Sulong. This requires Safe Sulong to execute inline assembly and provide functionality that is typically provided by the operating system [37]. Once Safe Sulong has reached a degree of completeness that enables it to execute larger applications, we will conduct a case study to determine which features of Lenient C are most useful for large real-world programs and which features are still missing. In particular, we have yet to determine which of the two strategies of converting between pointers and integers is most suitable in practice. Lenient C still lacks stricter semantics for standard library functions, preprocessing and other issues (e.g., related to const and restrict qualifiers). Furthermore, C/C++ concurrency semantics remain (among others) unsatisfactory [1, 8], and Lenient C currently lacks stricter semantics for multithreading. We will consider these issues as part of our future work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. We also thank Ingrid Abfalter, whose proofreading and editorial assistance greatly improved the manuscript. We thank all members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at Johannes Kepler University Linz for their support and contributions. We thank Benoit Daloze and Mike Hearn for comments on an early draft. The authors from Johannes Kepler University Linz were funded in part by a research grant from Oracle.

REFERENCES

[1] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

[2] Daniel Julius Berstein. 2015. boringcc. (2015). https://groups.google.com/forum/m/#!msg/boring-crypto/48qa1kWignU/o8GGp2K1DAAJ (Accessed August 2017).

[3] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

[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. https://doi.org/10.1109/CGO.2011.5764689

[5] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. 2007. RICH: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering* (2007), 28.

[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] Pascal Cuoq, Matthew Flatt, and John Regehr. 2014. Proposal for a Friendly Dialect of C. (2014). https://blog.regehr.org/archives/1180 (Accessed August 2017).

[9] Pascal Cuoq, Loïc Runarvot, and Alexander Cherepanov. 2017. Detecting Strict Aliasing Violations in the Wild. In *Verification, Model Checking, and Abstract Interpretation: 18th International Conference, VMCAI 2017, Paris, France, January 15–17, 2017, Proceedings*, Ahmed Bouajjani and David Monniaux (Eds.). Springer International Publishing, Cham, 14–33. https://doi.org/10.1007/978-3-319-52234-0_2

[10] Erik D. Demaine. 1998. C to Java: converting pointers into references. *Concurrency: Practice and Experience* 10, 11-13 (1998), 851–861. https://doi.org/10.1002/(SICI)1096-9128(199809/11)10:11/13<851::AID-CPE385>3.0.CO;2-K

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 1, Article 2 (Dec. 2015), 29 pages. https://doi.org/10.1145/2743019

[12] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *Proceedings of the 2015 IEEE Security and Privacy Workshops (SPW '15)*. IEEE Computer Society, Washington, DC, USA, 73–87. https://doi.org/10.1109/SPW.2015.33

[13] M Anton Ertl. 2015. What every compiler writer should know about programmers or "Optimization" based on undefined behaviour hurts performance. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*.

[14] Jon Eyolfson and Patrick Lam. 2016. C++ const and Immutability: An Empirical Study of Writes-Through-const. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*. 8:1–8:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.8

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

[16] 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

[17] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 336–345. https://doi.org/10.1145/2813885.2737979

[18] 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. http://dl.acm.org/citation.cfm?id=647057.713871

[19] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A Formal C Memory Model Supporting Integer-pointer Casts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 326–335. https://doi.org/10.1145/2737924.2738005

[20] 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

[21] Robbert Krebbers. 2014. An Operational and Axiomatic Semantics for Non-determinism and Sequence Points in C. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2535838.2535878

[22] 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 (Accessed August 2017).

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[24] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[25] Johannes Martin and Hausi 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

[26] 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 '16)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/2908080.2908081

[27] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. (Feb. 2017). https://www.vusec.net/download/?t=papers/safeinit_ndss17.pdf

[28] musl libc. 2017. (2017). https://www.musl-libc.org/ (Accessed August 2017).

[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. (2010), 31–40. https://doi.org/10.1145/1806651.1806657

[30] 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 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

[31] 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

[32] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. (1993), 297–306. https://doi.org/10.1145/170035.170081

[33] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. 2009. Precise Garbage Collection for C. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM '09)*. ACM, New York, NY, USA, 39–48. https://doi.org/10.1145/1542431.1542438

[34] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. (2010). https://blog.regehr.org/archives/213 (Accessed August 2017).

[35] John Regehr. 2015. The Problem with Friendly C. (2015). https://blog.regehr.org/archives/1287 (Accessed August 2017).

[36] 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 (Accessed August 2017).

[37] Manuel Rigger. 2016. Sulong: Memory Safe and Efficient Execution of LLVM-Based Languages. In *ECOOP 2016 Doctoral Symposium*. http://ssw.jku.at/General/Staff/ManuelRigger/ECOOP16-DS.pdf

[38] 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

[39] Armin Rigo and Samuele Pedroni. 2006. PyPy's Approach to Virtual Machine Construction. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753

[40] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. 2004. Enhancing Server Availability and Security Through Failure-oblivious Computing. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251254.1251275

[41] John Rose. 2012. CompressedOops. (2012). https://wiki.openjdk.java.net/pages/diffpages.action?pageId=11829259&originalId=26312779 (Accessed August 2017).

[42] Robert C. Seacord. 2008. *The CERT C Secure Coding Standard* (1st ed.). Addison-Wesley Professional.

[43] Robert C. Seacord. 2016. Uninitialized Reads. *Queue* 14, 6, Article 50 (Dec. 2016), 17 pages. https://doi.org/10.1145/3028687.3041020

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

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

[46] E. Stepanov and K. Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 46–55. https://doi.org/10.1109/CGO.2015.7054186

[47] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and Analyzing Compiler Warning Defects. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 203–213. https://doi.org/10.1145/2884781.2884879

[48] What's the difference between memcpy and memmove? 2017. (2017). http://c-faq.com/ansi/memmove.html (Accessed August 2017).

[49] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 405–419. https://doi.org/10.1145/3064176.3064211

[50] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *NDSS*.

[51] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined Behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems (APSYS '12)*. ACM, New York, NY, USA, Article 9, 7 pages. https://doi.org/10.1145/2349896.2349905

[52] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 260–275. https://doi.org/10.1145/2517349.2522728

[53] 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)*. ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[54] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[55] 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 *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 133–144. https://doi.org/10.1145/2647508.2647517

[56] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. 2016. Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 541–552. https://doi.org/10.1145/2976749.2978403

[57] 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

[58] Yves Younan. 2015. FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*. https://doi.org/10.14722/ndss.2015.23190