Increasing the Robustness of C Libraries and Applications through Run-time Introspection

Abstract

In C, low-level errors such as buffer overflow and use-after-free are a major problem since they cause security vulnerabilities and hard-to-find bugs. Libraries cannot apply defensive programming techniques since objects (e.g., arrays or structs) lack run-time information such as bounds, lifetime, and types. To address this issue, we devised introspection functions that empower C programmers to access run-time information about objects and variadic function arguments. Using these functions, we implemented a more robust, source-compatible version of the C standard library that validates parameters to its functions. The library functions react to otherwise undefined behavior; for example, when detecting an invalid argument, its functions return a special value (such as `-1` or `NULL`) and set the `errno`, or attempt to still compute a meaningful result. We demonstrate by examples that using introspection in the implementation of the C standard library and other libraries prevents common low-level errors, while also complementing existing approaches.

1 Introduction

Since the birth of C almost 50 years ago, programmers have written many applications in it. Even the advent of higher-level programming languages has not stopped C's popularity, and it remains widely used as the second most popular programming language [32]. However, C provides few safety guarantees and suffers from unique security issues that have disappeared in modern programming languages. The most severe issue in C are buffer overflow errors, where a pointer that exceeds the bounds of an object is dereferenced [7]. Other security issues include use-after-free errors, invalid free errors, reading of uninitialized memory, and memory leaks. Numerous approaches exist to prevent such errors in C programs by detecting these illegal patterns statically or during run-time, or by making it more difficult to exploit them [33, 31, 40]. When an error happens, run-time approaches abort the program which is more desirable than risking incorrect execution, potentially leaking user data, executing injected code, or corrupting program state.

However, we believe that programmers (especially those writing libraries) could better respond to illegal actions in the application logic, if they could check such invalid actions at run-time and prevent them from happening. For example, if programmers could check that an access would go out-of-bounds

- in a user application, they could prevent the access, and could call an application-specific error handling function that would print an error message and abort execution.
- in a server application, they could log the error and ignore the invalid access to maintain availability of the system.
- in the C standard library, they could set the global integer variable `errno` to an error code, for example to `EINVAL` for invalid arguments. Furthermore, a *special value* (such as `-1` or `NULL`) could be returned to indicate that something went wrong.

In this paper, we present a novel approach that empowers C programmers to query properties of an *object* (primitive value, struct, array, union, or pointer), so that they can perform explicit sanity checks and react accordingly to invalid arguments or states. These properties comprise the bounds of an object, the memory location, the number of arguments of a function with varargs, and whether an object can be used in a certain way (e.g., called as a function that expects and returns an `int`). Ultimately, this increases the robustness of libraries and applications, defined as "[t]he degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [14].

We implemented such introspection capabilities for Safe Sulong [23], an interpreter with a dynamic com-

piler for C. Safe Sulong prevents buffer overflows, use-after-free, and other memory errors by checking accesses and aborting execution upon an invalid action. By using introspection, programmers can check and prevent illegal actions, which enables them to override the default behavior of aborting the program when an illegal action occurs in order to maintain availability. Additionally, explicit checks prevent lurking flaws that could otherwise stay undetected. For example, even in the case that a function does not actually access an invalid position in the buffer, most memory safety approaches cannot detect when a wrong array size is passed to the function. Using introspection, the passed array size can be validated against the actual one. The presented approach is *complementary* to other means of preventing memory errors, and does not aim to replace them.

As a case study, we demonstrate how the introspection functions facilitate re-implementing the C standard library (`libc`) to validate input arguments. We use this `libc` in Safe Sulong as a source-compatible, more robust drop-in replacement for the GNU C Library. In contrast to the GNU C Library and other implementations, it can prevent illegal actions and react by returning special values and setting *errno*, or attempting to compute a meaningful result. Our standard library correctly implements the specification, since the introspection checks only react to actions that would cause undefined behavior.

In summary, this paper contributes the following:

- We present introspection functions designed to allow programmers to prevent illegal actions that are specific to C (Section 3).
- We demonstrate how we implemented the introspection functions in Safe Sulong, a C interpreter with a dynamic compiler (Section 4).
- As a case study, we show how using introspection increases the robustness of the `libc` that we distribute with Safe Sulong (Section 5).

2 Background

In C, the lack of type and memory safety causes many problems such as hard-to-find bugs and security issues. Moreover, manual memory management puts the burden of deallocating objects on the user. Consequently, C programs are plagued by vulnerabilities that are unique to the language. Errors cause undefined behavior, so compiled code can crash, compute unexpected results, and corrupt or read neighboring objects [35, 36]. It is often not possible to design C functions in such a way that they are secure against usage errors, since they cannot validate passed arguments or global data. The following is an incomplete list of errors and vulnerabilities in C programs that we target in this paper.

```
void read_number(char* arr, size_t length) {
  int i = 0;
  if (length == 0) return;
  int c = getchar();
  while (isdigit(c) && (i + 1) < length) {
    arr[i++] = c;
    c = getchar();
  }
  arr[i] = '\0';
}
// ...
char buf[10];
read_number(buf, -1);
printf("%s\n", buf);
```

Figure 1: Out-of-bounds error example

Out-of-bounds errors. Out-of-bounds accesses in C rank under the most dangerous software errors [26, 7], since unlike higher-level languages, C does not specify automatic bounds checks. Additionally, objects have no run-time information attached to them, so functions that operate on arrays require array size arguments. Alternatively, they need conventions such as terminating an array by a special value.

Figure 1 shows a typical buffer overflow. The `read_number()` function reads digits entered by the user into the passed buffer `arr` and validates that it does not write beyond its bounds. However, its callee passes `-1` as the `length` parameter which is (through the `size_t` type) treated as the unsigned number `SIZE_MAX`. Thus, the bounds check is rendered useless and if the user enters more than nine digits the `read_number()` function overflows the passed buffer. A recent similar real-world vulnerability includes CVE-2016-3186, where a function in libtiff casted a negative value to `size_t`. As another example, in CVE-2016-6823 a function in ImageMagick caused an arithmetic overflow that resulted in a wrong image size. Both errors resulted in a subsequent buffer overflow.

Memory management errors. Objects that are allocated in different ways (e.g., on the stack or by `malloc()`) have different lifetimes which influences how they can be used. For example, it is forbidden to access memory after it has been freed (otherwise known as an access to a *dangling pointer*). Other such errors include freeing memory twice, freeing stack memory or static memory, or calling `free()` on a pointer that points somewhere into the middle of an object [22]. Figure 2 shows an example of a use-after-free and a double-free error. Firstly, when `err` is non-zero, the allocated pointer `ptr` is freed, and later accessed again as a dangling pointer in `logError()`. Secondly, the code fragment attempts to free the pointer again after logging the error,

```
char* ptr = (char*) malloc(SIZE * sizeof(char));
if (err) {
  abrt = 1;
  free(ptr);
}
// ...
if (abrt) {
  logError("operation aborted", ptr);
  free(ptr);
}
// ...
void logError(const char* message, void* ptr) {
    logf("error while processing %p", ptr);
}
```

Figure 2: Use-after-free error which is based on an example from the CWE wiki

which results in a double free vulnerability. C does not provide mechanisms to retrieve the lifetime of an object, which would allow checking and preventing such conditions. Consequently, use-after-free errors frequently occur in real-world code. For example, in CVE-2016-4473 the PHP Zend Engine attempted to free an object that was not allocated by one of the `libc`'s allocation functions. Other recent examples include a dangling pointer access and a double free error in OpenSSL (CVE-2016-6309 and CVE-2016-0705).

Variadic function errors. Variadic functions in C rely on the user to pass a count of variadic arguments or a format string. Furthermore, a user must pass the matching number of objects of the expected type. Figure 3 shows an example that uses variadic arguments to print formatted output, similar to C's `sprintf()` function. It is based on a function taken from the PHP Zend Engine. As arguments, the function expects a format string `fmt`, the variadic arguments `ap`, and a buffer `xbuf` to which the formatted output should be written. To use the function, a C programmer has to invoke a macro to set up and tear down the variadic arguments (respectively `va_start()` and `va_end()`). Using the `va_arg()` macro, `xbuf_format_converter()` can then directly access the variadic arguments. The example shows how a string can be accessed (format specifier `"%s"`) that is then inserted into the buffer `xbuf`.

The function uses the format string to determine how many variadic arguments should be accessed. For example, for a format string `"%s %s"` the function attempts to access two variadic arguments that are assumed to have a string type. Accessing a variadic argument via `va_arg()` usually manipulates a pointer to the stack and pops the number of bytes that correspond to the specified data type (`char *` in our example). Attackers can exploit that the function cannot verify the number and the types of the passed variadic arguments in so-called *format string attacks* where the function reads or writes the stack due

```
static void xbuf_format_converter(void *xbuf, ↩
    const char *fmt, va_list ap) {
  char *s = NULL;
  size_t s_len;
  while (*fmt) {
    if (*fmt != '%') {
      INS_CHAR(xbuf, *fmt);
    } else {
      fmt++;
      switch (*fmt) {
        // ...
        case 's':
        s = va_arg(ap, char *);
        s_len = strlen(s);
        break;
        // ...
      }
      INS_STRING(xbuf, s, s_len);
    }
  }
}
```

Figure 3: Example usage of variadic functions taken from the PHP Zend Engine

to nonexistent arguments [6, 27].

In CVE-2015-8617, this function was the sink of a vulnerability that existed in PHP-7.0.0. The `zend_throw_error()` function called `xbuf_format_converter()` with a `message` string that was under user-control. Following, an attacker could use format specifiers without matching arguments to read and write from memory, and thus execute arbitrary code. As another example, in CVE-2016-4448 a vulnerability in libxml2 existed because format specifiers from untrusted input were not escaped.

Lack of type safety. Due to the lack of type safety a user cannot ensure whether an object referenced by a pointer corresponds to its expected type [16]. Figure 4 demonstrates this for function pointers. The `apply()` function expects a function pointer that accepts and returns an `int`. It uses the function to transform all elements of an array. However, its callee might pass a function that returns a double; a call on it would result in undefined behavior. Such "type confusion" cannot be avoided when calling a function pointer, since objects have no types attached that could be used for validation.

Unterminated strings. Unterminated strings are a problem, since the string functions of the `libc` (and sometimes also application code) rely on strings ending with a `'\0'` (null terminator) character. However, C standard library functions that operate on strings lack a common convention on whether to add a null terminator [21]. Additionally, it is not possible to verify whether a string is properly terminated without potentially causing buffer overreads. Figure 5 shows an example of an unterminated string vulnerability. The `read` function reads a file's contents into a string `inputbuf`.

```
int apply(int* arr, size_t n, int f(int arg1)) {
  if (f == NULL) {
    return -1;
  }
  for (size_t i = 0; i < n; i++) {
    arr[i] = f(arr[i]);
  }
  return 0;
}

double square(int a) {
  return a * a;
}

apply(arr, 5, square);
```

Figure 4: Example for type confusion

```
read(cfgfile, inputbuf, MAXLEN);
char buf[MAXLEN];
strcpy(buf, inputbuf);
puts(buf);
```

Figure 5: Example fragment that may produce and copy an unterminated string

After the call, `inputbuf` is unterminated if the file was unterminated or if `MAXLEN` was exceeded. This is likely to cause an out-of-bounds write in `strcpy()`, since it copies characters to `buf` until a null terminator occurs. Recent similar real-world vulnerabilities include CVE-2016-7449 where `strlcpy()` was used to copy untrusted (potentially unterminated) input in GraphicsMagick. Further examples include CVE-2016-5093 and CVE-2016-0055 where strings were not properly terminated in the PHP Zend Engine as well as in Internet Explorer and Microsoft Office Excel [19].

Unsafe functions. Some functions in common libraries such as the `libc` have been designed in a way that "can never be guaranteed to work safely" [5, 2]. The most prominent example is the `gets()` function, which reads user input from `stdin` into a buffer passed as an argument. Since `gets()` lacks a parameter for the size of the supplied buffer, it cannot perform any bounds checking and overflows the user-supplied buffer if the user input is too large. Although C11 replaced `gets()` with the more robust `gets_s()` function, legacy code might still require the unsafe `gets()` function. In general, functions that lack size arguments which prevents safe access to arrays cannot be made safe without breaking source and binary compatibility.

3 Introspection functions

To empower C programmers to validate arguments and global data, we devised introspection functions to query properties of C objects and the current function (see Appendix A). The functions only allow programmers to inspect objects and not to manipulate them; therefore, the presented functions are not a full reflection interface.

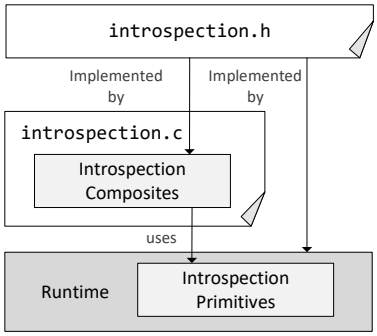

Figure 6: The introspection interface is implemented through introspection primitives in the runtime and through composites (written in C)

We designed the functions specifically to enable their users to prevent buffer overflow, use-after-free, and other common errors specific to C. Through introspection, users can validate certain properties (memory location, bounds, and types) before performing an operation on an object. Additionally, introspection allows quering the number of passed variadic arguments and validating their types.

We built introspection based on several *introspection primitives* (see Figure 6). These primitives are a minimal set of C functions that require run-time support. We also designed *introspection composites*, which are implemented as normal C functions, and are based on the introspection primitives or on other composites. The introspection functions that we expose to the user contain both selected primitives and composites. In the following, we denote internal functions that are private to the implementation with an underscore prefix.

3.1 Object Bounds

Most importantly, we provide functions that enable the user to perform bounds checks before accessing an object. Simply providing a function that returns the size of an object is insufficient, since a pointer can point to the middle of an object. Instead, we require the runtime to provide two functions to return the space (in bytes) to the left and to the right of a pointer target: `_size_left()` and `_size_right()`. Their result is only defined for *legal* pointers which we define as pointers that are pointing to valid objects (not `INVALID`, see Section 3.2).

Figure 7 illustrates the function return values when passing a pointer to the middle of an integer array

```
int *arr = malloc(sizeof(int) * 10);
int *ptr = &(arr[4]);
printf("%ld\n", size_left(ptr)); // prints 16
printf("%ld\n", size_right(ptr)); // prints 24
```

Figure 7: Example on how to query the space to the left and to the right of a pointee

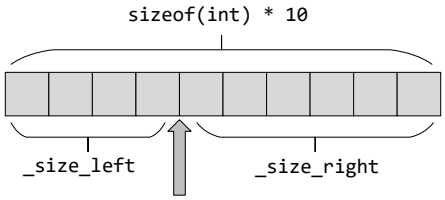

Figure 8: Memory Layout of the Example in Figure 7

to these functions. For the pointer to the fourth element of the ten-element integer array, `_size_left()` returns `sizeof(int) * 4`, and `_size_right()` returns `sizeof(int) * 6`. Figure 8 shows the corresponding memory layout. On an architecture where an `int` is four bytes large the functions return 16 and 24, respectively.

We do not expose these two functions to the user, but base the composite functions `size_left()` and `size_right()` on them which return `-1` if the passed argument is not a legal pointer or out-of-bounds. Figure 9 shows the implementation of `size_left()`. First, the function checks that the pointer is legal using `location()` (see Section 3.2). Then, it checks that the space to the left and to the right right of the pointer is not negative, that is, the pointer is in-bounds. If both checks succeed, the function returns the space to the left of the pointer using `_size_left()`; otherwise, it returns `-1`.

Figure 10 shows how using `size_right()` improves `read_number()`'s robustness (see Figure 1): if `arr` is a valid pointer, but points to memory that cannot hold `length chars`, we can prevent the out-of-bounds access by aborting the program. Note, that the check also detects lurking bugs since it aborts even if less than `length` characters are read. If `arr` is not a valid pointer the return value of `size_right()` is `-1`.

```
long size_left(const void *ptr) {
  if (location(ptr) == INVALID) {
      return -1;
  }
  bool inBounds = _size_right(ptr) >= 0 &&
    _size_left(ptr) >= 0;
  if (!inBounds) {
      return -1;
  }
  return _size_left(ptr);
}
```

Figure 9: Implementation of `size_left()` using the functions `location()`, `_size_left()`, and `_size_right()`

```
void read_number(char* arr, size_t length) {
  int i = 0;
  if (length == 0) return;
  if (size_right(arr) < length) abort();
  // ...
}
```

Figure 10: By using the `size_right()` function we can avoid out-of-bounds accesses in `read_number()`

3.2 Memory location

Querying the memory location of an object (e.g., stack, heap, global data) allows a user to obtain information about the lifetime of an object. For example, it enables users to prevent use-after-free errors by detecting whether an object has already been freed. Another use case is validating that no stack memory is returned by a function. A user can also check whether a location refers to dynamically allocated memory to ensure that `free()` can be safely called on it. For this purpose, we provide a function `location()` that determines where an object lies in memory.

The function returns one of the following enum constants:

- `INVALID` locations denote `NULL` pointers or deallocated memory (freed heap memory or dead stack variables). Programs must not access such objects.
- `AUTOMATIC` locations denote non-static stack allocations. Functions must not return allocated stack variables that were declared in their scope, since they become `INVALID` when the function returns. Also, stack variables must not be freed.
- `DYNAMIC` locations denote dynamically allocated heap memory created by `malloc()`, `realloc()`, or `calloc()`. Only memory allocated by these functions can be freed.
- `STATIC` locations denote statically allocated memory such as global variables, string constants, and static local variables. Static compilers usually place such memory in the text or data section of an executable. Programs must not free statically allocated memory.

Figure 11 shows how differently allocated memory relates to the enum constants used by `location()`.

Based on `location()`, we provide a function `freeable()` to conveniently check whether an allocation can be freed. As Figure 12 demonstrates, an freeable

```
int a; // STATIC
void func() {
  static int b; // STATIC
  int c; // AUTOMATIC
  int* d = malloc(sizeof(int) * 10); // DYNAMIC
  free(d); // INVALID
}
```

Figure 11: Example on how the `location()` enum constants relate to variables in a program

```
bool freeable(const void *ptr) {
  return location(ptr) == DYNAMIC &&
    _size_left(ptr) == 0;
}
```

Figure 12: By using `location()` and `_size_left()` we can check whether an object can be freed

object's location must be `DYNAMIC` and its pointer must point to the beginning of an object. Figure 13 shows how we can use the `freeable()` function to improve the robustness of the code fragment shown in Figure 2. It ensures that freeing the pointee is valid, and thus prevents invalid free errors such as double freeing memory. Still, the `logError()` function may receive a dangling pointer as an argument. To resolve this, we can check in `logError()` whether the pointer is valid (see Figure 14).

Note, that some libraries such as OpenSSL use custom allocators to manage their memory. Custom allocators are out of scope for this paper, but could be supported by providing source-code annotations for allocation and free functions; this information could then be used by the runtime to track the memory. The annotations for the allocation functions would need to specify how to compute the size of the allocated object, and the location of the allocated memory. Additionally, it might be desirable to add further enum constants, for example, for shared, file-backed, or protected memory. We omitted additional constants for simplicity.

```
char* ptr = (char*) malloc(SIZE * sizeof(char));
if (err) {
  abrt = 1;
  if (freeable(ptr)) free(ptr);
}
// ...
if (abrt) {
  logError("operation aborted", ptr);
  if (freeable(ptr)) free(ptr);
}
```

Figure 13: By using the `freeable()` function we can avoid double-free errors

```
void logError(const char* message, void* ptr) {
  if (location(ptr) == INVALID) {
    log("dangling pointer passed to logError!");
  } else {
    logf("error while processing %p", ptr);
  }
}
```

Figure 14: By using the `location()` function we can avoid use-after-free errors

3.3 Type

We provide a function that allows the programmer to validate whether an object is *compatible to* (can be treated as being of) a certain type. Such a function enables programmers to check whether a function pointer actually points to a function object (and not to a `long`, for example), and whether it has the expected function signature. As another example, programmers can use the function as an alternative to `size_right()` and `size_left()` to verify that a pointer of a certain type can be dereferenced.

C only has a weak notion of types which makes it difficult to design expressive type introspection functions. For example, it is ambiguous whether a pointer of type `int*` that points to the middle of an integer array should be considered as a pointer to an integer or as a pointer to an integer array. Another example is heap memory which lacks a dynamic type; although programmers usually coerce them to the desired type, objects of different types can be stored. Even worse, when writing to memory, objects can be partially overwritten; for instance, half of a function pointer can be overwritten with an integer value making it difficult to decide whether the pointer points is still a valid function pointer.

Instead of assuming that a memory region has a specific type we designed a function that allows the programmer to check whether the memory region is compatible with a certain type (similar to [16]). The `try_cast()` function expects a pointer to an object as the first argument and tries to cast it to the `Type` specified by the second argument. If the run-time determines that the cast is possible it returns the passed pointer, otherwise it returns `NULL`. The cast is only possible if the object can be read, written to, or called as the specified type.

The `Type` object is a recursive struct which makes it possible to describe nested types (known as type expressions [1]). For example, a function pointer with an `int` parameter and `double` as the return type can be represented by a tree of three `Type` structs. The root struct specifies a `function` type and references a struct with an `int` type as the argument type as well as a struct with a `double` type as the return type. Since manually constructing `Type` structs is tedious, we specified an *optional*

```
int apply(int* arr, size_t n, int f(int arg1)) {
  if (size_right(arr) < sizeof(int) * n || ↩
      try_cast(&f, type(f)) == NULL) {
    return -1;
  }
  for (size_t i = 0; i < n; i++) {
    arr[i] = f(arr[i]);
  }
  return 0;
}
```

Figure 15: By using `try_cast()` we can ensure that we can perform an indirect call on the function pointer in `apply()`

operator `type()`. As an argument, it requires an expression *example value* whose declared type is returned as a `Type` run-time data structure. The declared type is a compile-time property, so we want to resolve the `type()` operator during compile-time; following, the user cannot take `type()`'s address and call it indirectly. The operator is similar to the GNU C extension `typeof` which yields a type that can be directly used in variable declarations or casts.

Figure 15 shows how the type introspection functions make the function `apply()` (see Figure 4) more robust: `apply()` uses `try_cast()` to check whether the runtime can treat its first argument as the specified function pointer. Its second argument is the `Type` object that the `type` operator constructs from the declared function pointer type. The `try_cast()` function returns the first argument if it is compatible to the specified function pointer type; otherwise, it returns `NULL`. Besides preventing calling invalid function pointers, `apply()` prevents out-of-bounds accesses by validating the array size.

The `try_cast()` function is similar to C++'s `dynamic_cast()`. However, we want to point out that C++'s `dynamic_cast()` works only for class checks (which are well-defined), while our approach works for all C objects. We believe that the exact semantics of `try_cast()` should be implementation-defined, since runtime information could differ between implementations. For example, depending on the runtime's knowledge of data execution prevention, it might either allow or reject the cast of a non-executable `char` array filled with machine instructions to a function pointer. Also, different use cases exist and a security-focused runtime might have more sources of run-time information and be more restrictive than a performance-focused runtime. For example, a traditional runtime would (for compatibility) allow dereferencing a hand-crafted pointer, as long as it corresponds to the address of an object, while a security-focused runtime could disallow it. Thus, depending on the underlying runtime, compiler, and ABI the `try_cast()` can return different results.

```
double avg(int count, ...) {
  if (count == 0 || count != count_varargs()) {
    return 0;
  }
  int sum = 0;
  for (int i = 0; i < count; i++) {
    int *arg = get_vararg(i, type(&sum));
    if (arg == NULL) {
      return 0;
    } else {
      sum += *arg;
    }
  }
  return (double) sum / count;
}
```

Figure 16: By using `count_varargs()` and `get_varargs()` we can use variadics in a robust way

3.4 Variadic arguments

Our introspection interface provides macros to query the number of variadic arguments and enables programmers to access them in a type-safe way. They are implemented as macros and not as functions, since they need to access the current function's variadic arguments. The introspection macros make using variadic functions more robust and are, for example, effective to prevent format string attacks [6].

Querying the number of variadic arguments can be achieved by calling `count_varargs()`. The standard `va_arg()` macro reads values from the stack while assuming that they correspond to the user-specified type. As a robust alternative, introspection composites can use `_get_vararg()` to directly access the passed variadic arguments by an argument index. To access the variadic arguments in a type-safe way, we introduced a `get_vararg()` macro that is exposed to the user and expects a type that it uses to call `try_cast()`. Figure 16 shows an example of a function that computes the average of `int` arguments. It uses `count_varargs()` to verify the number of variadic arguments and ensures that the $\texttt{i}^{\texttt{th}}$ argument is in fact an `int` by calling `get_vararg()` with `type(&sum)`. If an unexpected number of parameters or an object with an unexpected type is passed, the function returns `0`.

For backwards compatibility, we used the introspection intrinsics to make the standard vararg macros (`va_start()`, `va_arg()`, and `va_end()`) more robust. Firstly, `va_start()` initializes the retrieval of variadic arguments. We modified it such that it allocates a struct (using the `alloca()` stack allocation function) and populates it using `_get_vararg()` and `count_varargs()`. The struct comprises the number of variadic arguments, an array of addresses to the variadic arguments, and a counter to index them. Secondly, `va_arg()` retrieves the next variadic argument. We modified it such that it

checks that the counter does not exceed the number of arguments, increments the counter, indexes the array, and casts the variadic argument to the specified type using `try_cast()`. If the cast succeeds, the argument is returned; otherwise a call to `abort()` exits the program. Finally, `va_end()` performs a cleanup of the data initialized by `va_start()`. We modified it such that it resets the variadic arguments counter.

Using the enhanced vararg macros improves the robustness of the `xbuf_format_converter()` function (see Figure 3), since the number of format specifiers has to correspond with the number of arguments. Thus, it would be impossible to exploit the function through format string attacks. Note that the modified standard macros abort when they process invalid types or an invalid number of arguments, while the intrinsic functions allow users to react to invalid arguments in other ways.

4 Implementation

We implemented the introspection primitives in Safe Sulong [23], which is an execution system for low-level languages such as C. Its core is an interpreter written in Java that runs on top of the JVM. Unlike its counterpart Native Sulong [24], Safe Sulong uses Java objects to represent C objects. By relying on Java's bounds and type checks, Safe Sulong efficiently and reliably detects out-of-bounds accesses, use-after-free, and invalid free. When detecting such an invalid action, it aborts execution of the program. Section 4.1 gives an overview of the system, and Section 4.2 describes how we implemented the introspection primitives.

4.1 System Overview

Figure 17 shows the architecture of Safe Sulong. It comprises the following components (most of them are available under open-source licenses):

Clang. Safe Sulong executes LLVM Intermediate Representation (IR), which represents C functions in a simpler, but lower-level format. LLVM is a flexible compilation infrastructure [18], and we use LLVM's front end Clang to compile the source code (libraries and the user application) to the IR. LLVM is available at `http://llvm.org/`.

LLVM IR. LLVM IR retains all C characteristics that are important for the content of this paper. It can, for instance, contain external function definitions and function calls. By executing LLVM IR, Safe Sulong can execute all languages that can be compiled to this IR, including C++ and Fortran. It is even possible to execute programs without available source code using binary translators that convert binary code to LLVM IR. For example, MC-Semantics [9] and QEMU [4] support x86, and LLBT [28] supports the translation of ARM code. Binary libraries that are converted to LLVM IR can then profit from enhanced libraries that Safe Sulong can execute such as the enhanced `libc`.

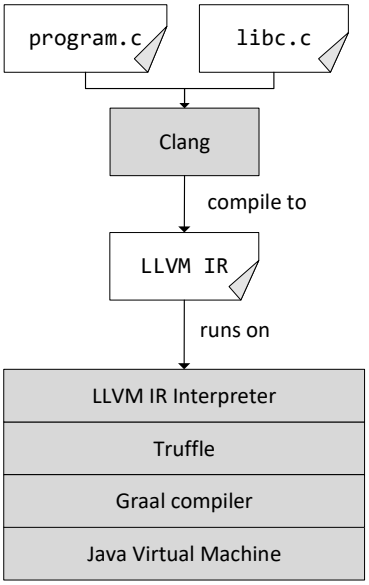

Figure 17: Overview of Safe Sulong

Truffle. We used Truffle [38] to implement our LLVM IR interpreter. Truffle is a language implementation framework written in Java. To implement a language, a programmer writes an Abstract Syntax Tree (AST) interpreter in which each operation is implemented as an executable node. Nodes can have children that parent nodes can execute to compute their results. Truffle is available at `https://github.com/graalvm/truffle`.

Graal. Truffle uses Graal [39], a dynamic compiler, to compile frequently executed Truffle ASTs to machine code. Graal applies aggressive optimistic optimizations based on assumptions that are later checked in the machine code. If an assumption no longer holds, the compiled code *deoptimizes* [13], that is, control is transferred back to the interpreter and the machine code of the AST is discarded.. Graal is available at `https://github.com/graalvm/graal-core`.

LLVM IR Interpreter. The LLVM IR interpreter is the core of Safe Sulong; it executes both the user application as well as the enhanced `libc`. First, a front end parses the LLVM IR and constructs a Truffle AST for each LLVM IR function. Then, the interpreter starts executing the `main` function AST, which can invoke

other ASTs. During execution, Graal compiles frequently executed functions to machine code. We have not yet made Safe Sulong's interpreter available under an open-source license. However, the basic Sulong interpreter without the safety features is available at `https://github.com/graalvm/sulong`.

JVM. The system can run efficiently on any JVM that implements the Java based JVM compiler interface (JVMCI [25]). JVMCI supports Graal and other compilers written in Java.

4.2 Objects and Introspection

The LLVM IR interpreter uses Java objects instead of native memory to represent LLVM IR objects (and thus C objects). Figure 18 illustrates our type hierarchy. Every LLVM IR object is a `ManagedObject` which has subclasses for the different types. For example, an `int` is represented by a `I32` object which stores the `int`'s value in the `value` field. Similarly, there are subclasses for arrays, functions, pointers, structs, and other types. In the introspection implementation we needed to expose properties of these Java objects to the user:

Bounds. The `ManagedObject` class provides a method `getByteSize()` which returns the size of an object. We represent pointers as objects of a `ManagedAddress` class that holds a reference to the pointee and a pointer offset that is updated through pointer arithmetics (`pointee` and `pointerOffset`). For example, for the pointer to 4^{th} element of an integer array in Figure 7, the `pointerOffset` is 16 and `pointee` references a `I32Array` that holds a Java `int` array (see Figure 19). If a user wanted to dereference the pointer, the interpreter would compute `pointerOffset / sizeof(int)` to index the array. We implemented the `size_right()` function by `ptr.pointee.getByteSize() - ptr.pointerOffset`.

Memory location. Although `ManagedObjects` live on the managed Java heap, the `location()` function needs to return their *logical* memory location. This location is stored in a field of the `ManagedObject` class. Depending on whether an object is allocated through `malloc()`, as a global variable, as a static local variable, or as a constant, we assign a different flag to its `location` field; calls to `free()` and deallocation of automatic variables assign `INVALID` to this field. For instance, for an integer array that lives on the stack, the interpreter allocates an `I32Array` object and assigns `AUTOMATIC` to its `location`. After leaving the function scope, its `location` is updated to `INVALID`. When the `location()` function is called with a pointer to the integer array, it returns the `location` field's value.

Type. For implementing the `try_cast()` function, we check if the type of the passed object (given by its Java class) is compatible with the type specified by the `Type` struct. For example, to check whether we can call a pointer as a function with a certain signature, we first compare the passed pointer with a `Type` that describes this signature. If the pointer references a Sulong object of type `Function`, the argument and return types are compared. This is possible, since `Function` objects retain run-time information about their arguments and return types, which can be retrieved via the method `getSignature()`.

Variadic arguments. In Safe Sulong, a caller explicitly passes its arguments as an object array (i.e., a Java array of known length) to its callee. Based on the function signature and the object array, the callee can count the variadic arguments to implement `count_varargs()` and extract them to implement `_get_vararg()`.

5 Case Study: Safe Sulong's Standard Library

We distribute an implementation of the `libc` together with Safe Sulong. This `libc` uses introspection for checks that make it more robust against usage errors and attacks. For instance, its functions identify invalid parameters that would otherwise cause out-of-bounds accesses or use after frees. In such a case, the functions return special values to indicate that something went wrong, and then set `errno` to an error code. However, for functions where no special value can be returned (e.g. because the return type is `void`) setting `errno` would be meaningless, since functions are allowed to arbitrarily change `errno` even if no error occurred. In these cases, the functions still attempt to compute a meaningful result. Such behavior is compliant with the C standards, since we prevent illegal actions with undefined behavior that could crash the program or corrupt memory.

For applications and libraries that run on Safe Sulong, the distribution format is LLVM IR and not executable code. Our standard library improvements are binary compatible on the IR level which means that users do not have to recompile their applications when using our enhanced `libc`. In addition, this standard library is source-compatible, so a user is not required to change the program when using it. In the following, we give an overview of our enhanced library functions:

String functions. We made all functions that operate on strings (`strlen()`, `atoi()`, `strcmp()`, `printf()`, ...) more robust by computing meaningful results even when a string lacks a null terminator. They do not read or write outside the boundaries of unterminated strings, which

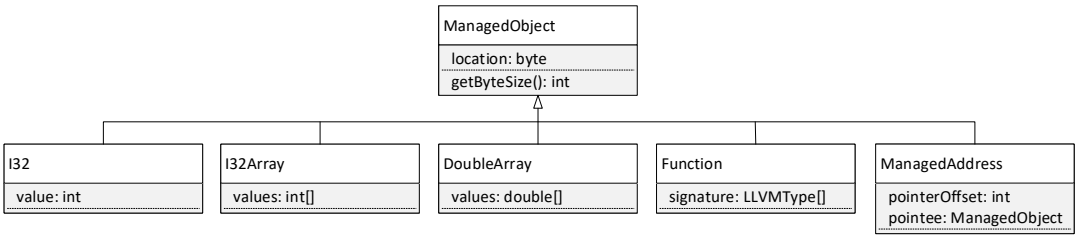

Figure 18: Sketch of the `ManagedObject` Class Hierarchy

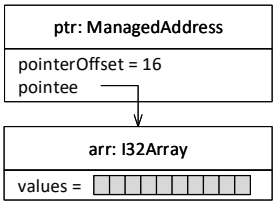

Figure 19: Representation of a pointer to the 4^{th} element of an int array (resulting from pointer arithmetics)

```
size_t strlen(const char *str) {
  size_t len = 0;
  while (size_right(str) > 0 && *str != '\0') {
    len++;
    str++;
  }
  return len;
}
```

Figure 20: Robust implementation of `strlen()` that also works for unterminated strings

makes them robust against common string vulnerabilities. The functions increase availability of the system since unterminated strings passed to the `libc` do not cause crashes. Note, that when a function outside the `libc` relies on a terminated string, it will still cause an out-of-bounds access which causes Safe Sulong to abort execution. Thus, increased availability does not harm confidentiality (e.g., by leaking data of other objects) and integrity (e.g., by overwriting other objects).

For instance, Figure 20 shows how we improved `strlen()` by preventing buffer overflows when iterating over the string, as well as non-legal pointers (where `size_right()` returns `-1`). For terminated strings, `strlen()` iterates until the first '`\0`' character to return the length of the string. For unterminated strings, the function cannot return `-1` to indicate an error, since `size_t` is unsigned, so we also do not set `errno`. Instead, it iterates until the end of the buffer and returns the size of the string until the end of the buffer.

```
void *realloc(void *ptr, size_t new_size) {
  if (location(ptr) == INVALID) {
    return malloc(new_size);
  } else {
    void *new = malloc(new_size);
    size_t old_size = size_right(ptr);
    size_t copy_size = new_size > old_size ? ↩
        old_size : new_size;
    memcpy(new, ptr, copy_size);
    if (freeable(ptr)) {
      free(ptr);
    }
    return new;
  }
}
```

Figure 21: Robust implementation of `realloc()` that prevents invalid frees and other memory errors

The enhanced string functions also allow the execution of the code fragment in Figure 5. Even though the source string may be unterminated, `strcpy()` will not produce an out-of-bounds read, since it stops copying when reaching the end of the source or destination buffer. The call to `puts()` also works as expected, and prints the unterminated string.

Functions that free memory. We made functions that free memory (`realloc()` and `free()`) more robust by checking if their argument can safely be freed. For example, Figure 21 shows how we implemented `realloc()` in terms of `malloc()` and made it more robust. Apart from verifying that the argument can be freed using `freeable()`, it also uses `location()` to check whether the passed pointer is valid. In Safe Sulong, `malloc()` is written in Java and allocates a Java object. By using the introspection functions we could conveniently and robustly implement `realloc()` in C, without having to maintain a list of allocated and freed objects.

Format string functions. We made input and output functions that expect format strings more robust. Examples are the `printf()` functions (`printf()`, `fprintf()`, `sprintf()`, `vfprintf()`, `vprintf()`, `vsnprintf()`, `vsprintf()`) and the `scanf()` functions (`scanf()`, `fscanf()`, ...). These functions expect

```
void qsort(void *base, size_t nitems, size_t ↩
    size, int (*f)(const void *, const void*)) {
  int (*verifiedPointer)(const void *, const ↩
      void*) = try_cast(&f, type(f));
  if (size_right(base) < nitems * size || ↩
      verifiedPointer == NULL) {
    errno = EINVAL;
  } else {
    // qsort implementation
  }
}
```

Figure 22: Robust `qsort()` implementation that checks whether it can call the supplied function pointer

```
char *gets(char *str) {
  int size = size_right(str);
  return gets_s(str, size == -1 ? 0 : size);
}
```

Figure 23: Robust implementation of `gets()` that uses the more robust `gets_s()` in its implementation

format strings that contain format specifiers, and matching arguments that are used to produce the formatted output. Since the functions are variadic, we added checks to verify that the number of format specifiers is equal to the actual number of arguments using `count_varargs()`. Also, the functions use `get_vararg()` to verify the argument types. This prevents format string vulnerabilities and out-of-bounds reads in the format string as demonstrated in the implementation of `strlen()`.

Higher-order functions. We enhanced functions that receive function pointers such as `qsort()` or `bsearch()`. Figure 22 shows how `qsort()` can use `try_cast()` to verify that `f` is a function pointer that is compatible with the specified signature. Furthermore, the functions verify that no memory errors such as buffer overflows can occur.

gets() and gets_s(). While C11 replaced the `gets()` function with `gets_s()`, Safe Sulong can still provide a robust implementation for `gets()` (see Figure 23). Since `size_right()` can determine the size of the buffer to the right of the pointer we can call it and use the returned size as an argument to the more robust `gets_s()` function. If the pointer is not legal we pass `0` which `gets_s()` handles as an error. We also made `gets_s()` more robust against erroneous parameters (see Figure 24). By using `size_right()` we can validate that the size parameter `n` is at least as large as the remaining space right to the pointer. The check prevents buffer overflows for `gets()` and `gets_s()`, and also passing of dead stack memory or freed heap memory.

```
char *gets_s(char *str, rsize_t n) {
  if (size_right(str) < (long) n) {
    errno = EINVAL;
    return NULL;
  } else {
    // original code
  }
}
```

Figure 24: Robust implementation of `gets_s()` that verifies the passed size argument

6 Limitations

We presented an implementation of the introspection functions, and showed how it can be used to provide an enhanced version of the C standard library. However, we identified the following limitations:

Performance. Using our enhanced `libc`, Safe Sulong's performance is currently 2.3× slower than executables compiled by Clang with all optimizations turned on (`-O3` flag). To measure the performance impact of introspection in our safe implementation of the `libc`, we ran six benchmarks of the Computer Language Benchmark Game (see `https://benchmarksgame.alioth.debian.org/`) and the whetstone benchmark, once with the enhanced `libc` and once with the original `libc`. We could not find any observable overhead. This comes partly from the fact that Sulong's interpreter caches objects and classes that occur during execution, a technique known as *dispatch chains* [20]. Our dynamic compiler identifies checks against such cached objects as redundant (e.g., through conditional elimination [29]) and eliminates them. In future work, we want to evaluate Safe Sulong on larger benchmarks and lower its performance overhead.

Runtime support. Introspection requires information about run-time properties of objects in the program. While interpreters and virtual machines often maintain this information, runtimes that execute native programs compiled by static compilers such as Clang or GCC do not. We want to point out that debug metadata (obtained by compiling with the `-g` flag) cannot provide per-object type information needed for introspection. However, it has been shown that per-object information (such as types) can be added with low costs to static compilation approaches [16], and hence make it feasible to implement the introspection functions in their runtimes.

Fully-reflective environment. Our approach exposes introspection functions that retrieve information about objects and variadic functions. It lacks a fully-fledged reflection mechanism, since we only expose information that we deemed useful for preventing common low-level

security issues. However, our implementation has no conceptual limitations that would restrict exposing other dynamic information, or would allow the manipulation of objects.

Other security errors. Our approach cannot be used to check all actions that can be performed in C. For example, the introspection interface lacks functions to determine read and write permissions for memory locations, and to identify uninitialized stack variables which must not be read. Such errors are less commonly exploited, or result in buffer overflows that the user can again check and prevent. Thus, we omitted such functions.

C is used for efficiency. Two of the C/C++ tenets are that "you don't pay for what you don't use" [30], and to "trust the programmer" [3]. Hence, programmers often eschew checks that are possible even without introspection functions [11]. Furthermore, it has been argued that security techniques that introduce overheads of roughly more than 10% do not tend to gain wide adoption in production [31]. However, C library functions often crash or compute erroneous results. For example, for 2016 an all-time high of 1326 overflow vulnerabilities was recorded [8]. In the context of the Internet of Things [37], this number is likely to grow as increasingly many devices are exposed to the web. Consequently, we believe that there is a need for the safe execution of legacy C code (at the expense of performance) as an alternative to porting programs to safer languages.

Existing code. Some approaches allow enhancing existing libraries that are only available as binaries [11]. By using the enhanced `libc` instead of other C standard libraries, our approach makes the interaction between the application and `libc` safer. However, it requires the user to actively make use of introspection to make application functions more robust.

7 Related Work

C Memory safety approaches. For decades, academia and industry has been coming up with approaches to tackle errors specific to C, especially memory safety errors. Thus, there is a plethora of approaches that deal with these issues, both static and run-time approaches, both hardware- and software-based. We consider our approach as a run-time approach since the checks (specified by programmers in their programs) are executed during run time. Existing run-time approaches include Address Space Layout Randomization (ASLR), canaries, data execution prevention, data space randomization, and bounds checkers. Literature already extensively describes these approaches and provides a historical overview of memory errors and defense mechanisms [33], an investigation of the weaknesses of current memory defense mechanisms including a general model for memory attacks [31], and a survey of vulnerabilities and run-time countermeasures [40]. Using introspection to prevent memory errors is a novel approach that is complementary to existing approaches. It is complementary, since the programmer can check for and prevent an invalid action; if the check is omitted and an invalid access occurs, an existing memory safety solution could still prevent the access. This allows, for example, maintaining availability of a system.

Run-time types for C. The closest and most recent related work is `libcrunch` [16], a system that detects type cast errors at run time. It is based on `liballocs` [15], a run-time system that augments Unix processes with allocation-based types. `libcrunch` provides an `__is_a()` introspection function that exposes the type of an object. It uses this function to validate type casts and issues a warning on unsound casts. In contrast to our approach, `libcrunch` checks for invalid casts automatically, so the `__is_a()` function is not exposed to the programmer nor are there other introspection functions. However, we believe that the system could be extended to provide additional run-time information that could be used to implement the introspection primitives. Typical overheads of collecting and using the type information are between 5-35%, which demonstrates that introspection functions are feasible in static compilation approaches.

Our `try_cast()` introspection function resembles `librunch`'s `__is_a()` function. However, the exact semantics of our `try_cast()` function are dependent on the runtime, while `__is_a()` is strictly specified. `libcrunch` pragmatically assumes that all memory has a type, and infers types for untyped allocations such as those allocated by `malloc()`. To accommodate real world code, `libcrunch` relaxes some of the rules about which type casts are possible and notes that it has not always been straightforward to decide which type casts should be considered valid. While `libcrunch`'s cast assumptions work well for checking type errors, we also wanted to allow other use cases of `try_cast()` and thus refrained from a strict specifiction of it. Note, however, that a run-time that uses `libcrunch` could implement the `try_cast()` function using `__is_a()`.

Static vulnerability scanners. Static vulnerability scanners can identify calls to unsafe functions such as `gets()` depending on a security policy specified in a vulnerability database [34]. Such approaches have to conservatively decide whether a call is allowed, instead of validating parameters at run-time through introspection, such as our approach does. Nowadays, most compilers issue a warning when they identify

a call to an unsafe function such as `gets()`, but not necessarily for other, slightly safer functions, such as `strcpy()`.

Fault injection to increase library robustness. Fault injection approaches generate a series of test cases that exercise library functions in an attempt to trigger a crash (or other erroneous behavior) in them. After identifying a non-robust function, these approaches allow programmers to manually or automatically harden their libraries by introducing checks that verify the function parameters. For instance, HEALERS [11, 12] automatically generates a wrapper that sits between the application and its shared libraries to handle or prevent illegal parameters. To check the bounds of heap objects passed to the functions, the approach instruments `malloc()` and stores bounds information. In contrast to our solution, the approaches above support pre-compiled libraries. However, they can only generate wrapper checks where run-time information is explicitly available in the program. Additionally, they refrain from allowing the user to specify the action in case of an error, and always set *errno* and return an error code.

Replacing (parts of) the libc. SFIO [17] is a `libc` replacement and addresses several of its problems. It mainly improved completeness and efficiency, however, it also introduced safer routines for functions that operate on format strings. Additionally, the SFIO standard library functions are more consistent in their arguments and argument order, and thus less error-prone to use than some of the `libc` functions. [21] presented the less error-prone `strlcpy()` and `strlcat()` functions as replacements for the `strcpy()` and `strncat()` functions. Unlike our improved C standard library, these approaches lack source compatibility.

Safer implementation of library functions. To prevent format string vulnerabilities in the `printf` family of functions, FormatGuard [6] uses the preprocessor to count the arguments to variadic functions during compile time, and checks that the number complies with the actual number at run time. FormatGuard replaces the `printf` functions in the C standard library with more secure versions, while retaining compatibility with most programs. From a user perspective, FormatGuard is similar to Safe Sulong's standard library, since both provide more robust C standard library functions. However, our approach only works for runtimes that implement the introspection primitives, while StackGuard works for arbitrary compilers and runtimes. On the other hand, our approach can also verify bounds, memory location, and types of objects.

Restricting buffer overflows in library functions. Libsafe [2] replaces calls to unsafe library functions (such as `strcpy()` and `gets()`) by wrappers that ensure that potential buffer overflows are contained within the current stack frame. It can only prevent stack buffer overflows, since it checks that write accesses do not extend beyond the end of the buffer's stack frame. In contrast to that, approaches exist that only protect against heap buffer overflows caused by C standard library functions [10]. By intercepting C standard library calls, the approach keeps track of heap memory allocations and performs bounds checking before calling the C standard library functions that operate on buffers. Both approaches work with any existing pre-compiled library, but do not protect against all kinds of buffer overflows. With our approach, a programmer can implement checks that prevent both heap and stack overflows, and use the introspection interface to also prevent use-after-free and other errors.

8 Conclusion

We presented an introspection interface for C that programmers can use to make their applications and libraries more robust. The introspection functions expose properties of objects (bounds, memory location, and type) as well as properties of variadic functions (number of variadic arguments and their types). We described an implementation of the introspection primitives in Safe Sulong, a system that provides memory-safe execution of C code. We demonstrated that the approach is complementary to existing memory safety approaches since programmers can use it to react to and prevent errors in the application logic. Finally, we showed how we used the introspection interface to provide a safe, source-compatible C standard library.

References

[1] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

[2] BARATLOO, A., SINGH, N., AND TSAI, T. Libsafe: Protecting critical elements of stacks. *White Paper* (1999). `http://www.research.avayalabs.com/project/libsafe`.

[3] C99 COMMITTEE. *Rationale for International Standard–Programming Languages–C. Revision 5.10*. 2003.

[4] CHIPOUNOV, V., AND CANDEA, G. Dynamically translating x86 to llvm using qemu. Tech. rep., École polytechnique fédérale de Lausanne, 2010.

[5] COMMON WEAKNESS ENUMERATION. Cwe-242: Use of inherently dangerous function. `https://cwe.mitre.org/data/definitions/242.html`.

[6] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium* (2001), vol. 91, Washington, DC.

[7] COWAN, C., WAGLE, F., PU, C., BEATTIE, S., AND WALPOLE, J. Buffer overflows: Attacks and defenses for the vulnerability of

the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings* (2000), vol. 2, IEEE, pp. 119–129.

[8] CVEDETAILS.COM. Cve details. vulnerabilities by type, 2016. `https://www.cvedetails.com/vulnerabilities-by-types.php`.

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

[10] FETZER, C., AND XIAO, Z. Detecting heap smashing attacks through fault containment wrappers. In *Reliable Distributed Systems, 2001. Proceedings. 20th IEEE Symposium on* (2001), IEEE, pp. 80–89.

[11] FETZER, C., AND XIAO, Z. An automated approach to increasing the robustness of c libraries. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on* (2002), IEEE, pp. 155–164.

[12] FETZER, C., AND XIAO, Z. A flexible generator architecture for improving software dependability. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on* (2002), IEEE, pp. 102–113.

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

[14] IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (Dec 2010), 1–418.

[15] KELL, S. Towards a dynamic object model within unix processes. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2015), ACM, pp. 224–239.

[16] KELL, S. Dynamically diagnosing type errors in unsafe code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2016), ACM, pp. 800–819.

[17] KORN, D. G., AND VO, K.-P. Sfio: Safe/fast string/file io. In *USENIX Summer* (1991), pp. 235–256.

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

[19] LU, K. Analysis of cve-2016-0059 - microsoft ie information disclosure vulnerability discovered by fortinet, 2016. `https://blog.fortinet.com/2016/02/19/analysis-of-cve-2016-0059-microsoft-ie-information-disclosure-vulnerability-discovered-by-fortinet`.

[20] MARR, S., SEATON, C., AND DUCASSE, S. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. *SIGPLAN Not. 50*, 6 (June 2015), 545–554.

[21] MILLER, T. C., AND DE RAADT, T. strlcpy and strlcat-consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 175–178.

[22] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not. 45*, 8 (June 2010), 31–40.

[23] RIGGER, M. Sulong: Memory safe and efficient execution of llvm-based languages. In *ECOOP 2016 Doctoral Symposium* (2016).

[24] RIGGER, M., GRIMMER, M., WIMMER, C., WÜRTHINGER, T., AND MÖSSENBÖCK, H. Bringing low-level languages to the jvm: Efficient execution of llvm ir on truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2016), VMIL 2016, ACM, pp. 6–15.

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

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

[27] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium* (2001), pp. 201–220.

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

[29] STADLER, L., DUBOSCQ, G., MÖSSENBÖCK, H., WÜRTHINGER, T., AND SIMON, D. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala* (2013), ACM, p. 9.

[30] STROUSTRUP, B. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[31] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. Sok: Eternal war in memory. In *Proceedings of SP '13* (2013), pp. 48–62.

[32] TIOBE. Tiobe index for november 2016, 2016. `http://www.tiobe.com/tiobe-index/`.

[33] VAN DER VEEN, V., DUTT SHARMA, N., CAVALLARO, L., AND BOS, H. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses* (2012), RAID'12, pp. 86–106.

[34] VIEGA, J., BLOCH, J.-T., KOHNO, Y., AND MCGRAW, G. Its4: A static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference* (2000), IEEE, pp. 257–267.

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

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

[37] WEBER, R. H. Internet of things–new security and privacy challenges. *Computer law & security review 26*, 1 (2010), 23–30.

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

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

[40] YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Comput. Surv. 44*, 3 (June 2012), 17:1–17:28.

A Introspection functions

Table 1 shows the functions and macros of the introspection interface. Internal functions that are private to the implementation are denoted with an underscore prefix.

Object bounds functions		
`long _size_right(void *)`	Primitive internal	Returns the space in bytes from the pointer target to the end of the pointed object. This function is undefined for illegal pointers.
`long _size_left(void *)`	Primitive internal	Returns the space in bytes from the pointer target to the beginning of the pointed object. This function is undefined for illegal pointers.
`long size_right(void *)`	Composite	Returns the remaining space in bytes to the right of the pointer. Returns -1 if the pointer is not legal or out-of-bounds.
`long size_left(void *)`	Composite	Returns the remaining space in bytes to the left of the pointer. Returns -1 if the pointer is not legal or out-of-bounds.
Memory location functions		
`Location location(void *)`	Primitive	Returns the kind of the memory location of the referenced object. Returns -1 if the pointer is `NULL`.
`bool freeable(void *)`	Composite	Returns whether the pointer is freeable (i.e., `DYNAMIC` non-null memory; pointer referecing the beginning of an object).
Type functions		
`void* try_cast(void *, struct Type *)`	Primitive	Returns the first argument if the pointer is legal, if it is within bounds, and if the referenced object can be treated as being of the specified type, `NULL` otherwise.
Variadic function macros		
`int count_varargs()`	Primitive	Returns the number of variadic arguments that are passed to the currently executing function.
`void* _get_vararg(int i)`	Primitive internal	Returns the $\texttt{i}^{\texttt{th}}$ variadic argument (starting from 0) and returns `NULL` if `i` is greater or equal to `count_varargs()`.
`void* get_vararg(int i, Type* type)`	Composite	Returns the $\texttt{i}^{\texttt{th}}$ variadic argument (starting from 0) as the specified `type`. Returns `NULL` if the object cannot be treated as of the specified type or if `i` is greater or equal to `count_varargs()`.

Table 1: Functions and macros of the introspection interface