Im-C – A Memory-Safe C Interpreter Providing a Better Learning, Testing, and Debugging Experience

Masaki Kunii
Kyoto University of Advanced Science, Japan

Ian Piumarta
Kyoto University of Advanced Science, Japan

Abstract

While it is a highly flexible and efficient language, C programming can be annoying for beginners who have difficultly finding, debugging, or understanding errors – especially those related to memory manipulation. This paper describes an initial phase of Im-C, a project whose goal is to develop a number of related environments that progress from easy-to-understand "C-with-objects" to a safe environment for both testing and learning to write "real" C programs, bringing more fun to the programming experience. In the current phase an experimental C interpreter has been developed to provide a foundation for real-time validation of C program behavior. The interpreter identifies errors related to memory manipulation, pointer operations and array access, by storing all values as objects containing meta-data, while preserving the semantics expected by C programmers. We validate the effectiveness of our interpreter by comparing its ability to detect a range of both common and obscure errors against the two most popular and mature tools currently available.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases C programming Language, debugging Tools, memory Safety

Digital Object Identifier 10.4230/OASIcs.Programming.2025.12

1 Introduction

The C programming language [3] is used in a wide range of fields, including missing-critical applications such as systems programs, operating systems, and embedded systems. However, its high degree of freedom and relatively low level semantics makes it prone to bugs, especially those caused by memory manipulation errors and undefined behaviors. Students who are not already familiar with machine language programming also suffer a steep learning curve when trying to understand how to safely manipulate memory and pointers. Error messages from C compilers can be difficult for beginners to decode, creating a barrier to understanding the exact causes of problems and the programming techniques that can mitigate them. In the most extreme case, a student might have nothing more informative than "bus error" or "segmentation fault" to indicate their program is not correct.

In this study we reports on an early experimental component of the Im-C system. Im-C is a project whose goal is to mitigate some of the problems that accompany C programming, especially those related to memory and pointers, improving the learning experience for novice programmers and students.

Im-C aims to provide a safe and easy-to-understand C programming environment and an (optionally) simplified C grammar. In the current phase we have developed a C interpreter with enhanced error detection and reporting.

In addition to the usual out-of-bounds, dangling pointer, and other run-time errors traditionally detected by C safety checkers, Im-C aims to warn students about undefined behaviors. These often do not produce warnings or fatal run-time behavior, but they do

Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025).
Editors: Jonathan Edwards, Roly Perera, and Tomas Petricek; Article No. 12; pp. 12:1–12:18

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

have platform-dependent results which makes the program non-portable. This will help learners quickly understand problems in their code, make effective corrections, rapidly learn the "rules" (often subtle and not widely discussed) of safe and portable C programming.

We hope that Im-C will help not only early stage programmers but also in some cases professional programmers working on production-quality code. For example, data structures are often written as small, self-contained components with their own unit tests. Running those unit tests under Im-C can potentially uncover errors that would otherwise go undetected.

The rest of this paper is organized as follows. Section 2 discusses the two popular run-time C verifiers whose goals are most similar to ours. Section 3 describes our approach to detecting run-time errors in C programs. Section 4 presents some of the design of the Im-C interpreter, particularly those aspects most closely related to detecting memory and pointer manipulation errors. Section 5 describes the test suite we use to evaluate error detection, Section 6 presents the results of running the tests with the three verifiers, and Section 7 discusses those results. Finally, Section 8 presents some concluding remarks and our intentions for future work on Im-C.

2 Related Work

Program verifiers have been around for a long time. Perhaps the first static verifier was `lint` [2], which picked fluff out of C programs by parsing and analyzing them. Dynamic behavior verification arrived later with a few commercial tools which in turn inspired open-source projects including Valgrind and then ASan. Since the latter two are now the most widely-used run-time behaviour verification tools we will concentrate on them.

The address space sanitizer ASan [7] was introduced by Google in 2012 and is now included as a compilation option in several popular C compilers such as `gcc` [8] and Clang [9]. ASan modifies a program at compile time, inserting additional code to monitor memory accesses and detect errors. Several data structures are used at runtime including red zones, which are regions of memory added around allocated blocks to catch out-of-bounds accesses, and shadow memory, which tracks the state of each byte in the application's memory to detect invalid accesses. ASan also quarantines freed memory blocks to detect use-after-free errors. These mechanisms help ASan to identify a number of different memory errors including out-of-bounds accesses, dangling pointers (use-after-free errors), and memory leaks.

Valgrind [5] began in the early 2000s as a cache profiling and memory debugging tool. It is language-independent and works directly with machine instructions, dynamically recompiling them into a program with additional instrumentation that measures or verifies the program's behavior. Various "sub tools" are available, each of which adds different instrumentation to the recompiled program. Memcheck is the default and most commonly used tool. It verifies the addressability (whether an address refers to a valid region of memory) and the validity (whether the contents of memory are initialized) of every memory access as the program runs. It also inserts red zones around allocated memory to detect off-by-one errors, where a program might inadvertently attempt to access memory just before, or just after, an allocated region of memory.

CCured [4] accepts a mostly C89-compatible language as input and performs static analyses of pointer usage. Additional code is then inserted into the program, when necessary, for bounds checking or to ensure the results of arbitrary casts are used safely.

Im-C takes a very different approach to detecting memory- and pointer-related errors. Whereas ASan modifies a compiled program at compile time and Valgrind modifies a compiled program at run time, Im-C interprets the C source program directly without any compilation.

Addressability and validity of addresses and the data they refer to are verified by tracking how each address is created, how it is subsequently modified, and how uses of it relate to the object from which it was created – in particular whether they are legal and have well-defined behavior.

Fail-Safe C [6] is a compiler for C89 that includes run-time checks of memory and pointer use. It has the most similar run-time safety strategy compared to Im-C, keeping meta data for allocated memory and pointers that allow uses of them to be checked. Like Im-C, it uses a garbage collector to reclaim memory; the `free()` function marks memory as unusable but does not actually de-allocate it until it becomes unreachable.

CINT [1] is a C interpreter that can call, or be called from, compiled code. It is used primarily for rapid prototyping (eliminating the compiler from the development cycle), documentation (running example statements stored in text files and inserting up-to-date output into the documentation), and to allow tools to interoperate with compiled code through C statements generated as plain text. It does not have any special features that enhance error detection or safety. Its relationship to Im-C is therefore limited to the execution mechanism and not to the enhancement of the end-user experience.

3 Methodology

The goal of Im-C is to provide an environment in which C programs can be run safely, with most memory manipulation errors, in particular, being detected and properly reported instead of leading to "segmentation fault" or "bus error" at run time. It is aimed at student writing terminal-based projects, or professional programmers who are writing standalone software components with unit tests.

Im-C executes C programs from source code. It parses the program into an intermediate representation which is then immediately interpreted. In the absence of programming errors, the program should behave exactly as it would if statically compiled to a binary executable by a standard compiler, and produce identical output.

Im-C does not currently aspire to be a complete implementation of the C language, including its many runtime libraries. It does aim to provide just enough of the language and runtime libraries to support writing useful programs, especially those that students might write or that implement common data structures and pointer-based algorithms with their unit tests.

To encourage good programming practices, Im-C is slightly more strict than standard C when parsing the program. At run-time, Im-C detects a wide range of programming errors that would be silently undetected, or would exhibit platform-dependent behavior, when run as a fully-compiled program.

3.1 Parse-time errors

During parsing, Im-C imposes more stringent rules on program syntax in an effort to encourage programmers to write more consistent, readable, and maintainable programs. The intention is to encourage programmers, especially students, to adopt good practices. Since these rules do not directly affect run-time error detection we will not describe all of them, but a representative few are as follows.

In standard C declarations the "**`typedef`**" keyword can appear anywhere in the list of type qualifiers, but in Im-C it must appear first.

An integer type name (e.g., "**`short`**" or "**`long`**") cannot be omitted when declaring an **`unsigned`** variable or value, whereas in standard C it defaults to "**`int`**".

During assignment, no automatic type conversions are applied and the programmer is required to cast the r-value appropriately if its type does not match that of the l-value.

3.2 Run-time errors

To validate Im-C's operation, and to compare it with other run-time safety systems, we developed a small suite of test programs that are compiled without complaint by most C compilers but, when run, exhibit incorrect behaviour or catastrophic failure. These test programs are described in Section 5 and their source code is presented in Appendix A; they should be mostly self-explanatory. The tests are designed to cause a number of problems which would, ideally, all be detected by a run-time C verification mechanism.

Uninitialised values occur when the program attempts to use a value that has not been initialised. Uninitialised global variables and arrays are implicitly filled with 0 in C, but Im-C will warn about their use to encourage all memory to be explicitly initialised. Uninitialised automatic local variables and arrays are not implicitly initialised and attempting to use a value stored in one is an error. Using an uninitialised value in memory allocated with `malloc()` is also an error.

Dangling pointers are pointers that refer to memory that is no longer valid. They can be caused by storing the address of a local variable that is subsequently destroyed when control exits from its block, or by freeing allocated memory while a pointer continues to store an address within it. Creating a dangling pointer in these ways is not an error, but attempting to dereference one is.

Multiple free errors occur when a heap-allocated block of memory is freed more than once. This is related to the dangling pointers since it occurs when a dangling pointer is passed as the argument to `free()`.

Invalid pointers refer to regions of memory that do not exist. Casting an arbitrary integer into a pointer, then attempting to dereference it, is an error. Dereferencing a NULL pointer is a kind of invalid pointer error.

Invalid free is similar and occurs when `free()` is called with a pointer that refers to something other than a heap-allocated block of memory.

Memory leaks occur when a program exits without freeing all the memory that it allocated. While not an error, a long-running program can encounter issues due to memory exhaustion. We intend eventually to detect leaked memory as it becomes unreachable, but for now Im-C issues warnings about leaked heap-allocated memory at the end of execution.

Out-of-bounds access errors occur when the program attempts to use an invalid index into an array. They also occur if the program attempts to dereference an address that lies beyond the memory that it refers to. For example, by taking the address of a variable, incrementing it, and then attempting to access memory at the resulting address.

Illegal pointer comparisons occur when two addresses are compared for order in memory (for example, using "`<`") but they refer to elements of different arrays or different heap-allocated blocks. For example, comparing the order of two addresses into the same array is not an error, but comparing two addresses in different arrays, or the addresses of two integer variables, is.

Illegal pointer increment is not an error, but should not occur in the vast majority of bug-free C programs. Illegal pointer values occur when the program increments a pointer beyond the first byte following the array or heap block that it refers to. Similarly, decrementing a pointer to an offset less than -1 relative to the object that it refers to is probably indicative of a programming error. These situations arise easily in programs containing off-by-one errors, or where the programmer uses "`<=`" as the condition in a loop that is intended to increment a pointer only up to the end of an array.

4 Design

Im-C is implemented as a typical interpreted language. Source code is parsed, converted into a tree representation, verified for type correctness, and then tree nodes are interpreted by a recursive evaluator function to execute the program.

In the following sections we describe those aspects of Im-C's design that are most closely related to detecting memory and pointer errors.

4.1 Parsing

Im-C first parses the source file into a forest of abstract syntax trees (ASTs). Each AST corresponds to one top-level declaration or definition in the program. During parsing, the stricter syntactic rules described in Section 3.1 are enforced. Some rewriting of ASTs is then performed to make them easier to interpret.[1]

Im-C maintains a stack of dictionaries representing variable scopes. Initially the stack contains one dictionary representing the global scope. ASTs representing global variables and `typedef`ed names are added to the global scope.

Each AST is then type checked, during which the stricter rules related to explicit casts during assignment are enforced.

4.2 Type representation

The fundamental types are represented by atoms such as `Tchar`, `Tfloat`, and `Tvoid`. Their sizes correspond to a machine with an LP64 architecture (`int`s are 32-bits whereas `long`s, `double`s, and pointers are 64-bits).[2]

Pointer types are represented by wrapping another type in a `Tpointer` object, for example, strings have type `Tpointer(Tchar)`.

Arrays are similar to pointers but also include a size which counts the number of elements; e.g., `Tarray(5, Tpointer(Tchar))` represents an array of five strings. The size of an array is the product of its `size` and the size of its referent type.

Structures are represented by objects containing the structure `tag`, the `size` of the structure (in bytes), and a list of `members` in which each element is a `Variable` object (see Section 4.3).

4.3 Run-time meta information

The Im-C interpreter is object-oriented and represents program elements and all run-time values as objects. For example, the value stored in a "`char *`" variable pointing to a string literal will be a `String` object which contains both the memory needed to store the characters

[1] For example, C's declarations specify a type followed by one or more declarators that are expressions involving the variable (or function) being declared, showing how it is used to yield a value of the specified type; for example, "`int *x[3];`" means "if you write '`*x[1]`' in your program, the result will have type `int`". The corresponding ASTs are difficult to use because the variable names are buried deeply inside the declarators; for example, the above declaration is parsed as "decl(`int`, array(3, pointer(`x`)))". Declarations are therefore rewritten to hoist the variable names out of the expressions and substitute the base type in their place; for example, "decl(`x`, array(3, pointer(`int`)))". Further details are beyond the scope of this paper.

[2] In future we may make these sizes configurable, if it proves useful to detecting portability problems related to data type sizes. Embedded systems, in particular, often have data types that are narrower than usual.

and the length of the string, to allow detection of out-of-bounds accesses. Run-time checks ensure that this variable can only ever refer to a `String` object (representing a string literal), or to an object representing indexable memory that stores values of the appropriate type, such as an array of `char`.

Operations on memory involve only a few program elements and run-time values. Safety is ensured by storing these elements and values as objects carrying all the semantic information available about the value, preserving details such as the types and sizes of memory blocks, the provenance of pointer addresses, and so on. Operations performed on these elements and values check for the run-time errors listed above. The following objects are relevant to pointer and memory operations.

Variables are stored as dictionary values in the scope stack. Each value is a `Variable` object that contains:

- the `name` of the variable;
- the `type` of the value stored in the variable (reading and writing the variable will access this value);
- the `value` stored in the variable (itself an object, carrying semantic meta-information).

Memory allocated dynamically in the heap by `malloc()`, or allocated statically for the contents of a `struct`, is represented by a `Memory` object that contains:

- the `base` of the allocated memory in the actual heap (this is the "backing store" for the contents of the memory);
- the `size` of the memory, measured in bytes;
- a `heap` flag that indicates the memory was obtained `malloc()` or similar;
- a `dead` flag that indicates that heap memory has been `free()`d or that automatic memory on the stack is no longer valid because the compound statement in which it was allocated has exited.

Arrays are represented by an `Array` object which contains:

- the `type` of the elements in the array;
- the `base` of the array's memory, which itself will be a `Memory` object;
- the `size` of the array, in elements (not bytes).

String literals are represented by `String` objects that contain:

- `elements` (characters) of the string; and
- the `size` of the string, in bytes.

Whether or not string literals can be modified at run-time is platform-dependent. Im-C enforces the conservative (and portable) rule that they are read-only. Mutable strings, having storage allocated explicitly by the program, will be represented as `Array` or `Memory` objects.

Functions are represented by `Function` objects that contain:

- the `name` of the function;
- the `type` of the function (which includes return type and parameter types);
- the `parameters` (a sequence of `Variable` objects);
- a `variadic` flag that is true if the parameters ended with "`...`";
- the `body` of the function (an object representing a block statement);
- the compiled `code` implementing the body of the function.

Pointer values represent the address of something and are represented by `Pointer` objects containing:

- the `type` of the pointer value;

- the `base` of the value that the pointer refers to. This might be a `Variable` (the result of taking the address of a variable), a `Memory` object (the result of `malloc()`, for example), an `Array` object (arrays in C are really constant pointers to their first element), or an `Integer` object (the result of casting `0` to a NULL pointer, for example).
- the `offset` from the beginning of the `base` object.

Pointers are therefore stored as "base+offset" which allows Im-C to calculate whether a pointer has been incremented byeond the memory that it is associated with, and also to check whether two pointers refer to the contents of the same memory (in particular, that two pointers refer to elements within the same array).

4.4 Execution and error detection

Once parsing and type-checking are complete, the object associated with "`main`" is found in the global scope and verified to be a function of the correct type. The AST associated with the body of the `main` function is then executed by the interpreter, passing values for `argc` and `argv` according to arguments given on the Im-C command line.

Type checking has already verified that the program obeys the static type rules of C, with our slightly more strict rules for casts. Detecting errors in the run-time use of pointers and memory as outlined in Section 3.2 is done by the interpreter. In the rest of this section we describe only those interpreter mechanisms related to this error detection.

4.4.1 Increment and decrement operators

These apply to `Integer`, `Float`, `Pointer`, and `Array` values. In the first two cases, the amount of increment indicated by the operand is added to the value and a new object representing the incremented result created.

In the case of pointers, a new `Pointer` object is created by coping the pointer being incremented and then increment the `offset` field of the copy by the amount of the operand. If the resulting pointer is before its associated base memory, or beyond the byte immediately following it, then an *invalid pointer* error is reported.

For `Array`s, a new `Pointer` is created whose `type` is the same as the array, `base` is the array itself, and the offset is the operand specifying how much has been added to the array.

4.4.2 Pointer dereference

The pointer's `base` is inspected. If it is an `Integer` then the pointer is either NULL or was cast from an arbitrary integer constant. In either case an *invalid pointer* error is reported.

In the case of a `Variable`, the offset of the pointer is checked. If it is non-zero then the pointer has been incremented or decremented away from the original variable and now points to invalid memory. An *invalid pointer* error is reported.

In the case of `Memory`, the offset of the pointer is scaled by the pointer's referent type size and the result compared to the memory's `size`. If the pointer falls outside it then an *invalid pointer* error is reported. Otherwise the `type` of the pointer is used to read a value from memory and encapsulate it in an object of the appropriate type (`Integer`, `Float`, `Pointer`, etc.).

4.4.3 Array indexing

The checks are a combination of those described for pointer increment (the effective address behaves like a pointer obtained by adding the array value to the index) and pointer deference.

4.4.4 Assignment

The l-value can be a `Variable`, pointer dereference operator, array index operator, or a structure member operator.

For non-pointer `Variable`s, type checking has already verified the assignment. The r-value is stored into the variable's `value` and the assignment is complete.

For pointer `Variable`s, the r-value might be a different but compatible type. An `Integer` r-value is converted into the `base` of a new `Pointer` object with the same `type` as the variable and stored into its `value`. A `Pointer` r-value is assigned unmodified into the variable's `value`, unless its type differs from that of the `Variable` in which case a new pointer is allocated and stored to make the types match.

Array r-values are converted into a new `Pointer` of the appropriate type, sharing the same `base` as the `Array`.

Similarly, a `String` literal assigned to a character pointer variable is converted into a corresponding `Pointer` to complete the assignment.

4.4.5 Pointer comparisons

These are often overlooked by runtime safety mechanisms for C. For example, magnitude comparisons between pointers into different memory blocks or arrays are undefined.

`Pointer`s are stored as `base`+`offset`. Equality comparisons compare both the `base` and the `offset` for equality. During magnitude comparisons, Im-C checks that both `base`s are the same. If they are the same then the magnitudes of the `offset`s are compared; if they are not the same then an *illegal comparison* error occurs.

4.4.6 Variable access

When a `Variable` is created and added to the current scope, its `value` field is set to `nil` (the undefined object). Whenever the `value` of a variable is read, Im-C checks whether it is still `nil` and, if so, reports an `uninitialised variable` error.

5 Evaluation

We evaluated ASan, Valgrind, and Im-C using a test suite written specifically for the purpose of exposing many common memory errors as well as some typically found in student programs. The source code of the test suite is presented in Appendix A. The programs and the errors they contain are described briefly in Table 1.

To pass a test the verifier must detect the error, classify it correctly, and provide relevant information to let the programmer locate and address the problem. Detecting an error but mis-classifying it or failing to provide useful contextual information is counted as a partial pass. A test fails if the verifier does not detect the error.

The four "`pointer-`" tests perform operations that have undefined behaviour. The C standard only guarantees correct, reproducible results for comparisons and arithmetic involving pointers that both refer to the same block of allocated memory or to the same array object (or to the byte following the end of that array). Other pointer arithmetic and comparisons are undefined behavior and the result is implementation dependent, and therefore non-portable. We consider these rules to apply equally to a pointer to a variable, which we treat essentially as a pointer to an array of that variable's type containing a single element.

Table 1 Programs in the test suite.

program	*run-time error*
`dangling-pointer`	Accessing a local variable whose scope has ended.
`dangling-pointer-2`	Accessing heap memory after it has been freed.
`invalid-free`	Freeing an address that was not obtained through `malloc()`.
`multiple-free`	Freeing the same pointer twice.
`segmentation-fault`	Storing a value into address zero.
`invalid-pointer`	Storing a value into a pointer obtained from an arbitrary integer.
`memory-leak`	Exiting without freeing all allocated memory.
`null-pointer`	Printing a string stored at address zero.
`uninitialised`	Using the value of an uninitialized variable to initialze another variable.
`uninitialised-2`	Using an uninitialized value obtained by initializing a variable from another uninitialized variable.
`out-of-bounds-access`	Indexing an array one past its last valid inde.x
`out-of-bounds-access-2`	Accessing a variable at an offset from the address of an adjacent variable.
`pointer-compare`	Comparing pointers that refer to elements of different arrays.
`pointer-increment`	Incrementing a pointer past the byte following its object in memory.
`pointer-out-of-bounds`	Using the address of a non-existent element of an array.
`pointer-out-of-bounds-2`	Invalid access of valid memory adjacent to an array.
`use-after-free`	Storing into heap memory after it has been freed.
`use-after-free-2`	A realistic program that traverses a linked list while making a subtle but commonly seen use-after-free error when de-allocating the list.

6 Results

Table 2 shows the results of running the test suite (see Appendix A) with the three error detectors on GNU/Linux using `gcc` (ASan) version 13.3.0, `libc` version 6.8.0, and Valgrind version 3.22.0.

ASan passed 14 of the tests and failed 4. Valgrind passed 11 of the tests, partially passed 1, and failed 6. Im-C passed 17 of the tests and partially passed 1.

ASan mis-diagnosed `dangling-pointer-2` as a heap use-after-free even though the pointer involved was to a dead local variable in the stack (note 1). The output was as follows.

```
==1514299==ERROR: AddressSanitizer: heap-use-after-free on
address 0x502000000010
READ of size 4 at 0x502000000010
    #0 0x571a51afb302 in main dangling-pointer-2.c:12
```

Im-C correctly identified leaked heap memory blocks but currently does not track or report the location in the source program where the memory was allocated (note 2).

```
heap memory not freed at end of program: <0x101142d50+4>
heap memory not freed at end of program: <0x101142d20+4>
 ... (18 more similar messages)
```

Table 2 Results of running the three verifiers against the test suite. "◯" = detection, "×" = no detection, "△" = partial or mis-classified detection. Notes: [1] No information about the source statement that allocated the object is provided. [2] Initializing a variable from an uninitialized variable propagates the uninitialized value but does not report it as an error unless the variable's value is actually used.

	ASan	**Valgrind**	**Im-C**
dangling-pointer	◯	◯	◯
dangling-pointer-2	◯	◯	◯
invalid-free	◯	◯	◯
invalid-pointer	◯	◯	◯
memory-leak	◯	◯	△[1]
multiple-free	◯	◯	◯
null-pointer	◯	◯	◯
out-of-bounds-access	◯	◯	◯
out-of-bounds-access-2	◯	×	◯
pointer-compare	×	×	◯
pointer-increment	×	×	◯
pointer-out-of-bounds	◯	×	◯
pointer-out-of-bounds-2	◯	×	◯
segmentation-fault	◯	◯	◯
uninitialized	×	×	◯
uninitialized-2	×	△[2]	◯
use-after-free	◯	◯	◯
use-after-free-2	◯	◯	◯
pass / partial / fail	14 / 0 / 4	11 / 1 / 6	17 / 1 / 0

Table 3 Performance penalty of verification for call intensive (`nfib`) and memory intensive (`sieve`) programs.

`nfib` benchmark

verifier	*n*	*calls*	*time*	*cps*	*% C*
none	45	3,672,623,805	1.03	3,565,654,180	100
ASan	45	3,672,623,805	1.24	2,961,793,391	83.1
Valgrind	38	126,491,971	1.12	112,939,260	3.17
Im-C	32	7,049,155	1.50	4,699,437	0.13
Python	36	29,860,703	1.33	22,434,788	0.63

`sieve` benchmark

verifier	*time*	*% C*
none	0.00346	100
ASan	0.00675	51.3
Valgrind	3.594	0.1
Im-C	0.70	0.49
Python	0.287	1.21

Valgrind detects the use of uninitialized values but does not detect, for example, the initialization of a variable from another uninitialized variable (note 3). Instead Valgrind propagates the uninitialized value through copies and will report it only when eventually used in an expression.

Table 3 shows the performance of two benchmarks running normally ("*none*") and under each of the three verifiers. In all cases (except for Im-C) the program was compiled with "`-O2`" optimization enabled. For perspective, the performance of Python (a highly optimised interpreted language) for the same benchmark is also shown.

The first benchmark is a doubly recursive function `nfib(n)` which calculates the n^{th} member of a sequence closely related to the Fibonacci sequence except that the result of `nfib(n)` is the number of function calls needed to calculate that result:

```
unsigned int nfib(unsigned int n) {
  if (n < 2) return 1;
  return 1 + nfib(n-1) + nfib(n-2);
}
```

The value of `n` was modified for each scenario to keep the running time just over one second. For each scenario the table shows the value of `n`, the result (number of function calls made by the program), the time taken in seconds, the number of calls per second, and the percentage of the speed (measured in calls per second) compared to the non-verified case.

The second benchmark is a prime number generator using the sieve of Eratosthenes which counts the number of primes between 2 and 2048000:

```
int sieve(void) {
  int prime[N], count = 0; // N == 2048000
  for (int i = 0;  i < N;  ++i) prime[i] = 1;
  for (int i = 2;  i < N;  ++i) {
    if (prime[i]) {
      ++count;
      for (int j = i + i;  j < N;  j += i) prime[j] = 0;
    }
  }
  return count;
}
```

It is memory intensive, making heavy use of a large array but no function calls. For the faster running times an outer loop was added to the benchmark to keep the running time over one second and a corresponding scaling applied to the recorded time to yield the values shown in the table.

7 Discussion

Im-C performed well in all but one test. It failed to report the location where leaked memory blocks were allocated. The parser will soon add information indicating where in the source each AST node was generated which will allow us to report this information.

Valgrind failed to detect the `out-of-bounds-access-2` error, which accesses a local variable in the stack as a pointer offset from an adjacent variable in the stack. This is because Valgrind works with the compiled executable only, and does not consider the semantics of the source program. In particular, it has no information about the sizes of arrays or other program elements that contain a given memory location. It therefore cannot distinguish adjacent variables in a stack from an array, and will not report any misbehavior when a pointer to one variable is de-referenced with an offset that leads to an adjacent variable. ASan catches this error because of the red zones it places around variables. However, if the offset happens to exceed the size of the red zone then ASan would also fail to catch this error. Im-C can always catch this error because part of the semantic information it preserves during execution is the identity of the variable whose address was taken to create the pointer. When using that address to access the stack, if any offset applied does not precisely cancel any increment applied to the address then an error can be reported.

The `pointer-out-of-bounds` and `pointer-out-of-bounds-2` errors were correctly detected by both ASan and Im-C, because of ASan's use of red zones around all objects and Im-C's verification that an index lies within the bounds of the array or memory object that is being accessed.

Both Valgrind and ASan failed to detect or report an error with the `pointer-compare` and `pointer-increment` tests. For Valgrind, this is understandable since it has no semantic information from the original source code to determine when a pointer is being moved outside the immediate vicinity of the object to which it refers. ASan, however, could include a check on this kind of pointer manipulation, but it chooses not to. Im-C detects these issues because every points knows the kind and size of the object to which it refers.

Valgrind failed to detect errors in the `pointer-out-of-bounds` test. These errors are illegal references to memory just beyond an array, or relative to a pointer beyond the array with a negative index to the element just before the array. Again this is because Valgrind has no information about the boundaries of variables and arrays allocated on the stack. ASan caught these errors because of the red zones it places around such objects. Im-C caught these errors because pointers know which object they refer to and so all accesses to them through pointers, with or without an offset, can be verified to lie within the memory allocated for that object.

For `uninitialized` and `uninitialized-2`, both ASan and Valgrind failed to detect a variable being initialized from the value of an uninitialized variable. In the case of ASan, this is because it does not track uninitialized memory. Valgrind differentiates between initialized and uninitialized memory but allows uninitialized values to be copied to other locations, propagating the uninitialized value without reporting an error. Valgrind will eventually report an error when an uninitialized value is used in an expression. Im-C detects the initialisation of a variable from an uninitialized variable immediately which we consider to be more useful, and safer, behavior.

While ASan comes close to the detection rate of Im-C, and is certainly a lot faster, we think Im-C has a valuable role to play informing students, in particular, about errors in pointer manipulation and comparisons. We do not expect students to have read the C standard, and so highlighting issues with undefined behavior, such as comparing pointers based on different objects in memory, is valuable. The diagnostics currently produced indicate incompatible pointer comparisons but in future could be improved to explain exactly why the incompatibility exists, for example, highlighting that not only are they different but one object resides in the stack and the other in the heap.

Many more pointer manipulation errors and questionable practices could be detected by Im-C. An example is verification of pointer casts such as

```
struct foo { int   i; };
struct bar { int *pi; };
...
  struct foo *pf = (struct foo *)malloc(sizeof(*pf));
  void *pv = (void *)pf;
  struct bar *pb = (struct bar *)pv;
  printf("%d\n", *pb->pi);
...
```

in which Im-C detects the out-of-bounds access to an 8-byte value in a 4-byte block of memory while fetching the second argument of `printf`, but misses the opportunity to report (at least) a warning for the line before that, when a pointer to memory containing a `struct foo` is cast to a pointer to memory containing a `struct bar`.

ASan exhibited only a slight performance penalty for verification, running the benchmark at more than 83% normal speed. ASan benefits from being part of the normal compiler, with optimizations applied to the additional instrumentation it adds to the program.

Valgrind imposed significant performance penalty, running the benchmark at just over 3% normal speed or 26 times slower than ASan. Valgrind is essentially a virtual machine that dynamically re-compiles the program while it is running to add the instrumentation, although

such a high penalty was surprising. Investigating the causes are outside the scope of this paper but we speculate that some of the instrumentation such as propagating uninitialised values might be costlier than ASan's simpler handling of them, and that the dynamic code generator in Valgrind might perform fewer, or less aggressive, optimizations than the compiler hosting ASan.

As expected, Im-C was by far the worst performer achieving only 0.13% the speed of normal C – 24 times slower than Valgrind, 640 times slower than ASan, and 770 times slower than the program running unverified. In spite of this, it still achieved close to 5 million interpreted function calls per second. We believe this is sufficient for student programs and unit tests that exercise self-contained data structures and other non-compute intensive algorithms.[3]

8 Conclusion

We presented an experimental component of Im-C, which aims to detect errors related to memory and pointer manipulation that novices tend to make, or which are more subtle and even professional programmers might overlook. It provides a runtime C verification environment using a C interpreter. We validated its effectiveness compared to the two most popular and mature, modern tools, ASan and Valgrind.

Using the test suite we created, we were able to detect all the errors that ASan and Valgrind could detect. In addition, we were able to detect pointer comparisons with undefined behavior and pointer manipulations that resulted in pointers that would, if used, result in undefined behavior, that the other tools did not detect.

Our method of representing values with rich semantic meta-information was helpful in informing the user of why an error occurred, although at present the location of the error is not always reported. In future, by recording source location in each AST node, we will be able to pinpoint the exact location of all errors. By including this information in the object that are created during execution we will be able to report the statement responsible for allocating memory that is involved in a run-time error.

It is not yet possible to describe the cause of every error. Future work will consider how to enhance the traceability of values and program code to try to better explain the cause of errors.

Being an AST interpreter, Im-C is much slower than ASan and Valgrind which both execute an instrumented version of the program as native machine instructions. While Im-C can never close the gap completely, we intend to narrow it by converting the AST to a flat intermediate representation and executing it using a virtual machine.

References

1 I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, Ph. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Gonzalez Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Marcos Segura, R. Maunder, L. Moneta, A. Naumann, E. Offermann, V. Onuchin, S. Panacek, F. Rademakers, P. Russo, and M. Tadel. Root – A C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12):2499–2512, 2009. 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. doi:10.1016/j.cpc.2009.08.005.

[3] To put this in perspective, Python 3.10 (which is heavily optimized for speed) runs the same benchmark 4.5 times faster than Im-C. Changing Im-C'e execution mechanism from an AST interpreter to a VM would easily close that gap.

2 S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, 1977.

3 Brian Kernighan and Dennis Ritchie. *The C programming language*. Prentice Hall, 1988.

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

5 Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. doi:10.1145/1250734.1250746.

6 Yutaka Oiwa. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1542476.1542505.

7 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

8 Richard M Stallman et al. Using the gnu compiler collection. *Free Software Foundation*, 4(02), 2003.

9 University of Applied Sciences Karlsruhe. *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* http://www.iwi.hs-karlsruhe.de.

A Test suite

This appendix presents the source code of the test suite that we used to evaluate Im-C and the other run-time safety systems. The inclusion of necessary header files has been shown only once for brevity.

The final program, `use-after-free-2`, realistically reproduces an error that is subtle, potentially undetected during testing (failure of the program is platform-dependent), and often seen in student projects that use linked lists.

```
// common header files

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
```

```
// dangling—pointer

int *alloc() {
  int i, *pi = &i;
  printf("%p\n", pi);
  return pi;
}

int main() {
  int *ptr = alloc();
  assert(ptr != 0);
  printf("%d\n", *ptr); // invalid memory
  *ptr = 42;
  printf("%d\n", *ptr);
  return 0;
}
```

```
// dangling−pointer−2

int main() {
  int *ptr = (int *)malloc(sizeof(*ptr));
  assert(ptr != 0);
  *ptr = 42;
  printf("%d\n", *ptr);
  free(ptr);
  printf("%d\n", *ptr);
  return 0;
}
```

```
// invalid−free

int *alloc() {
  int i, *pi = &i;
  printf("%p\n", pi);
  return pi;
}

int main() {
  int *ptr = alloc();
  free(ptr);
  return 0;
}
```

```
// multiple−free

int main() {
  int *ptr = (int *)malloc(sizeof(*ptr));
  assert(ptr != 0);
  free(ptr);  printf("%p\n", ptr);
  free(ptr);  printf("%p\n", ptr);
  return 0;
}
```

```
// segmentation−fault

int main() {
  int *ptr = (void *)0; // NULL
  *ptr = 42;
  return 0;
}
```

```
// invalid−pointer

int main() {
  int i, *ptr;

  ptr = &i;
  printf("%p\n",ptr);
  *ptr = 42;                    // legal access
  printf("%d\n", *ptr);

  ptr = (int *)(intptr_t)0xDeadD0d0;
  printf("%p\n",ptr);
  *ptr = 42;                    // illegal access
  printf("%d\n", *ptr);

  return 0;
}
```

```c
// memory−leak

int main() {
  for (int i = 0;  i < 10;  ++i) {
    int *ptr = (int *)malloc(sizeof(*ptr));
    assert(ptr != 0);
    printf("%p\n", ptr);
    *ptr = i;
  }
  return 0;
}
```

```c
// null−pointer

int main() {
  char *ptr = (void *)0; // NULL
  printf("%s\n", ptr);
  return 0;
}
```

```c
// uninitialised

int main()
{
  int a, b = a;
  return 0;
}
```

```c
// uninitialised−2

int main()
{
  int a, b = a;
  printf("%d\n", b);
  return 0;
}
```

```c
// out−of−bounds−access

int main() {
  int array[5] = { 0, 1, 2, 3, 4 }, i = 4;
  printf("%d\n", array[i++]);
  printf("%d\n", array[i++]); // out of bounds
  return 0;
}
```

```c
// out−of−bounds−access−2

int main() {
  int i = 0, j = 42, k = 0, *pj = &j;
  printf("%d\n", *pj);
  printf("%d\n", pj[0]);
  printf("%d\n", *(pj+1)); // out of bounds
  printf("%d\n", pj[1]);   // out of bounds
  return 0;
}
```

```
// pointer−compare

int main() {
  int array[5] = { 0, 1, 2, 3, 4 };
  int brray[5] = { 0, 1, 2, 3, 4 };
  int *p = array + 2;
  int *q = array + 4;
  int *r = brray + 4;
  if (p > q) printf("compare KO\n");
  else       printf("compare OK\n");
  // illegal comparison
  if (p < r) printf("compare KO\n");
  else       printf("compare KO\n");
  assert(!"this should not be reached");
  return 0;
}
```

```
// pointer−increment

int main() {
  int array[5] = {0, 1, 2, 3, 4};
  int *ptr = array + 4;
  ++ptr;  printf("%p\n", ptr);
  ++ptr;  printf("%p\n", ptr); // out of bounds
  return 0;
}
```

```
// pointer−out−of−bounds

int main() {
  int array[5] = {0, 1, 2, 3, 4};
  int *ptr = &array[5];
  *ptr = 42;
  return 0;
}
```

```
// pointer−out−of−bounds−2

int main() {
  int padl, array[5] = { 0, 1, 2, 3, 4 }, padr;
  int *ptr = &array[5]; // OK
  ptr[-2] = 42;         // OK
  ptr[-6] = 666;        // illegal
  return 0;
}
```

```
// use−after−free

int main() {
  int *ptr = (int *)malloc(sizeof(*ptr));
  assert(ptr != 0);
  *ptr = 42;
  printf("%d\n", *ptr);
  free(ptr);
  printf("%d\n", *ptr); // use after free
  *ptr = 43;            // use after free
  return 0;
}
```

```
// use−after−free−2

struct Link {
  int          data;
  struct Link *next;
};

struct Link *newLink(int data, struct Link *next) {
  struct Link *link =
     (struct Link *)malloc(sizeof(*link));
  link->data = data;
  link->next = next;
  return link;
}

int main()
{
  struct Link *list = 0;

  // create linked list of 10 elements
  for (int i = 0;  i < 10;  ++i)
    list = newLink(i*i, list);

  // print contents of list
  for (struct Link *ptr = list;
       ptr;
       ptr = ptr->next) {
    printf("%d\n", ptr->data);
  }

  // deallocate list
  for (struct Link *ptr = list;
       ptr;
       ptr = ptr->next) { // ptr is dangling here
    free(ptr);            // after this free
  }

  return 0;
}
```