Memory-safe Execution of C on a Java VM

Matthias Grimmer
Johannes Kepler University Linz, Austria
matthias.grimmer@jku.at

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

Chris Seaton
Oracle Labs, United Kingdom
chris.seaton@oracle.com

Thomas Würthinger
Oracle Labs, Switzerland
thomas.wuerthinger@oracle.com

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

Abstract

In low-level languages such as C, spatial and temporal safety errors (e.g. buffer overflows or dangling pointer dereferences) are hard to find and can cause security vulnerabilities. Modern high-level languages such as Java avoid these problems by running programs on a virtual machine that provides automated memory management.

In this paper we show how we can safely execute C code on top of a modern runtime (e.g., a Java Virtual Machine) by allocating all data on the managed heap. We reuse the memory management of the runtime, hence, we can ensure *spatial* and *temporal* safety with little effort. Nevertheless, we retain all characteristics that are typical for unsafe languages (such as pointer arithmetic, pointers into objects, or arbitrary type casts). We discuss how our approach complies with the C99 standard.

Compared to an optimized unsafe execution of a C program (compiled with the GNU C compiler and all optimizations enabled) our approach has overhead of 15% on average (peak-performance).

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments, Code generation, Interpreters, Compilers, Optimization

Keywords C, Memory Safety, ManagedC, Truffle, Graal, Virtual Machine, Optimization, Dynamic Compilation

PLAS'15, July 06 2015, Prague, Czech Republic.

http://dx.doi.org/10.1145/2786558.2786565

1. Introduction

Buffer overflows, dangling pointers, invalid deallocations, or null pointer accesses are common programming errors when using an unsafe low-level language such as C. These pitfalls represent software bugs that are hard to find and can cause security vulnerabilities of software projects [7, 8, 42].

Modern high-level programing languages (such as Java) avoid these issues by running the program on a virtual machine (VM) that provides an automated memory management (i.e. a garbage collector). Memory allocations and deallocations are exclusively done by the underlying VM and untyped memory accesses, including pointer arithmetic, are strictly forbidden. Instead, programmers use references to access data on the managed heap and the VM checks each access to ensure memory safety.

Such modern languages are widely used. However, programmers are still writing code in unsafe languages like C because they want to reuse modules already written in this language, achieve higher performance than what is normally possible in a memory-safe high-level language, or generally want to use the most appropriate language for a given task.

To ensure memory safety of C applications there exist various approaches, e.g. [5, 6, 9–11, 14, 20, 23–26, 26–30, 32, 33, 35, 36, 41].

All these approaches require considerable engineering effort, but none of them reuses advanced components of modern high-level language environments such as automatic memory management to ensure *spatial* and *temporal* safety. In this paper we present a novel approach for safely executing C on top of a modern runtime (e.g. the Java Virtual Machine (JVM)). Our approach reuses the automatic memory management and the memory-safe access mechanisms of the JVM for implementing the unsafe language C and hence ensures *spatial* and *temporal* safety with minimal effort.

In this paper, we show how a C language implementation on top of the JVM can safely execute C code by allocating

all data on the managed Java heap, i.e., memory safety is achieved by allocating C objects as Java objects and by replacing unsafe C accesses with safe Java accesses. For example, our C language implementation allocates an unsafe C `int` array as a safe Java `int` array and checks whether all accesses to the array elements fall within the valid array bounds even if they are done using pointer arithmetic.

We retain all characteristics that are typical for unsafe languages (such as pointer arithmetic, pointers that point into objects, or arbitrary casts) by mapping C pointer values to members of a Java object.

For implementing our approach, we extend TruffleC [16], an environment that executes C code on top of the JVM but does *not* guarantee memory safety. In this paper we introduce *ManagedC*, a modified version of TruffleC that guarantees memory safety.

In summary this paper contributes the following:

- We show how we use managed objects to represent C allocations and still retain the characteristics of unsafe languages. We discuss how our approach complies with the C99 standard [2].
- We describe how our approach allows reusing the automatic memory management of the JVM for a memory-safe execution of C.
- We compare the peak-performance of (*unsafe*) TruffleC with the performance of (*safe*) ManagedC after dynamic compilation. ManagedC is 7% faster than TruffleC and has an overhead of 15% compared to the best GCC performance (peak-performance after an initial warm-up).

2. System Overview

In this section we introduce TruffleC, a dynamically compiling C interpreter on top of the JVM, which is based on the Truffle framework and the dynamic Graal compiler. Also, we define the properties of a *memory-safe* program execution.

2.1 Truffle and Graal

Truffle [45, 46] is a platform for building high-performance language implementations. A Truffle language implementation is an AST interpreter, where the source code of the guest language is compiled to an AST that can then be executed. Truffle ASTs consist of nodes that define `execute` methods, which are recursively evaluated.

A Truffle AST can speculatively rewrite itself with a *specialized* variant at run time [45], based on profile information, which is collected at run time. If assumptions about a running program turn out to be wrong, specialized Truffle ASTs revert to a more generic unspecialized form.

Truffle guest language implementations are dynamically executed by the Graal VM [31], a modified version of the Java HotSpot™ VM. The GraalVM reuses all components of the HotSpot™ VM, including the garbage collector and the interpreter. However, it adds a new dynamic compiler

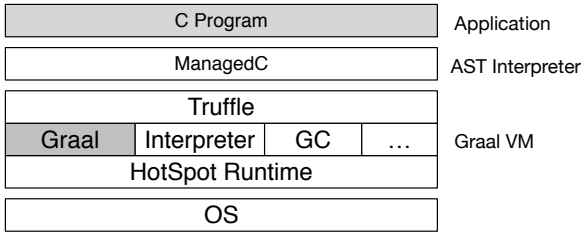


Figure 1: Layered approach of hosting the C language on top of Truffle.

(the Graal compiler), which is entirely written in Java. This allows the compiler to be used as a library by the Truffle framework: When Truffle discovers an AST that exceeds a predefined execution threshold, it uses the Graal compiler to transform the tree to highly efficient machine code. The compilation process first inlines all `execute` methods into a single method and then performs aggressive intra-method optimizations. This process is called *partial evaluation*. The Graal compiler inserts *deoptimization* points [21] at every point in the code where assumptions might become invalid, i.e., where specialized trees have to be reverted to a more generic version. Deoptimization transfers the control from compiled machine code back to the AST interpreter where the affected AST nodes eventually rewrite themselves. Figure 1 shows how TruffleC uses the Truffle framework, which itself is hosted by the Graal VM.

2.2 TruffleC

TruffleC [16] is a Truffle-based C language implementation. It dynamically executes C code and heavily uses the *self-optimization* capability of Truffle: TruffleC uses polymorphic inline caches [22] to efficiently handle function pointer calls, profiles branch probabilities to optimistically remove dead code, profiles run-time values, and replaces them with constants if they do not change over time. TruffleC performs well compared to standard C compilers such as GCC [4] or LLVM/Clang [3] in terms of peak performance [16].

TruffleC allocates C data on the native heap rather than on the garbage-collected Java heap and uses the same alignment as conventional C compilers do. This allows TruffleC to support any kind of pointer arithmetic and to replicate the same behavior as conventionally compiled C code.

TruffleC represents all pointers to the native heap as `CAddress` Java objects that wrap a 64-bit value [17] (a raw memory address). In this paper we call data on the native heap *native data*. TruffleC uses *unsafe* memory operations to access native data, hence the original version of TruffleC does not improve on memory safety.

2.3 Memory Safety

A program execution is considered *memory safe* [24, 25, 40] if it ensures *spatial* and *temporal* safety:

Spatial safety ensures that pointers are only dereferenced if they point to a valid memory region [37], i.e., if the access is within the bounds of the accessed object. This prevents errors such as *null pointer dereferences* or *buffer overflows*.

Temporal safety ensures that pointers are not used after the referenced object has been deallocated [37]. This prevents errors such as *dangling pointers* or *illegal deallocations* (e.g. calling `free` on a pointer twice).

3. Managed Addresses

In the following section we describe how ManagedC represents memory addresses and allocations of a C program.

We use `MAddress` objects to represent pointer values: `MAddress` objects are *fat pointers*, i.e., they store a reference to a *data* object and an *offset* (Figure 2). The *data* field references an object on the Java heap that we use as a replacement for a native allocation. In this paper we call a Java object that replaces a native allocation a *managed allocation*.

The *offset* is a byte offset relative to the *data* object. It is updated by address computations and pointer arithmetic and allows us to represent pointers that point into an object (e.g. to a field).

When dereferencing an `MAddress` object, the *offset* is passed to the *data* object (the managed allocation), which has a *layout* that maps the *offset* to a member. Thus we can check whether a pointer references a valid member.

Section 6.2.6.1. §4 of the C99 standard [2] defines the *object representation* of C objects: The object representation is the set of $n \times$ `CHAR_BIT` bits, where n is the size of an object of that type in bytes. An `MAddress` object uses a unique ID of the *data* object plus the *offset* as its object representation. Even though a pointer is a `MAddress` object, the system provides the illusion that it can be read as a word-sized sequence of bytes. Based on this object representation, `MAddress` objects can be compared to each other as well as converted to integers.

3.1 Address Computation and Pointer Arithmetic

Figure 3 shows the declaration of a `struct S`. We assume the object representation of the `struct` on a x86-64 platform: The `int` member `a` (4 bytes) is stored at offset 0 and the `double` member `b` (8 bytes) at offset 8. The 4 bytes from offset 4 to 8 are not in use and their content is therefore *undefined*.

Accessing array elements or `struct` members in C involves an address computation. The same is true for pointer arithmetic. Such computations can be conveniently done with `MAddress` objects.

To compute the address of an array element, a C compiler would multiply the array index with the element size and add this as an offest to the address of the array. ManagedC uses `MAddress` objects instead of raw addresses; it also multiplies

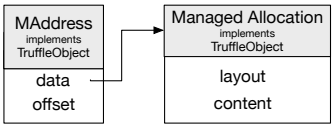


Figure 2: `MAddress` objects are pointers to managed allocations.

```
struct S {
   int a;
   double b;
};
```

```
void foo() {
   struct S *s =
        malloc(sizeof(struct S));
   s->a = 12;
   s->b = 4.2;
}
```

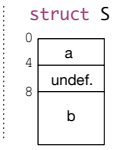


Figure 3: Writing a `struct` member.

the index with the element size and stores this value in the *offset* field of an `MAddress` object.

For processing the assignment `s->b = 4.2;` a C compiler would add the offset of b (8) to the address of the object referenced by `s` and assign the value `4.2` to this address. ManagedC represents the pointer `s` using an `MAddress` object that references a managed allocation of type `S`; its offset is 0. When accessing `s->b`, we copy the `MAddress` object of `s` and add the value 8 to its *offset* field thus referencing the field `b`. ManagedC uses this new `MAddress` object to access the member `b`.

For pointer arithmetic, Section 6.5.6 of the C99 standard [2] defines the following semantics: For an addition/-subtraction, one operand shall be an integer type. When two pointers are subtracted, both shall point to elements of the same array object and the result is the difference of the subscripts of the two array elements.

We implement these semantics as follows: When an integer value is added/subtracted to/from a pointer, we add/-subtract this integer value times the size of the pointed-to type to/from the *offset* value of an `MAddress` object. If two address values are subtracted, we calculate the difference of the subscripts using the *offset* values of the `MAddress` objects. If two addresses do not reference the same array object, the result is undefined, which conforms to the standard (see Section 6.5.6. §9 of the C99 standard).

3.2 Compliance With the C99 Standard

Section 6.2.6.1 §4 and §5 of the C99 standard define requirements on the representation of pointer types and Section 6.3.2.3 defines the conversions that are valid on pointer values. In the following list we explain how ManagedC complies with these requirements:

- 6.2.6.1 §4: An `MAddress` object has an *object representation*, which is a unique ID of the *data* plus the *offset*. Two pointers that reference the same *data* object thus have the same object representation.

- 6.2.6.1 §5: It is not possible for programmers to handcraft a valid object representation of a pointer. If the programmer constructs an object representation that is equal to the object representation of a valid pointer and if she converts that object representation back to a pointer, this pointer can still not be dereferenced.

  The C standard explicitly states that modifying an object representation by an lvalue expression that does not have character type is not allowed, but it does not say anything about modification with an lvalue expression that has character type. We decided that directly modifying an object representation always creates an invalid pointer. Otherwise it would be possible to access objects that should be inaccessible.
- 6.3.2.3 §1, 2, 7, and 8: A pointer of type A can be converted to a pointer of type B without modifying the corresponding `MAddress` objects (i.e., the *data* and *offset* fields remain unchanged).
- 6.3.2.3 §3 and 4: The integer constant `0` can be assigned to an `MAddress` object. The *data* of such an `MAddress` object is then `null` and the *offset* is `0`, which we consider the *null pointer constant*. This constant compares unequal to any other pointer to an object or function.
- 6.3.2.3 §5: An integer may be converted to any pointer type by storing the integer value in the `offset` field of the `MAddress` object and setting the *data* to `null`. According to the C99 standard, converting an integer to a pointer results in an undefined behavior. In our case, the pointer value cannot be dereferenced (the *data* is `null`), which in accordance with the standard.
- 6.3.2.3 §6: Any `MAddress` object may be converted to an integer type, which uses the object representation of a pointer (i.e., the unique ID of the *data* object plus the *offset*).

In ManagedC, each pointer variable starts as a *null pointer*, i.e., the *data* is `null` and hence the pointer cannot be dereferenced. A valid pointer value is produced by applying the address-of operator (`&`) on any C object or by an allocation of memory (e.g., by a manual `malloc` or an automatic stack allocation). In these cases, we create an `MAddress` object where the *data* references the managed allocation. These valid pointer values can then be modified by pointer arithmetic and address computations (i.e., by updating the *offset* of an `MAddress` object; see Section 3.1).

If the *data* of an `MAddress` is `null`, any access to it causes a high-level run-time error. This allows us to detect *null pointer accesses*.

4. Object Access: Spatial and Temporal Safety

In this section we describe how MangedC ensures *spatial* and *temporal* safety of C programs. We distinguishes between a *strict mode* and a *relaxed mode*:

When running ManagedC in *strict mode*, only *well-defined memory accesses* are allowed: A *well-defined access* cannot read from uninitialized memory and the pointer used in the access must reference a valid destination, i.e., the value at the destination must have a type that is expected by the access operation. For example, it is not allowed to dereference a *type-punned* pointer, i.e., a pointer of type B that was casted from a pointer of type A. An access via such a pointer would not resolve to a valid destination. Sections 6.5 §5 and 6.3.2.3 §7 of the C99 standard state that the program shall not access type-punned pointers [2].

When running ManagedC in *relaxed mode*, programmers can read from uninitialized memory and can access any memory destination (*undefined memory access*). For example, a pointer can be casted to a pointer with a different type (*type-punning*) and it is possible to dereference it. In these cases, ManagedC mimics the behavior of plain C compilers but still ensures spatial and temporal safety in the sense of the definitions above.

4.1 Well-Defined Access Operations

In ManagedC, objects of a C program can be represented as *Primitive objects*, *Array objects*, *Function objects*, and *Structured objects* (see Figure 4). All of them can be referenced by an `MAddress` object.

Primitive objects: We represent primitive C values (e.g., `int`, `double`, ...) as Java objects that box a Java primitive value (see Figure 4a). In strict mode, it is only possible to access a *Primitive object* if the access operation is well-defined.

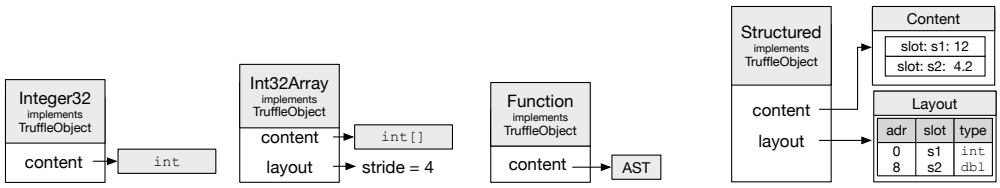
Array objects: We represent C arrays by objects that box a Java array. For example, if C code allocates an `int` array, ManagedC allocates an `Int32Array` object that wraps a Java `int` array (see Figure 4b). Again, in strict mode, it is only possible to access an *Array object* if the access operation is well-defined, i.e., if the type of the access operation matches the element type and the *offset* of the `MAddress` object is aligned (the *offset* must be a multiple of the array's element size).

An array object holds the size of an element in bytes. When an element is accessed, the *offset* of the `MAddress` is divided by this size to get the index of the array element.

Function objects: C functions are represented as ASTs, wrapped in `Function` objects (see Figure 4c). `MAddress` objects can point to a `Function` object (function pointers), which is then executed by ManagedC.

Structured objects: C `structs` and `unions` are represented by so-called `Structured` objects that consist of a *content* and a *layout*:

The *content* is an object of type `DynamicObject` [43] that contains the values of all assigned members. This



(a) Primitive integer. (b) Integer array. (c) Function. (d) Any memory allocation.

Figure 4: Memory-safe representation of C data.

`DynamicObject` was originally designed for dynamic guest language implementations on top of Truffle (e.g. TruffleJS [43] and TruffleRuby [34]) to represent their dynamic data structures, however, ManagedC uses it to represent managed allocations. The `DynamicObject` is an efficient data structure that allows adding and removing members to and from the object at run time (members of a `DynamicObject` are stored in *slots*).

The slots of a `DynamicObject` can contain primitive values, address values, as well as other managed allocations. The *content* of the `Structured` object of `struct S` (see Figure 3) contains two values:

$$\{\text{slot1} \mapsto 12, \text{slot2} \mapsto 4.2\} \tag{1}$$

The *layout* is a map storing the offsets, types and slots of the members and is used to map an *offset* to the corresponding *slot* and its *type* (see Figure 4d). The *layout* of the `Structured` object of `struct S` (see Figure 3) contains two entries:

$$\{0 \mapsto (\text{slot1}, \text{int}), 8 \mapsto (\text{slot2}, \text{double})\} \tag{2}$$

If a C program accesses a `Structured` object with a certain *offset*, the *offset* is mapped to a *slot* of the *content*. For example, when accessing the member `b` of `struct S`, the *layout* maps the offset `8` to *slot2*, which contains an 8-byte `double` value.

In strict mode, it is only possible to access `Structured` objects if the operation is well-defined, i.e., if the *layout* can map an *offset* to a *slot* and if the type of the access operation matches the type of the *slot*.

For non-primitive members (e.g., arrays and `structs`) of a `struct` or `union` we allocate a separate *Array*/`Structured` object and store it into the *slot* of the *content*. When accessing sub-objects (e.g. when a `struct` has an array member), we compute the *offset* within the sub-object and access it in the same way. `Union` member *slots* can be accessed with the types of the `union` members.

4.2 Access Operations in Relaxed Mode

When running in *relaxed* mode, ManagedC allows *accessing uninitialized memory* and *dereferencing type-punned pointers*.

4.2.1 Allocations Without Type Information

Programmers can allocate memory without providing information about its type. In this case, we create a `Structured` object with a *layout* starting in an *uninitialized* state, i.e., if no data was written to this allocation yet, the *layout* does not contain any entries. A write operation then adds a new *slot* to the *content* and creates an entry in the *layout*. In other words, when a pointer `p` is dereferenced to write to a field (e.g. `p->a = 3;`), a new *slot* (e.g., `slot0`) is added to the *content* (here containing the value `3`), and a corresponding entry is made in the *layout* table (e.g., $0 \mapsto (\text{slot0}, \text{int})$).

A read operation that the *layout* cannot map to a *slot* because the memory is uninitialized produces a default value.

4.2.2 Undefined Object Accesses

ManagedC maps the *offset* of an `MAddress` object to a member of a managed allocation. If this is not possible or if the mapped member does not have the type that is expected by the access operation, we call this an *undefined access*:

An undefined read operation to a *Primitive object* returns the same value as a raw memory read would have produced, i.e., we mimic the object representation of native C primitives. Consider the following example:

```
double v = 42.5;
double *d = &v;
int i = ((int *)d)[0];
```

The read operation returns an `int` value that contains the lower 4 bytes of the `double` value `42.5`.

Upon an undefined write access to a *Primitive object*, we first allocate a new `Structured` object and copy the data to its *content*. We also initialize the *layout*: The *layout* for a *primitive object* has one entry. The program finally accesses this `Structured` object and uses it for the rest of the program execution.

In case of an undefined read operation to an *Array object*, ManagedC reads from all elements that overlap with the accessed element and returns the same value as a raw memory read would have produced. Let us assume that we cast an `int` array to a `double*`:

```
int a[5];
double v = ((double *)a)[0];
```

The read operation reads the `int` elements (32 bits) at indexes 0 and 1 and composes their values to a 64 bit `double` value.

Again, undefined write operations transform the *Array object* to a `Structured` object.

A `Structured` object can handle any undefined access operation. We use the *layout* to map the *offset* of an `MAddress` object to a *slot* and hence mimic the object representation of a native allocation on the x86-64 platform.

An undefined read operation reads all *slots* that overlap with the accessed element, composes their values, and returns the bit pattern at the given offset. This produces the same value as a raw memory read from native data would have produced. Let us assume that we cast a pointer `s` (of type `Struct S*`) to a pointer of type `int*` and perform an *undefined* read operation:

```
int *i = (int*)s;
int v = i[2];
i[2] = 13;
```

The read access of `i[2]` should return the first 4 bytes of the `double` value `s->b`. Thus, ManagedC reads *slot2* (the `double` value `s->b`) and returns the first 4 bytes of the value at offset 8 encoded as an `int`.

In an undefined write operation, a C program can partially overwrite one or more members of a `Structured` object. In `i[2] = 13`, the program attempts to write a 4 byte `int` value to offset 8, which is mapped to an 8 byte `double` slot. This means that slots in a `Structured` object need to be partially overwritten, which we implement as follows: We remove all slots that are partially overwritten. Then we add a new slot to the *content* for the new value and also slots for the remaining bytes of the partially overwritten slots.

For the write access `i[2] = 13`, we first remove the entry $8 \mapsto (\text{slot2}, \text{double})$ from our *layout* and also remove `slot2` from the *content*. Afterwards we add two new entries to our *layout*: $\{8 \mapsto (\text{slot2}, \text{int}), 12 \mapsto (\text{slot3}, \text{undefined})\}$. `Slot2` contains the value `13` and `slot3` contains the remaining 4 bytes of the `double` value `4.2`, which was previously stored at offset `8`. The result is a *layout* with three entries:

$$\{0 \mapsto (\text{slot1}, \text{int}), 8 \mapsto (\text{slot2}, \text{int}), \\ 12 \mapsto (\text{slot3}, \text{undefined})\} \quad (3)$$

The *content* contains three values:

$$\{\text{slot1} \mapsto 12, \text{slot2} \mapsto 13, \text{slot3} \mapsto ...\} \quad (4)$$

4.3 Resolving Access Operations at Run Time

In the previous sections we described how we represent objects that are referenced by C pointers as managed allocations. Now we will show how these objects and their members are accessed in the AST. This is done by sending messages to the managed allocations and by resolving these *messages* at run time [15, 18].

```
int *p = malloc(n * sizeof(int));
int **pp = &p;
```

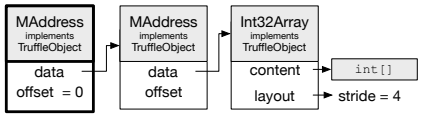


Figure 6: `MAddress` objects representing a chain of pointers.

All kinds of managed allocations are accessed in a uniform way. Every Java object that implements `TruffleObject` can be accessed via *messages*, hence, all managed allocations implement this interface (see Figure 4). There are read and write messages for all primitive C types as well as messages for reading and writing an `MAddress`.
In response to such a message, a `TruffleObject` returns an AST snippet that contains object-specific operations for executing the access operation. For example, a `Structured` object returns an AST snippet that maps an *offset* to a *slot* and eventually accesses this *slot* whereas an `Int32Array` returns an AST snippet that divides an *offset* by 4 (the size of an element) and accesses a Java integer array.

Upon first execution, every message to an object resolves to an object-specific AST snippet (message resolution) [15, 18], which is then inserted into the enclosing AST as a replacement for the message. In other words, message resolution replaces object-independent messages by object-specific ASTs.

During later execution, the value of an `MAddress` can change so that it points to a different kind of managed allocation. In order to detect that, a guard is inserted into the AST that checks the object's layout before accessing the object.

Figure 5 shows the AST for the statement `s->b = 4.2`, where a *WriteDouble* message node is used to write the `double` value `4.2` to the `struct` member `b`. Message resolution replaces the *WriteDouble* node with a *SafeWriteSlot* (a `Structured`-specific AST node). Before the AST accesses `s`, it checks if its data really references a `Structured` object (*is `Structured`*? node). If during execution `s` would change to point to a `DoubleArray` say, the execution would fall back to sending the *WriteDouble* message again, which would be resolved to a `DoubleArray`-specific AST snippet. If the object access is *polymorphic* in the sense that it accesses different kinds of managed allocations over time, ManagedC links the different object-specific AST snippets to a chain like an inline cache [22] and therefore caters for good performance.

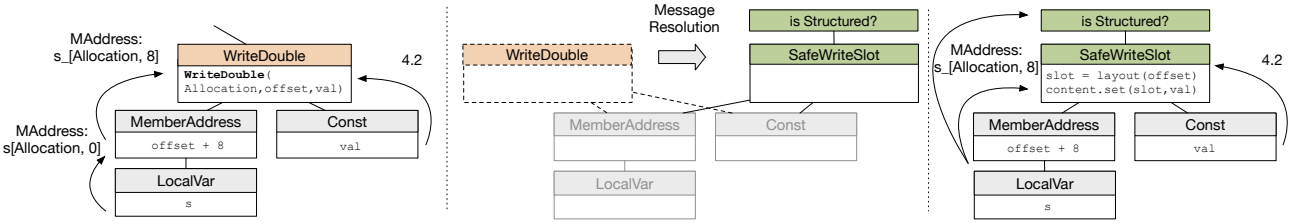


Figure 5: Resolving managed allocation-specific access operations.

The AST snippets returned by `Structured` objects are later specialized according to their execution profile (Section 2.1). This gives good results because of the following observations: First, it is likely that a C program accesses an allocation in a well-defined fashion, which means that the *layout* of a `Structured` object does not change during run time. Second, statements that access an allocation are likely to do so with a constant *offset*. For example, a `struct` access `s->b` always accesses the `Structured` object with the same *offset*.

Thus, we speculate that the *offset* of the `MAddress` and the *layout* of the `Structured` object are constant. So we cache the *slot* and directly access the *content* without any lookup in the *layout*. We guard these assumptions by a run-time check and fall back to the lookup if the check fails.

Object accesses via *messages* and *message resolution* are Truffle mechanisms that were already published in [15, 18]. We use these techniques for accessing managed allocations.

Each managed allocation returns an AST snippet that makes sure that the *offset* is within the bounds of the allocation. These run-time checks ensure *spatial safety*, which prevents errors such as *null pointer dereferences* or *buffer overflows*.

4.4 Allocations and Deallocations

We distinguish between stack allocations (memory that is automatically allocated when a function is called and automatically deallocated when the function returns) and heap allocations (memory that is manually allocated using `malloc`, `calloc`, or `realloc`, as well as memory that lives throughout the entire program execution such as static local variables):

Stack allocations: When a C function is called, a new stack frame is created for its local variables. ManagedC allocates such stack frames as managed allocations (i.e., on the heap and not on the stack). When a function returns, its frame is marked as *deallocated*. It is reclaimed automatically by the GC of the JVM as soon as it is not referenced any longer.

Heap allocations: When a C program manually allocates memory, ManagedC represents it as a managed allocation. When this memory is manually freed, the allocation is marked as *deallocated* but is not removed. If the C program tries to access this allocation later or if it calls `free` twice, we report a high-level run-time error.

We use the memory management of the JVM to guarantee temporal safety: object deallocation is done automatically by the GC if and only if the object is not referenced anymore. Marking managed allocations as deallocated allows us to mimic the behavior of C by simulating deallocations. We mark manged objects as deallocated by setting the *content* reference (see Figure 4) to `null`. We can detect if a pointer is used after the referent has been deallocated, i.e, when a dangling pointer is accessed or an object is deallocated twice. ManagedC also makes sure that only valid allocations are accessed, i.e., `MAddress` objects whose *data* field references a life managed allocation. This ensures temporal safety.

ManagedC can even do away with memory leaks that result from forgetting to deallocate objects. Such errors cannot occur in ManagedC, because the GC automatically frees a managed allocation as soon as it is not referenced any longer. In fact, manual deallocation becomes superfluous, because managed allocations are garbage collected.

5. Applications, Trade-offs, and Future Work

ManagedC allows memory-safe execution of C code in *strict* and in *relaxed* mode: The *strict* mode can be used during development to detect undefined operations of a program, such as accessing type-punned pointers. To run existing source code without modification we offer the *relaxed* mode, which mimics the behavior of industry standard C compilers. In this mode, ManagedC can run existing code that depends on the behavior of industry-standard C compilers without sacrificing spatial or temporal memory safety.

In the following we discuss the trade-offs of ManagedC:

TruffleC limitations: TruffleC is not complete and is missing language features. For example, TruffleC only supports single-threaded programs [16]. As ManagedC is based on TruffleC, this restriction also applies to ManagedC. The missing support of multi-threading is also the reason why this paper evaluates ManagedC using the C99 standard rather than the C11 standard. However, we are convinced that a full implementation of all language features is possible with reasonable effort in the future.

ManagedC limitations: Managed allocations cannot be shared with precompiled native code. Therefore, ManagedC requires that the source code of the entire C program is available and is executed under ManagedC. We provide a Java implementation for functions that are not available in source code (e.g. functions of standard libraries). Rather than doing a native call, ManagedC then uses these Java implementations that substitute the native implementations. ManagedC currently has substitutions for various functions defined in `assert.h`, `limits.h`, `math.h`, `stdarg.h`, `stdbool.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`, and `wchar.h`. The list of standard library substitution is not yet complete, however, our future work will focus on completing this and hence make ManagedC more complete. Under these premises, we can run C programs entirely on top of the JVM without ever accessing native code or native data.

Memory trade-offs: Managed allocations in Java have an object header (16 bytes), which increases their size compared to native allocations. However, Graal's *partial escape analysis* [39] ensures that an object only gets allocated if it escapes its compilation scope. Otherwise Graal performs *scalar replacement* and does not allocate the object on the heap.

Future Work: Our future work will focus on these limitations: First, we will work on completeness of TruffleC, this includes adding features like multi-threading. Second, we will complete the list of substitutions of standard library functions. This work will make TruffleC and ManagedC applicable for full-sized applications.

6. Reaction to Undefined Operations or Memory Errors

ManagedC detects memory errors as well as undefined access operations at run time, immediately before memory is accessed in some illegal way. ManagedC throws a Java exception that describes the error and includes also a stack trace at the error position.

Currently, such Java exceptions are caught, the error and the stack trace are printed, and the program exits. Since the effects of memory-unsafe accesses and undefined memory accesses are not specified by the C standard, this behavior is fully compliant. This kind of error reporting could either be used during testing to find bugs, or at run time to further enforce correctness. Figure 7 shows an example of how memory-unsafe accesses are reported. The implementation of `doWork` erroneously loops with the condition `i <= N`, rather than `i < N`, which causes it to access one element beyond the end of the array within the loop. ManagedC reports this as `BufferOverflowError` and tells the user that the error occurred in `doWork` which was called by `main`.

7. Performance Evaluation

We measured the overhead of memory-safe managed allocations compared to unsafe native allocations by running a number of benchmarks.

7.1 Benchmarks

We used benchmarks from the SciMark benchmark suite[1] and from the Computer Language Benchmarks Game[2] that heavily access C structures and arrays. The benchmarks consist of a Fast Fourier Transformation (FFT), a Jacobi successive overrelaxation (SOR), a Monte Carlo integration (MC), a sparse matrix multiplication (SM), a dense LU matrix factorization (LU), a simulation of the N-body problem (NB), a generation of random DNA sequences (FA), a computation of the spectral norm of a matrix (SN), as well as the Fannkuch (FK) and Mandelbrot (MB) benchmarks, which both do a lot of integer and array accesses. These benchmarks suit well for our evaluation as they heavily access different managed allocations such as `Structured` objects, *array objects*, and *primitive objects*.

7.2 Experimental Setup

We ran the benchmarks on an Intel Core i7-4770 quad-core 3.4GHz CPU running 64 Bit Debian 7 (Linux3.2.0-4-amd64) with 16 GB of memory. TruffleC as well as ManagedC are based on Graal revision `5b24a15988fe` from the official OpenJDK Graal repository[3]. The evaluation reports scores (higher is better) for each benchmark and its configuration. Figure 8 shows the results of the benchmarks. The score is the proportion of the execution count of the benchmark and the time needed (executions/second).

We are interested in peak performance of long running applications after an initial warm-up. Hence, we measure the performance of C programs after dynamic compilation. An evaluation of warm-up performance of a Truffle language implementation is out of scope for this work.

We executed every benchmark 10 times after an initial warm-up of 50 iterations to arrive at a stable peak performance and calculated the average for each configuration using the arithmetic mean. We report the standard deviation by showing error bars in the chart. The x-axis of the chart shows the different benchmarks. The y-axis of the chart shows the average scores (higher is better) of the benchmarks, where 100 is the performance of GCC with optimization level *O0*. Where we summarize across different benchmarks (*Mean*) we report a geometric mean [1].

TruffleC uses the same memory management as plain C compilers, i.e., it allocates data on the native heap and accesses it via *unsafe* access operations. ManagedC replaces these *unsafe* allocations with *safe* managed allocations. In this evaluation we want to compare the performance differ-

[1] `http://math.nist.gov/scimark2/index.html`

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

[3] `http://openjdk.java.net/projects/graal/`

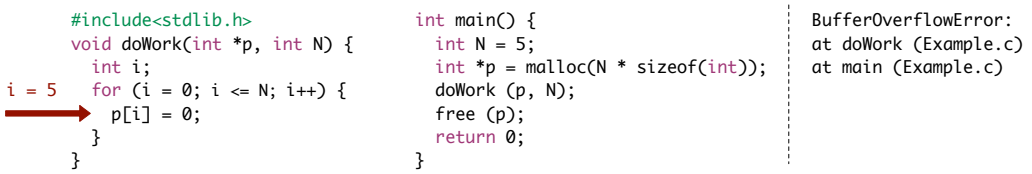


Figure 7: A memory-unsafe action being detected and reported.

ence between *unsafe* native allocations and *safe* managed allocations, hence we focus our evaluation on TruffleC compared to ManagedC. A detailed comparison of the TruffleC performance to industry-standard C compilers is out of scope for this work and can be found in [16]. In this paper, we also add the numbers for GCC with optimization level *O0* (baseline) and the numbers of *GCC best* (i.e., the best performance achieved with the optimization levels *O1*, *O2* and *O3*). This gives some ideas of how ManagedC compares to industry-standard C compilers.

ManagedC executes these benchmarks in *strict* mode (see Sections 5 and 4.1).

7.3 Discussion

ManagedC performs various checks when accessing a managed allocation. For example, when accessing an *Array object*, it does the following:

1. It checks if the managed allocation is not `null` and if the allocation is an *Array object* (see Section 4.3).
2. It checks if the *Array object* is live (see Section 4.4).
3. It checks if the index is in-bounds.
4. Finally, it checks if the type of the access operation matches the type of the array element.

Dynamic compilers (such as the Graal compiler) can eliminate many of these checks and can therefore reduce the run-time overhead [12, 13, 38, 44]. Using conditional elimination [38], the Graal compiler can remove redundant run-time checks by merging them. Conditional elimination reduces the number of conditional expressions by performing a data-flow analysis over the IR graph and pruning conditions that can be proven to be true. Loop invariant code motion [12, 13] moves run-time checks out of loops if the compiler can prove that a condition is loop-invariant. In this case, the static number of checks remains the same, but a check outside a loop is likely to be executed less often than inside a loop. Array bounds check elimination [44] fully removes bounds checks if it can prove that they never fail. Also, whenever possible, it moves bounds checks out of loops.

In addition to that, the Graal compiler speculatively moves guards out of loops even if they have a control dependence in the loop. In this case, the (possibly more strict) condition before the loop implies the guard in the loop, which allows removing the check inside the loop.

The performance evaluation shows that the safe execution of a C program with ManagedC is on average 7% faster compared to the unsafe execution with TruffleC. The performance difference can be explained as follows:

ManagedC performs equally well as TruffleC when the dynamic compiler is able to remove all run-time checks (FA, SN, MB, FFT). In other words, there is no performance difference between an unsafe memory access and a safe managed object access when the dynamic compiler can remove all run-time checks. Only if the dynamic compiler cannot remove all run-time checks (NB, LU), the overhead of ManagedC is 17% for LU and 24% for NB compared to TruffleC. Managed allocations allow the Graal compiler to apply a sophisticated partial escape analysis with scalar replacement [39] on the objects of a C program. If the compiler can eliminate the allocation of managed objects, ManagedC can even outperform TruffleC, the performance improvement varies between 9% (FK) and 38% (SOR).

If we compare the performance of ManagedC to the unsafe execution of C code compiled with *GCC best*, the overhead is 15% on average. The performance difference varies between 25% speedup to 106% slowdown of ManagedC compared to GCC best. A major part of this overhead is due to the performance difference between TruffleC and *GCC best* (TruffleC has an overhead of 23% compared to *GCC best*). Hence, if the TruffleC performance improves, the gap between ManagedC and *GCC best* will also get smaller. We are convinced that ongoing work on Truffle's dynamic compiler will improve the performance of TruffleC and thus also of ManagedC.

8. Related Work

Memory safety of C is a well researched area and can be categorized into *software-based* approaches and *hardware-based* approaches. ManagedC is a software-based approach, thus we focus on research in this area rather than on hardware-based research such as [6, 10, 26, 27, 32, 35, 36, 41]. Like existing literature surveys [25], we distinguish between *pointer-based approaches* and *object-based approaches*. Finally, we discuss the Boehm-Demers-Weiser GC, which provides temporal safeness. A survey of related work to interpretation of C code can be found in [16].

8.1 Pointer-based Approaches

Our technique is inspired by *pointer-based metadata* approaches: Pointer-based approaches [5, 24–26, 28, 29] store

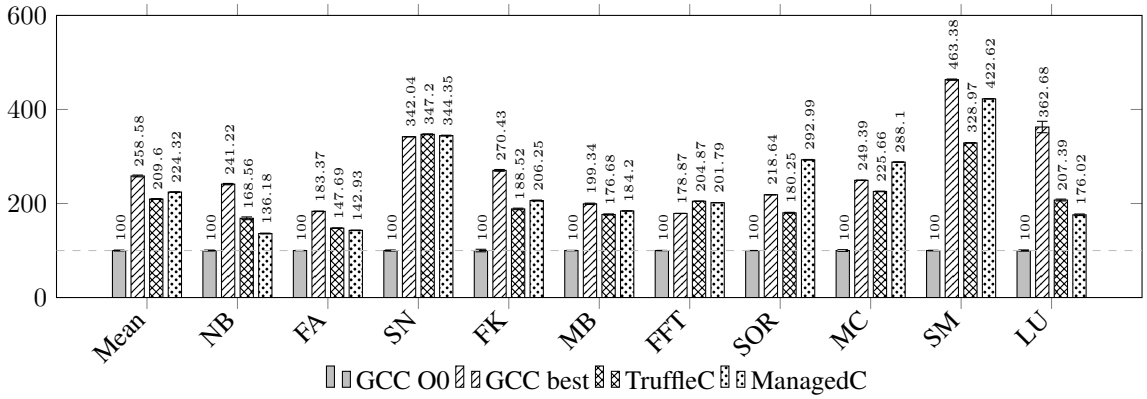


Figure 8: Performance numbers of TruffleC and ManagedC, relative to GCC *O0*. Higher is better.

additional information for each pointer of a C program so that pointers become multi-word values (*fat pointers*) [5, 29] that hold the actual pointer value along with metadata (e.g., the upper and the lower bounds of the referenced object). Pointer arithmetic then modifies the actual pointer value, but the metadata remains unchanged. When a pointer is dereferenced, the actual pointer value is checked against the bounds of the object, which ensures spatial safety.

To ensure temporal safety, every allocated object gets a unique identifier (capability) [5]. These capabilities stay in existence even after the deallocation of an object, which allows checking the pointer's validity when it is dereferenced.

SafeC [5] allows the detection of spatial and temporal memory errors by using fat pointers that encode the pointer value, base and size information, a storage class (Heap, Local, or Global allocation), and a capability to the referent. The run-time checks introduced by SafeC cause a run-time overhead between 130% and 540% [5].

CCured [28, 29] classifies pointers in three categories: *SAFE* pointers, which cannot be used for pointer arithmetic, array indexing or type casts and cause almost no run-time overhead; *SEQ* pointers, which are fat pointers that allow pointer arithmetic, array indexing and primitive casts; and *WILD* pointers, which support arbitrary casts but have additional metadata and require run-time checks. CCured programs have an overhead in the range of 3% to 87%.

Nagarakatte et al. [24] describe SoftBound [24] and CETS [25], which use compile-time transformations for detecting spatial and temporal safety violations. These approaches keep the metadata in a separate metadata space (in contrast to fat pointers), which retains memory layout compatibility. They report an average run-time overhead of 116% [25].

An `MAddress` object can be seen as a fat pointer (to ensure spatial safety) and the memory management of the JVM ensures temporal safety. The novelty in our approach is that we transferred the idea of *fat pointers* to a C interpreter (i.e., TruffleC), which is implemented in Java. It uses a dynamic compiler (i.e., Graal) and a sophisticated automated memory management (the JVM garbage collector). This allows us to ensure spatial and temporal safety with little effort: We use a *layout* to map the *offset* of an `MAddress` to the corresponding member of a managed allocation and eventually do a Java member access, which ensures spatial safety. We mark freed objects as deallocated, which allows us to ensure temporal safety. The deallocation of the object itself is then done by the GC of the JVM. Also, our approach is source-compatible with regular C programs and does not require any changes in them because we mimic the behavior of industry-standard C compilers. We dynamically compile ManagedC ASTs with a state-of-the-art dynamic compiler, which is very good at removing access bounds checks or moving them out of hot loops. After an initial warm-up and dynamic compilation of the AST we can report an average overhead of 15% (from 25% speedup to 106% slowdown) compared to programs that are compiled with the best possible optimization level of GCC.

8.2 Object-based Approaches

Object-based approaches track information about each object such as its status (allocated/deallocated) or its bounds and stores it in an auxiliary data structure [9, 11, 14, 20, 23, 30, 33]. Spatial and temporal safety is ensured by mapping pointer values to the tracked information (e.g., using a splay tree [23] or a trie [30]) and by checking that pointer arithmetic and pointer dereferencing fall within the bounds of the object. Jones and Kelly [23] report an overhead of $400 - 500\%$ compared to GCC performance.

Purify [20] traps every memory access by instrumenting the object code of a program and by allocating "red zones" before and after each allocation. They report a run-time overhead of 450% compared to the GCC performance.

Eigler's mudflap system [14] inserts an additional pass into GCCs normal compilation to instrument the C code and to assert a validity predicate at every use of a pointer. Mudflap caches the lookup in the auxiliary data structure

and has an average run-time overhead of 400% compared to unsafe execution.

Dhurjati and Adve [11] use a fine-grained partitioning of memory to provide run-time bounds checking for arrays and strings. They report an overhead of 12% compared to an unchecked execution.

Ruwase and Lam [33] prevent buffer overflows by introducing out-of-bound objects for all out-of-bound pointers. Any pointer derived from an out-of-bound object is bound-checked before it can be dereferenced. They report a run-time overhead in the range of $1\% - 130\%$.

Our system represents pointer values by `MAddress` objects that use a Java reference to refer to the allocation. We detect spatial memory errors when the *offset* of the `MAddress` object cannot be mapped to a member of the referent and temporal errors when the referent is not allocated (*data* is `null`) or the referent is marked as deallocated. Hence, we consider our approach as distinct from object-based approaches.

8.3 Boehm-Demers-Weiser GC

The Boehm-Demers-Weiss GC [19] is a conservative GC for C and C++ that can provide temporal safeness. It uses a mark-sweep algorithm and provides incremental and generational collection. It works with unmodified C programs by replacing native memory allocations with *GC allocations* and removing `free` calls completely. Also, it can run in a *leak detection* mode that allows ensuring temporal safety.

Like ManagedC, this approach automatically deallocates memory that is not used anymore. However, the architecture of the two approaches is different: ManagedC executes C code on a memory-safe runtime via AST interpretation. It introduces `MAddress` objects to represent pointers and allocates managed objects, which allows reusing the GC of the runtime without additional effort.

9. Summary

In this paper we presented ManagedC, a Truffle-based C interpreter with dynamic compilation that can execute C programs while ensuring spatial as well as temporal safety. By reusing the memory management of the JVM it was straightforward to build a memory-safe C interpreter. Rather than using native data, ManagedC allocates Java objects on the Java heap and uses `MAddress` objects to represent pointers. ManagedC ensures spatial safety by mapping the *offset* of an `MAddress` object to a member of a Java object and by performing a Java object access. Furthermore, only `MAddress` objects that point to a valid managed allocation can be dereferenced. Whenever a C application deallocates memory, we mark it as deallocated to ensure that it is not used anymore. Thus, we can detect dangling pointers and double deallocations and can therefore ensure temporal safety. ManagedC is in average 7% faster than (unsafe) TruffleC. Compared to the best GCC performance, ManagedC has an average overhead of 15%. We can achieve this performance because the dynamic compiler of the Truffle framework is especially good at removing the run-time checks that ensure safety.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs, the Institute for System Software and the Institute for Networks and Security at the Johannes Kepler University Linz for their valuable feedback on this work and on this paper. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results. *Communications of the ACM*, 1986.

[2] C99 Standard: ISO/IEX 9899:TC3. `www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`, 2007.

[3] Clang/LLVM. `http://clang.llvm.org/`, 2014.

[4] GCC (GNU C Compiler). `http://gcc.gnu.org/`, 2014.

[5] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. *SIGPLAN Not.*, 1994. . URL `http://doi.acm.org/10.1145/773473.178446`.

[6] W. Chuang, S. Narayanasamy, and B. Calder. Accelerating Meta Data Checks for Software Correctness and Security. *Journal of Instruction-Level Parallelism*, 2007.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Security*, 1998.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition.*, 2000. .

[9] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *ACM SIGOPS Operating Systems Review*, 2007.

[10] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. *ACM SIGOPS Operating Systems Review*, 2008.

[11] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, 2006.

[12] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. VMIL '13, 2013. . URL `http://doi.acm.org/10.1145/2542142.2542143`.

[13] G. Duboscq, T. Würthinger, and H. Mössenböck. Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. PPPJ '14, 2014. . URL `http://doi.acm.org/10.1145/2647508.2647521`.

[14] F. C. Eigler. Mudflap: Pointer Use Checking for C/C+. In *GCC Developers Summit*. Citeseer, 2003.

[15] M. Grimmer. High-Performance Language Interoperability in Multi-Language Runtimes. SPLASH '14, 2014.

[16] M. Grimmer, M. Rigger, R. Schatz, L. Stadler, and H. Mössenböck. TruffleC: Dynamic Execution of C on a Java Virtual Machine. PPPJ '14, 2014. . URL http://dx.doi.org/10.1145/2647508.2647528.

[17] M. Grimmer, T. Würthinger, A. Wöß, and H. Mössenböck. An Efficient Approach for Accessing C Data Structures from JavaScript. ICOOOLPS '14, 2014. . URL http://dx.doi.org/10.1145/2633301.2633302.

[18] M. Grimmer, C. Seaton, T. Wuerthinger, and H. Moessenboeck. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. MODULARITY '15, 2015.

[19] Hans-J. Boehm. A garbage collector for C and C++. http://www.hboehm.info/gc/, 2015.

[20] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[21] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. *SIGPLAN Not.* . URL http://doi.acm.org/10.1145/143103.143114.

[22] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91*. Springer, 1991. . URL http://dx.doi.org/10.1007/BFb0057013.

[23] R. W. Jones and P. H. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *AADEBUG*. Citeseer, 1997.

[24] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. PLDI '09, 2009. . URL http://doi.acm.org/10.1145/1542476.1542504.

[25] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler Enforced Temporal Safety for C. ISMM '10, 2010. . URL http://doi.acm.org/10.1145/1806651.1806657.

[26] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. ISCA '12. IEEE Computer Society, 2012. URL http://dl.acm.org/citation.cfm?id=2337159.2337181.

[27] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. CGO '14, 2014. . URL http://doi.acm.org/10.1145/2544137.2544147.

[28] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Code. POPL '02, 2002. . URL http://doi.acm.org/10.1145/503272.503286.

[29] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, May 2005. . URL http://doi.acm.org/10.1145/1065887.1065892.

[30] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of VEE'07*, 2007.

[31] Oracle. OpenJDK: Graal project. http://openjdk.java.net/projects/graal/, 2013.

[32] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA 2005*. IEEE, 2005.

[33] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *NDSS*, 2004.

[34] C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at Full Speed. In *DYLA'14*, 2014.

[35] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, 2012.

[36] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference, General Track*, 2005.

[37] M. S. Simpson and R. K. Barua. MemSafe: ensuring the spatial and temporal memory safety of Cat runtime. *Software: Practice and Experience*, 2013. . URL http://dx.doi.org/10.1002/spe.2105.

[38] L. Stadler, G. Duboscq, H. Mössenböck, T. Würthinger, and D. Simon. An Experimental Study of the Influence of Dynamic Compiler Optimizations on Scala Performance. SCALA '13, 2013. . URL http://doi.acm.org/10.1145/2489837.2489846.

[39] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. CGO '14, 2014. . URL http://doi.acm.org/10.1145/2544137.2544157.

[40] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013. .

[41] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA 2007*. IEEE, 2007.

[42] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*, 2000.

[43] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. PPPJ '14, 2014. . URL http://dx.doi.org/10.1145/2647508.2647517.

[44] T. Würthinger, C. Wimmer, and H. Mössenböck. Array Bounds Check Elimination for the Java HotSpot Client Compiler. PPPJ '07, 2007. . URL http://doi.acm.org/10.1145/1294325.1294343.

[45] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST Interpreters. DLS '12, 2012. . URL http://doi.acm.org/10.1145/2384577.2384587.

[46] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to rule them all. In *SPLASH'13*, 2013.