The Interface Definition Language for Fail-Safe C

Kohei Suenaga[†] Yutaka Oiwa[†] Eijiro Sumii[‡] Akinori Yonezawa[†]
{kohei, oiwa, sumii, yonezawa}@yl.is.s.u-tokyo.ac.jp

[†]Department of Computer Science, University of Tokyo
[‡]Department of Computer Science, University of Pennsylvania

Abstract. Fail-Safe C is a safe implementation of full ANSI-C being developed by Oiwa and Sekiguchi. It uses its own internal data representations such as 2-word pointers and memory blocks with headers describing their contents. Because of this, calls to external functions compiled by conventional compilers require conversion of data representations. Moreover, for safety, many of those functions need additional checks on their arguments and return values. This paper presents a method of semi-automatically generating a wrapper doing such work. Our approach is to develop an Interface Definition Language (IDL) to describe what the wrappers have to do before and after function calls. Our language is based on CamlIDL, which was developed for a similar purpose between Objective Caml and C. Our IDL processor generates code by using the types and *attributes* of functions. The attributes are additional information describing properties which cannot be expressed only by ordinary types, such as whether a pointer can be `NULL`, what range of memory can be safely accessed via a pointer, etc. We examined Linux system calls as test cases and designed a set of attributes necessary for generating their wrapper.

1 Introduction

The C language [9] is a programming language which was originally designed to develop UNIX. This language provides very flexible methods of memory access such as getting pointers to memory blocks, accessing memory via pointers, casting pointers into other types, and so on. Although these features enable programmers to write low-level operations efficiently and flexibly, they often cause critical security problems such as buffer overrun.

Fail-Safe C [13] is a full ANSI-C compatible compiler which makes programs written in C safe, preventing illegal memory accesses. Compared with previous studies which also make C programs safe [1, 3, 7, 12], Fail-Safe C can accept a larger set of programs. For example, Fail-Safe C can safely deal with casts from pointers to integers.

The previous safe C compilers can guarantee safety when the source files of a program are available and can be compiled with these compilers. However, they cannot make the program safe when some functions it calls (e.g., system calls)

are given as binaries compiled with usual C compilers such as GCC. Although some of the safe C compilers [3] provide helper functions to write wrappers for external functions, it is still tedious to write wrappers for all the external functions.

In this paper, we present a method of semi-automatically generating wrappers for external functions for Fail-Safe C. We designed an interface definition language (IDL) and implemented an IDL processor which generates wrappers from interface definitions of external functions. As long as the generated wrappers are used in calling external functions (and the functions behave correctly in accordance with the interface definitions), programs are guaranteed to run safely even when these functions are executed. There are many IDLs which interface safe languages with C [5, 6, 10]. However, they are designed primarily to convert data representation of one language to another, and do not take the safety of execution as important aspect. Our purpose is not only to interface Fail-Safe C with C, but to guarantee the safety of external function calls.

The wrappers' work is roughly categorized into two groups: checking preconditions and converting data representations. Because Fail-Safe C uses its original data representation, we have to convert representations of arguments and return values before and after calling external functions compiled by usual C compilers. Additionally, there are many functions which require some preconditions for safety before being called. Our IDL processor generates wrappers from given interface definitions which specify what preconditions have to hold, what kind of conversions are required, and so on. In execution time, the wrappers utilize the metadata that is added to arguments by the Fail-Safe C compiler to confirm preconditions like whether the passed memory block has sufficient size to call the function safely.

The rest of this paper is organized as follows: In Section 2, we briefly present how Fail-Safe C works. In Section 3, we examine by case study what wrappers have to do. After showing the syntax and semantics of our IDL in Section 4, we present the result of experiments in Section 5. Finally, we review previous work in Section 6 and discuss future work in Section 7.

2 Fail-Safe C

2.1 Data Representations

In this section, we briefly introduce the internal data representations used in Fail-Safe C. Further details are described in [13].

Memory Blocks. In the Fail-Safe C world, every memory block has a header. The header is a structure which contains size information of the memory block and a pointer to a structure called *TypeInfo*. Fail-Safe C performs boundary checks for every memory access by using the size information in the header. TypeInfo contains the name of the type of the memory block and a table of pointers to functions called *handler methods*.

Handler Methods. In the C language, one can arbitrarily cast a pointer into another type. This causes inconsistency between the static type of a pointer and the actual type of a memory block it points to. Thus, in general, we cannot trust a pointer's static type. Rather, we have to access a memory block according to its actual type.

For this purpose, Fail-Safe C augments each memory block with functions for accessing it, called handler methods. To be more specific, the TypeInfo structure of a memory block contains a table of pointers to its handler methods. Fail-Safe C replaces every memory access with calls to these handler methods. In accessing memory, handler methods perform safety checks such as alignment checks according to the memory block's layout. By using handler methods in each memory block, we can safely access a memory block even if a pointer's static type is different from the actual type of the memory block.

Fat Pointers. In Fail-Safe C, every pointer is represented in two words. The upper word represents the base address of the memory block it points to, while the lower word represents an offset from the base address. By separating the offset from the base address, Fail-Safe C can perform boundary checks fast. In addition, headers of memory blocks can be accessed fast and safely, because all headers are placed just before the base address. Moreover, Fail-Safe C can detect invalid arithmetics between pointers in different memory blocks by comparing their base addresses.

The least significant bit of a base address is used as a *cast flag*. This flag is on if the pointer was cast from a pointer of another type. In this case, handler methods must be used because the pointer's static type may be different from the memory block's actual type. On the other hand, if the flag is off, we can safely access the memory block in usual ways without using handler methods. By tracing whether a pointer is cast and using handler methods as necessary, Fail-Safe C achieves both accepting programs which are difficult to ensure safety because of cast pointers and accessing memory as fast as usual C can in many cases. Fig. 1 illustrates the relation among a fat pointer, a memory block and TypeInfo.

Fat Integer. In usual C, one can cast a pointer to an integer whose size is not smaller than the pointer's size. To allow this, Fail-Safe C represents integers in two words. A usual integer is represented in two words with the upper word set to zero. (Thus, programs which use an arbitrary integer as a pointer does not work in Fail-Safe C.) When a pointer is cast to an integer, Fail-Safe C writes the pointer's base address in the integer's upper word and the pointer's offset in the integer's lower word. When an integer is cast to a pointer, Fail-Safe C inspects the header pointed to by the upper word of the integer, and checks the type of the memory block. If the type is different from the pointer's type we are casting to, the cast flag of the pointer is set. With these procedures, we can maintain safety even when we cast a pointer to an integer and cast the integer back to another pointer. Note that only integers whose size is big enough to retain pointers are

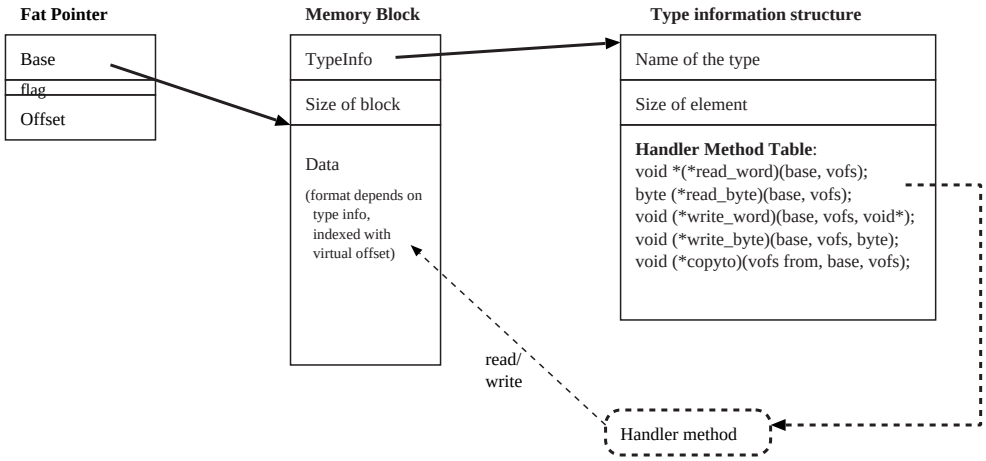

Fig. 1. Relation among a fat pointer, a memory block and a TypeInfo

represented in two words. Small integers like `char` are represented in the usual way.

2.2 What kind of safety does Fail-Safe C guarantee?

Most of unsafe behavior of C is caused by illegal memory accesses. Thus, the main focus of Fail-Safe C is on memory safety. However, not all of the unsafe behavior is caused by memory problems. For example, though integer overflow is not a memory problem, it is an unsafe behavior because it can lead to buffer overrun [2]. As discussed in [14, pp. 7–8], a programming language can be considered safe when programs written in the language can never go wrong – that is, at every point during execution, their next behavior is predictable from the semantics of the language. However, there are many "undefined" behavior in ANSI-C specification [9], including the cases of buffer overrun and even division by zero.

From these considerations, we state the safety of Fail-Safe C as follows:

> Assume that a semantics of C is defined. Fail-Safe C always aborts a program when its next behavior is undefined in the semantics.

Although we have to formally define the semantics of C in order to make this statement exact, we do not argue about such formalism here and use the above statement to informally understand the safety Fail-Safe C guarantees.

3 Wrappers

3.1 Control flow

Before explaining the details of wrappers, we briefly describe how wrappers generated from interface definitions work. Fig. 2 shows the flow of function calls. In this figure, a global variable `g` and three functions, `main`, `wrapper_f` and `f` are

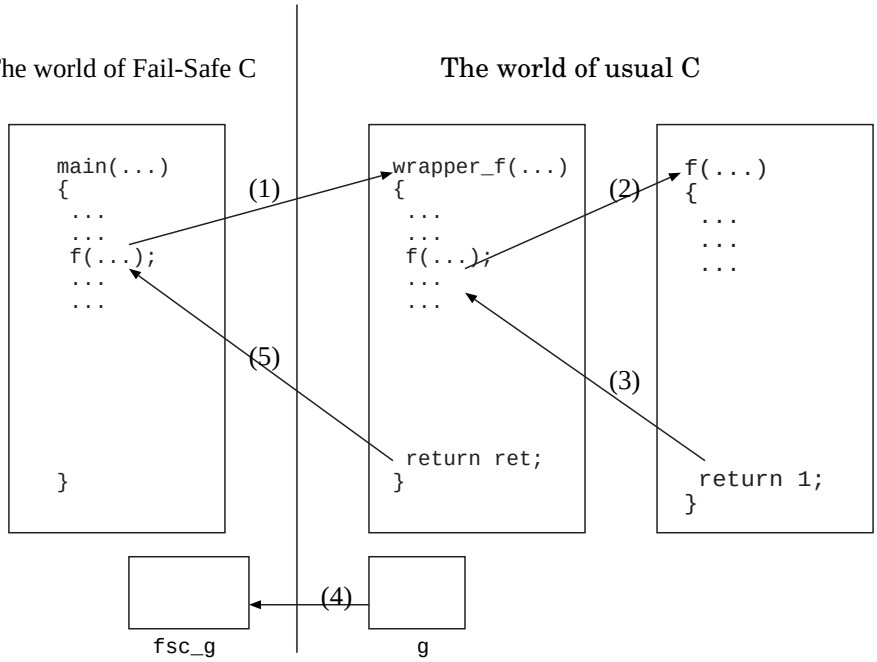

Fig. 2. Control flow of applications which use our IDL.

involved in an external function call. `main` is a function which is compiled by the Fail-Safe C compiler. `f` is an external function compiled by a usual C compiler. `wrapper_f` is a wrapper generated from the interface definition (written in our IDL) of `f`.

First, `main` calls the external function `f`. Fail-Safe C replaces this call by a call of `wrapper_f` [(1) in Fig. 2]. After `wrapper_f` confirming the preconditions specified by the interface definition of `f`, the representations of the original arguments of `f` are converted and passed to `wrapper_f`. Then, `wrapper_f` calls the external function `f` [(2)]. After `f` returns [(3)], `wrapper_f` converts the representation of the return value of `f` and pointer-type arguments passed to `wrapper_f` from the usual C's one to the Fail-Safe C's one. Besides, if the interface definition of `f` says that `f` updates global variables, say `g`, `wrapper_f` has to make

this update visible from `main`. To achieve this, our IDL processor prepares two regions for global variables. One is to retain a value of the global variable for Fail-Safe C's representation (`fsc_g` in Fig. 2), and the other is for the usual C's representation (`g` in Fig. 2). After `f` returns, `wrapper_f` copies the values of `g` to `fsc_g` with converting its representation [(4)].

3.2 What kind of safety does our IDL guarantee ?

As what safety Fail-Safe C guarantees was explained in 2.2, we will now explain the safety our IDL guarantees. The safety of our IDL is restricted in three ways. Firstly, our IDL guarantees only the safety Fail-Safe C guarantees. Thus, unsafe behavior that is not prevented by Fail-Safe C (e.g., fork bomb) is neither prevented by our IDL.

Secondly, our IDL assumes that Fail-Safe C itself does not have bugs. Thus, Fail-Safe C's safety always holds in Fail-Safe C's world especially just before calling external functions.

Lastly, we assume that each interface definition given to our IDL processor always agrees with the actual implementation of an external function. Thus, if an interface definition is wrong, safety is not guaranteed.

Then, in short, we can state the safety of our IDL as follows.

> A wrapper generated by our IDL safely calls an external function (in Fail-Safe C's sense) if the following conditions hold:
>
> – The safety of Fail-Safe C holds before calling external functions.
> – An interface definition agrees with the implementation of an external function.

3.3 Case Study: Linux System Calls

In this section, we describe the structure of wrappers. First, we explain by examples how wrappers should work. We use two linux system calls, `read` and `accept`, as examples.

`read` System Call. The Linux system call `read` is declared as follows:

```
int read(int fd, char *buf, int n);
```

`fd` is a file descriptor and `n` is the size of a memory block pointed to by `buf`. Data are written to the memory block `buf` points to if the return value is not `-1`. In this case, the return value indicates the size of written data. The return value `-1` means that data are not written due to some error. In this case, the global variable `errno` describes what the error is.

Preconditions. Firstly, we consider preconditions to call `read` safely. There are three preconditions that have to hold. First, because `n` is the size of a buffer, `n` $\geq$ 0 has to hold. Second, `buf` cannot be `NULL`. This can be confirmed by checking that the upper word (base address) of `buf` is not zero. Last, the size of the buffer has to be actually more than `n`. This can be confirmed from the size information in the header of the memory block. `read` does not perform write accesses to `buf` beyond the region specified by `n`. Therefore, as far as `read` is correctly implemented, buffer overrun does not occur when this function is called because the preconditions above are checked.

Converting data representations (before the call). After checking preconditions, a wrapper converts the data representations of arguments from Fail-Safe C's to usual C's. The wrapper converts `fd` and `n`, which have two-word representation, into the usual representation. As for `buf`, because the memory block `buf` points to has different layout, only converting the representation of the value of `buf` is insufficient. Thus, `buf` is converted to a pointer which points to a `n`-byte memory block newly allocated as in the usual C.[1]

Converting data representations (after the call). The wrapper converts the data representation of the return value, arguments and global variables from usual C's back to Fail-Safe C's. First, if the return value is not `-1`, the memory block `buf` points to has to be updated. To do this, the wrapper writes data from the memory block allocated before the function call (see above) back to the memory block pointed to by `buf`. Then the former memory block is deallocated.

If the return value is `-1`, a global variable `errno` is updated, indicating what error has occurred. In this case, the wrapper copies the value of the usual C's region to the Fail-Safe C's region converting its representation.

`accept` system call. Let us look into another example, the `accept` system call. It is declared as follows:

```
int accept(int socket,
           struct sockaddr *address,
           unsigned int *address_len);
```

where `sockaddr` is declared as follows:

```
struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
};
```

`socket`, `address` and `address_len` are, respectively, a socket descriptor, a pointer to `sockaddr` and a pointer to an integer which indicates the size of the structure `address` points to. `Address` may be `NULL`, in which case `address_len` is ignored.

[1] Actually, in this particular case, the conversion is not required since the representation of `char` is the same between Fail-Safe C and usual C.

A caller of `accept` passes `address` a pointer to `sockaddr`, which is cast from a pointer to a protocol specific struct. For example, if the caller uses the IP protocol, `address` is cast from a pointer to `sockaddr_in` defined as follows:

```
struct sockaddr_in {
    unsigned short sin_family;
    unsigned int sin_port;
    struct in_addr sin_addr; /* an IP address */
    unsigned char sin_zero[8];
};
```

It is guaranteed that such protocol specific structs contain an `unsigned short` member at the beginning. On the other hand, the static type of `sa_data`, an array of char, is usually different from its actual type.

If the return value of `accept` is not `-1`, the value is a descriptor of a newly created socket. In this case, the structure pointed to by `address` is updated, indicating the address with which the socket is connected. `address_len` is also updated, indicating the size of `address`. If the return value is `-1`, `errno` is updated to indicate what error has occurred.

Preconditions. We first see what preconditions are required to call this system call safely. The following four preconditions are to hold if `address` is not `NULL`:

- `address->sa_data` is not `NULL`.
- `address_len` is not `NULL`.
- `*address_len` $\geq$ `0`.
- The size of the memory block `address` points to is more than `*address_len` bytes.

Among the above conditions, the first and second can be confirmed by checking that the upper word of `address->sa_data` and `address_len` is not zero. The fourth condition is confirmed from the header of the memory block `address` points to.

Converting data representations (before the call). After confirming preconditions, we convert data representations of arguments as we did in the case of `read`. Firstly, the wrapper converts `socket`'s representation from two-word to one-word.

Secondly, if `address != NULL`, the wrapper has to allocate `*address_len` bytes of memory block and copy the contents of the memory block pointed to by `address`. During the copy, the representation of each member of the structure has to be converted. `Sa_family`, whose type is `short` and its representation is the same in Fail-Safe C and usual C, can be copied directly. Although `sa_data`'s actual type may be different from its static type as we mentioned before, we can copy data safely using handler methods.

Finally, if `address_len` is not `NULL`, the wrapper newly allocates an integer in the usual C's representation, copies `*address_len` to the integer converting its representation, and makes a pointer to the integer.

Converting data representations (after the call). After the function returns, the wrapper again has to convert back the representation of the return value and reflects side effects as in the case of `read`.

The return value and the global variable `errno` are converted completely in the same way as in `read`.

If the return value is not `-1`, the memory blocks pointed to by `address` and `address_len` may have been overwritten. Conversion of these values is done as follows. For `address_len`, the wrapper copies the value of the integer newly allocated before the call (see above) back to the memory block `address_len` points to. For `address`, the wrapper copies each member's value converting their representation. During the copy, the wrapper uses handler methods for `sa_data` as we did before calling.

3.4 Information required by IDL

From the case study above, we now see that our IDL needs at least the following information to generate wrappers.

- Whether a pointer can be `NULL`.
- What region of a memory block can be accessed safely via a pointer.
- What kind of side effects, such as updates of memory blocks and global variables, occur during a function call.
- Conditions which have to hold before a function call (for example, `n` $\geq$ `0` in `read`).

3.5 Structure of wrappers

On the basis of the case study in the previous sections, we split wrappers in the following five phases:

- Precondition Checking
- Decoding and Allocation
- Call
- Deallocation and Encoding
- Return

In the rest of this section, we explain what wrappers do in each phase.

Precondition Checking. In this phase, wrappers confirm that preconditions specified in interface definitions actually hold. For example, preconditions like whether a pointer is `NULL` and whether a specified region is safely accessible are confirmed, along with user-specified preconditions such as `n` $\geq$ `0`.

Decoding and Allocation. In this phase, wrappers convert the representations of arguments from Fail-Safe C's to usual C's. Integers, which are represented in two words—base and offset—are converted to one-word representation by adding the offset to the base. As for pointers, wrappers allocate memory blocks with the size which is specified by the pointer's attribute. Then, wrappers copy the contents of original memory blocks to the allocated memory blocks, converting the data representation. The result of the conversion is a pointer to the newly allocated memory block.

Call. After the two phases above, a wrapper calls an external function. If the previous two phases have finished normally, and if the implementation of the external function matches the interface definition, the call should be safe. (Even if there are postconditions that have to hold, wrappers do not check those. As mentioned in section 3.2, our IDL assumes that an interface definition matches the implementation of the external function. Under this assumption, if the preconditions hold, the postconditions should also hold.)

Deallocation and encoding. In this fourth phase, a wrapper (1) converts the representation of the return value and (2) reflects side effects caused by an external function call. The former work is exactly the reverse of the decoding and allocation phase. The latter work is as follows. Our IDL processor prepares memory that keeps Fail-Safe C's representation of each global variable. If an external function may have updated some global variables, the wrapper copies their values to the memory, converting their representation.[2]

Return. If there is a return value, the wrapper returns a value converted in the previous phase.

4 Our IDL

4.1 Syntax

In this section, we present the syntax of our IDL. Interface definitions written in our IDL almost look like C's header files which consist of declarations of functions, global variables and so on. One difference between our IDL and C's declaration is that types are annotated with *attributes* in our IDL. Attributes are additional information which cannot be expressed only with C's types, such as what region is accessible via a pointer.

Table 1 shows (part of) the syntax of our IDL.

[2] Conversely, in the decoding and allocation phase, we could copy the value of global variables from Fail-Safe C's memory into usual C's. However, this is not implemented so far, because we have never come across an external function which requires that.

attributes	::= ϵ | `[` { attribute, } `]`
attribute	::= ident
	| ident `(` { pure-expr, } `)`
	| `*` attribute
global-decl	::= attributes type declarator
struct-decl	::= `struct {`
	{ attributes type declarator }
	`}`
declarator	::= { `*`[`const`] } direct-declarator
direct-declarator	::= ident
	| `(` declarator `)`
	| direct-declarator `[` [pure-expr] `]`
	| direct-declarator `(` param-list `)`
param-list	::= { attributes type declarator, }
function-decl	::= attributes$_1$ type { `*` [`const`] } ident `(` param-list `)` attributes$_2$

Table 1. Syntax of our IDL (excerpt): { T } means a sequence of zero or more Ts. { T, } means a comma-separated sequence of zero or more Ts.

The metavariable "type" is a type of C. "Ident" is a string which can be used as an identifier in C. "Pure-expr" is a C's expression which does not contain side effects. An identifier `_ret` is reserved to refer to a return value in "pure-expr".

"Attributes" is information added to types which is needed to generate a wrapper. An attribute with `*` is given to a value of pointer type. For example, if an attribute `A*` is given to a value of type `T*`, it means that the attribute `A` is given to the type `T`.

"Global-decl" is a declaration of a global variable which may be updated by external functions. It is a global variable declaration of C accompanied with attributes. "Struct-decl" is a declaration of a structure which is used by interface definitions. It consists of a structure declaration of C and attributes of each member. "Function-decl" is a interface definition of an external function. "attributes$_1$" is attributes of the return value and "attributes$_2$" is attributes of the function.

4.2 Attributes and their semantics

In this section, we present the attributes in our IDL, designed based on the case study described in Section 3.3.

`always_null, maybe_null, never_null`. These attributes are given to a pointer type variable, meaning the pointer has to be always `NULL`, can be `NULL` and must not be `NULL`, respectively. For example, for the `read` system call in Section 3.3, `never_null` is given to `buf`.

If these attributes are given, the generated wrapper confirms if these conditions actually hold in Precondition Checking phase.

can_access_in_elem(e_1, e_2). This attribute is given to a variable of pointer type. It means that, if this attribute is given to a pointer argument p, the region from $p + e_1$ to $p + e_2$ must be readable and writable via p, where e_1 and e_2 are pure-exprs. In the case of `read` in Section 3.3, `can_access_in_elem(0, n - 1)` is given to `buf`.

If this attribute is given, the wrapper confirms that an indicated region is actually accessible from the header of the memory block in Precondition Checking phase. In Decoding and Allocation phase, the wrapper copies the contents of the given memory block to newly allocated one. In copying, the wrapper converts the representations of the contents of the memory block from the Fail-Safe C's one to the usual C's one.

can_access_in_byte(e). This attribute is given to a pointer type variable, meaning that e bytes from the pointer is accessible via the pointer. In the case of `accept` in Section 3.3, `can_access_in_byte(*address_len)` is given to `address`[3].

If this attribute is given, the wrapper confirms the indicated region is actually accessible from the header of the memory block pointed to by the pointer. The wrapper also copies the contents of the memory block to a newly allocated memory block in Decoding and Allocation phase.

string. This attribute is given to a pointer to `char`, meaning that the variable is a pointer to a null-terminated string. If this attribute is given, the wrapper confirms that the pointer is actually null-terminated in Precondition Checking phase and copies the string to a newly allocated memory block in Decoding and Allocation phase.

write_global(e_1, ident). This attribute is given to function declarations (specified at attributes$_2$ of Table 1.). `Ident` is a name of a global variable. This attribute means that if $e_1 \neq 0$ holds after the external function returns, `ident` is updated by the function. For example, `write_global(_ret == -1, errno)` is given to `read`. If this attribute is given, the wrapper copies the value of specified variable of the usual C's world to Fail-Safe C's one converting representation.

write(e, ident, e_1, e_2). This attribute is given to a function declaration. `Ident` is a pointer-type argument of the function which is not qualified by `const`. It means that, if $e \neq 0$ holds after the function returns, the region from `ident` $+e_1$ to `ident` $+e_2$ may be updated by the function. For example, `write(_ret != -1, buf, 0, _ret - 1)` is given to `read`. If this attribute is given, the wrapper writes back the contents of the specified region in Deallocation and Encoding phase. e_1 and e_2 can be omitted. In this case the wrapper writes back all of the accessible regions specified by other attributes.

[3] Because `address_len` is dereferenced at this point, the precondtions of `address_len` have to be confirmed before those of `address` are confirmed. There may be dependencies among arguments as in this case. For now, our IDL ignores these dependencies and users have to resolve them manually.

precond(e). This attribute is given to a function declaration. It means that $e \neq 0$ has to hold before the external function call. For example, `precond(n >= 0)` is given to `read`. The specified condition is checked in Precondition Checking phase.

4.3 An example of interface definition

Fig 3 shows an example of interface definition. We gave the interface definition of `open`, `read`, `write` and `accept`.

```
int errno;

struct sockaddr {
    unsigned short sa_family;
    [never_null] char *sa_data;
};

int open([never_null, string] const char *path, int flags)
  [write_global(_ret == -1, errno)];
int read(int fildes,
                 [never_null,
          can_access_in_elem(0, nbytes - 1),
          write(_ret != -1, 0, _ret - 1)] char *buf,
                 int nbytes)
  [precond(nbytes >= 0), write_global(_ret == -1, errno)];
int write(int fildes,
                  [never_null, can_access_in_elem(0, nbytes - 1)]
             const char *buf,
                  int nbytes)
  [precond(nbytes >= 0), write_global(_ret == -1, errno)];
int accept(int socket,
           [can_access_in_byte(*address_len),
            write(_ret != -1)] struct sockaddr *address,
           [write(_ret != -1)] int *address_len)
  [precond(address == NULL || address_len != NULL),
   write_global(_ret == -1, errno)];
```

Fig. 3. An example of interface definition

Note that each attribute can refer to every argument of the interface definition the attribute belongs to even if the attribute occurs before the argument it refers to. For example, although `buf` occurs before `nbytes` in `read`'s interface definition, `nbytes` can be referred to by `can_access_in_elem` which is given to `buf`.

5 Experiments

5.1 Method and results

We implemented our IDL processor described above in Objective Caml. Using this implementation, we measured the overhead caused by wrappers, comparing the time spent by programs which calls external functions in the following two cases:

- Programs compiled by Fail-Safe C and linked with Fail-Safe C run-time libraries. Of course it uses generated wrappers to call external functions.
- Programs compiled by gcc `-O3` which calls external functions directly.

We conducted experiments on machines with Sun UltraSPARC-II 400 MHz CPU with 13.0GB main memory.

Overhead of converting integers. First, we wrote a function `succ` as an external function. It receives one integer, adds 1 to it and returns. We also wrote a program which calls `succ` 10^7 times sequentially and compared the duration spent in the two cases mentioned before. The result of this experiment shows the overhead of converting integers. We show the result in Table 2. The overhead

	`succ`	`arraysucc`	`cp`
with wrapper (msec)	234	597	144
without wrapper (msec)	220	200	91
overhead (%)	6	199	58

Table 2. Overhead of each program

of the wrapper is only 6%.

Overhead of converting pointer-type arguments. Next, we wrote a function `arraysucc` and compiled it with gcc. It receives an array of 10^7 `char` and adds 1 to each element. We also wrote a program which calls `arraysucc` as an external function. The result of this experiment shows the overhead of converting pointer-type arguments. The result in Table 2 shows that the overhead of allocating memory and copying contents is very large, 199%, comparing with the overhead of converting integers.

In this experiment, we also measured how much time is spent in each phase. We show the result in Table 3.

Overhead of a more practical program. Last, we wrote a program which copies files of 10^6 bytes using `open`, `read`, `write` as external functions, and measured overhead. The result is shown in Table 2. The overhead is 58% with wrappers.

	P	D	C	E	Total
Execution time (msec)	0	326	110	160	596
Ratio (%)	0	54.7	18.5	26.8	-

Table 3. Overhead of converting pointer-type arguments. P: Precondition Checking, D: Decoding and Allocation, C: Call, E: Deallocation and Encoding

	P	D	C	E	Total
Execution time of `read`'s wrapper (msec)	1	16	46	14	77
Execution time of `write`'s wrapper (msec)	1	16	73	4	94

Table 4. Time spent in each phase of `read`'s and `write`'s wrapper. P: Precondition Checking, D: Decoding and Allocation C: Call, E: Deallocation and Encoding

Also in this experiment, we measured the time spent in each phase. We present the result of `read`'s and `write`'s wrapper in Table 4. In this program, The time spent for Call phase is dominant because file access is performed in this phase. However, focusing on the execution time of the wrapper itself, the overhead of Decoding and Allocation phase is large as in the case of `arraysucc`.

5.2 Discussion

From the experiments above, we see that most of overhead is caused by Decoding and Allocation phase. Thus, to reduce overhead of a whole wrapper, we need to reduce the overhead of this phase.

There are two possible solutions. First one is to omit copying in Decoding and Allocation phase. For example, in the case of `read`, the contents of the memory block pointed to by `buf` is never read and always overwritten. In such case, wrappers do not have to copy contents of memory block in Decoding and Allocation phase.

Second one is to omit allocation of memory block in Decoding and Allocation phase if representations of memory blocks are the same in usual C and Fail-Safe C. For example, again in the case of `read`, if `buf` points to a memory block whose actual type is `char`, all the wrapper has to do is to convert a two-word representation to one-word one, without allocating a new memory block.

6 Related Work

Many functional languages have its own IDLs. For example, H/Direct [5, 6] and CamlIDL [10] are IDLs to call external functions from functional languages. The former is an IDL for Haskell [8] and the latter is for Objective Caml [11]. Although Chez Scheme [4] has no independent IDL, it can call external functions using `foreign-procedure` function. This function receives an interface definition of an external function and returns closure that calls the external function. These

IDLs focus on conversion of data representation and pay less attention to safety checks than our IDL does. For example, although CamlIDL prepares `size_is` attribute which is a counterpart of our `can_access_in_elem`, wrappers generated by CamlIDL does not check buffer size even if the attribute specified. Because of this, if a buffer of insufficient size is passed to a generated wrapper, safety of an external function call is lost.

CCured [3, 12] is another safe implementation of C. This implementation analyses pointer usage statically and reduces needless dynamic checks. Because CCured also uses its own data representation, it also requires the data conversions before calling external functions. There are two methods to call external functions from programs compiled with CCured. The first one is to manually write wrappers for external functions. These wrappers check preconditions and converts data representation as ones which our IDL generates. The second one is to add the annotations that identify external function calls. With these annotations, CCured infers which data is used by the external functions. CCured separates the metadata of such data from the actual data. (The metadata is usually held in one memory block together with the actual data.) Because the layout of the actual data is the same as the usual C's, CCured can pass these data to external functions without converting representation. The second method, however, has a problem; it cannot be applied to external functions that have side effects. Although CCured may have to update the metadata after calling such functions, there is no way for programmers to express what updates have to be reflected. With our IDL, one can cope with external functions with side effects.

7 Conclusion and Future Work

In this paper, we proposed a method to semi-automatically generate wrappers to call external functions from Fail-Safe C. Using our implementation, we also measured overhead of generated wrappers.

For future work, we are planning to apply our approach to larger and more practical programs. Because the implementation of Fail-Safe C has not been completed yet, the amount of overhead for practical programs are yet to be seen. We are planning to examine all system calls and library functions, extract common features and identify a reasonable set of attributes. We will also examine the effectiveness of the optimizations mentioned in Section 5.2 by applying it to many functions.

Acknowledgment

We are grateful to the members of Yonezawa group, especially to Yoshihiro Oyama, for their comments on our work. Many ideas in this work stemmed from discussion with them.

References

1. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 290–301, 1994.
2. CERT/CC. CERT Advisory CA-2003-10 Integer overflow in Sun RPC XDR library routines, April 2003. http://www.cert.org/advisories/CA-2003-10.html.
3. Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, June 2003.
4. R. Kent Dybvig. *Chez Scheme User's Guide*, 1998. http://www.scheme.com/csug/index.html.
5. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pp. 153–162, 1998.
6. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pp. 114–125, 1999.
7. T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Technical Conference*, June 2002.
8. Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*, December 2002. http://www.haskell.org/definition/haskell98-report.pdf.
9. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language (Second Edition)*. Prentice Hall, 1988.
10. Xavier Leroy. *CamlIDL Users Manual*. INRIA Recquencourt, July 2001. http://camlidl.inria.fr/camlidl.
11. Xavier Leroy, Dmien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.06 Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, August 2002.
12. George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN–SIGACT symposium on Principles of Programming Languages*, pp. 128–139, 2002.
13. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An approach to making C programs secure (progress report). In *Proceedings of International Symposium on Software Security, Tokyo, Japan, November 8–10, 2002*, Vol. 2609 of *Lecture Notes in Computer Science*. Springer-Verlag, February 2003.
14. Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.