Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure

Progress Report

Yutaka Oiwa[1], Tatsurou Sekiguchi[1,2], Eijiro Sumii[1], and Akinori Yonezawa[1]

[1] University of Tokyo, 7–3–1 Hongo, Bunkyo-ku, Tokyo 113-0033 JAPAN
[2] PRESTO, Japan Science and Technology Corporation

Abstract. It is well known that programs written in C are apt to suffer from nasty errors due to dangling pointers and/or buffer overflow. In particular, such errors in Internet servers are often exploited by malicious attackers to "crack" an entire system, which becomes even social problems nowadays. Nevertheless, it is yet unrealistic to throw away the C language at once because of legacy programs and legacy *programmers*. To alleviate this dilemma, many approaches to safe implementations of the C language—such as Safe C and CCured—have been proposed and implemented. To our knowledge, however, none of them support all the features of the ANSI C standard *and* prevent all unsafe operations. (By unsafe operations, we mean any operation that leads to "undefined behavior", such as array boundary overrun and dereference of a pointer in a wrong type.)

This paper describes a memory-safe implementation of the *full* ANSI C language. Our implementation detects and disallows all unsafe operations, yet conforming to the full ANSI C standard (including casts and unions) and even supporting many "dirty tricks" common in programs beyond ANSI C. This is achieved using sophisticated representations of pointers (and integers) that contain dynamic type and size information. We also devise several techniques—both compile-time and runtime—to reduce the overhead of runtime checks.

1 Introduction

The C language, which was originally designed for writing early Unix systems, allows a programmer to code flexible memory operations for high runtime performance. Especially, the C language provides flexible pointer arithmetic and type casting of pointers, which can be used for direct access to raw memory. Thus it can be used as an easy-to-use replacement for assembly languages to write many low-level system programs such as operating systems, device drivers and runtime systems of programming languages.

Today, the C language is still one of the major languages for writing application programs including various Internet servers. As requirements for applications become complex, programs written in the C language have come to perform complex pointer manipulations very frequently, which tends to cause serious security bugs. In particular, destroying on-memory data structures by array buffer overflows or dangling pointers makes the behavior of a running program completely different from its text. In addition, by forging specially formed input data, malicious attackers can sometimes take over the behavior of such buggy programs. Most of recently reported security holes are due to such misbehavior.

To resolve the current situation, we propose a method for safe execution which can accept all programs written in conformity with the ANSI C specification. The notion of safety can be stated basically in terms of "well-definedness" in ANSI C. The phrase *behavior undefined* in the specification implies that any behavior including memory corruption or program crashes conforms to the specification. In other words, all unsafe behavior of existing programs corresponds to the undefined behavior in the specification. So, if the runtime system detects all operations whose behavior is undefined, we can achieve safe execution of programs. Our compiler inserts check code into the program to prevent operations which destroy memory structures or execution states. If a buggy program attempts to access a data structure in a way which leads to memory corruption, the inserted code reports the error and then terminates the execution. By using our compiler system instead of usual C compilers, existing C programs can be executed safely without change.

2 Basic Approach

This section explains our basic approach to preventing memory failure, using a simple model.

In a safe programming language with dynamic typing such as Scheme and Lisp, memory safety is usually assured by a tag assigned to each memory block as well as by runtime type checking. This idea can also be applied to the C language. (Actually, some interpreter implementations of the C language take this approach.) Each contiguous memory region has a tag which keeps type information about that region. Accordingly, a source program is transformed into an equivalent program with runtime checking inserted. (Currently, the target language is also C.) Every memory access is in principle type-checked at runtime, although most of such runtime checking is actually omitted without losing safety, thanks to our representations of integers and pointers described later.

2.1 Integer and Pointer Representations

We use a kind of smart pointers called *fat pointers*. A pointer is represented by a triple and every pointer operation in a source program is translated into operations on the triple. The representations of integers and pointers in our execution model can be described as follows:

$$v ::= \mathrm{num}(n) \mid \mathrm{ptr}(b, o, f)$$

An integer is denoted by $\mathrm{num}(n)$ where n is the value of the integer, while a pointer is denoted by $\mathrm{ptr}(b, o, f)$ where b is the base address of some contiguous memory region, o is an offset in the memory region, and f is a *cast flag*. The `NULL` value is represented by $\mathrm{num}(0)$. When the cast flag of a pointer is set, it indicates that the pointer *may* refer to a value of a type different from its static type.

$e ::=$	(typed expression)
$v : t$	(constant)
$x : t$	(variable)
$*\, e : t$	(dereference)
$*\, e = e : t$	(update)
$(t)e : t$	(cast)
$e + e : t$	(addition)
$\mathrm{new}\langle t\rangle(e) : t$	(new)
$\mathrm{let}\ x : t = e\ \mathrm{in}\ e : t$	(let binding)

Fig. 1. Typed expressions

2.2 Expressions and Heaps

A simple language with a type system and an operational semantics is introduced to illustrate how pointers and integers are handled in our implementation. Expressions of the language are defined in Fig. 1. *Every* expression in the language is typed. This reflects the fact that the semantics of C is defined according to the type of an expression. For example, when a pointer `p` points to some element in an array, `p + 1` have the address of the next element, while `(int)p + 1` evaluates to the address of the next byte. A type is either the integer type or a pointer type:

$$t ::= \mathrm{int} \mid t\ \mathrm{pointer}$$

A heap H is a partial mapping from base addresses to finite vectors of values. Pointer $\mathrm{ptr}(b, o, f)$ refers to the o-th element of a vector associated with b in a heap. Each element of a heap has its type, which is defined by a heap typing HT. A heap typing is a partial mapping from base addresses to vectors of types.

We always force heaps and heap typings to keep the invariant condition that each element in a heap is either:

1. $\mathrm{int}(v)$,
2. $\mathrm{ptr}(b, o, 0)$ of a correct type (i.e., the heap type of the region containing the value is $HT(b)$ pointer), or
3. $\mathrm{ptr}(b, o, 1)$ of any type.

The runtime system manipulates the cast flags so that the type of a pointer whose cast flag is off is always trustworthy, that is, the static type of such a pointer is correct. It only checks the type of the pointers whose cast flags are set.

We do not have a condition that an offset of a pointer is within a valid region since the ANSI C Standard allows a pointer which exceeds its limit by one when the control exits out of a for- or while-statement. Such an invalid pointer *per se* does not bring about undefined behavior unless it is dereferenced. We therefore postpone the boundary check until the pointer is dereferenced.

2.3 Pointer Operations

The relation $H, HT;\ e : t \rightsquigarrow H', HT';\ e' : t$ means that the expression $e : t$ reduces to $e' : t$ in one step under the heap H and the heap typing HT, and the heap and the heap typing are updated to H' and HT' respectively. In addition, $H, HT;\ e : t \rightsquigarrow \mathrm{fail}$ means that the evaluation of $e : t$ causes handled error under H and HT. The part "$H, HT;$ " may be omitted if the operations do not modify or depend on heap states.

Cast. The cast operation is defined as follows:

$$H, HT;\ (t)\,(\mathrm{num}(n)) : t \rightsquigarrow H, HT;\ \mathrm{num}(n) : t$$

$$\frac{HT(b) = t}{H, HT;\ (t\ \mathrm{pointer})\,(\mathrm{ptr}(b, o, f)) : t\ \mathrm{pointer} \rightsquigarrow H, HT;\ \mathrm{ptr}(b, o, 0) : t\ \mathrm{pointer}}$$

$$\frac{HT(b) \neq t}{H, HT;\ (t\ \mathrm{pointer})\,(\mathrm{ptr}(b, o, f)) : t\ \mathrm{pointer} \rightsquigarrow H, HT;\ \mathrm{ptr}(b, o, 1) : t\ \mathrm{pointer}}$$

$$H, HT;\ (\mathit{int})\,(\mathrm{ptr}(b, o, f)) : \mathit{int} \rightsquigarrow H, HT;\ \mathrm{ptr}(b, o, 1) : \mathit{int}$$

In conventional compilers, cast operations over pointers are usually an identity function. In Fail-Safe C, however, cast operations become more complex because our invariant conditions on pointer values refer to the static types of pointers. A cast operator checks the dynamic type of the region pointed to by a pointer, and if the pointer refers to a value of a wrong type, the cast flag of the pointer is set.[1]

Note that when a pointer value is cast to an integer, its result is still of the form $\mathrm{ptr}(b, o, 1)$, called a *fat integer*, so that it can be cast back to a pointer. This is due to an ANSI C requirement: if an integer variable has enough width (i.e. if `sizeof(int)` $\geq$ `sizeof(void*)`), then the variable must be able to hold pointer values which may be cast back to pointer types later.

[1] Alternatively, we can set the cast flag without runtime type checking. In this case, type check is performed on each memory access. We think this algorithm will have worse performance than that shown in the main text, because cast pointers are often used more than once, but rarely discarded without being used.

Dereference. A pointer in C may refer to a location outside of a valid region. So our compiler always inserts boundary checking code unless an optimizer recognizes the pointer to be inside a valid region [2,15,16]. It is impossible to make a pointer referring to a memory region from a pointer referring to another memory region in the ANSI-C. Consequently a pointer whose offset is outside of its referred region is always invalid. That is, when two pointers which refer to different memory regions are subtracted, the result is defined to be "undefined".

$$H, HT;\ *(\mathrm{num}(n)) : t \rightsquigarrow \mathrm{fail}$$

$$\frac{o \text{ outside } H(b)}{H, HT;\ *(\mathrm{ptr}(b, o, f)) : t \rightsquigarrow \mathrm{fail}}$$

If the cast flag of a pointer is unset (i.e., $f = 0$), the value read via the pointer is guaranteed to have the correct type by the invariant stated above. Therefore, no type checking is needed when such a pointer is dereferenced.

$$\frac{o \text{ inside } H(b) \qquad \mathrm{HT(b) = t}}{H, HT;\ *(\mathrm{ptr}(b, o, 0)) : t \rightsquigarrow H, HT;\ v : t} \quad \text{where } v \text{ is the } o\text{-th element of } H(b)$$

On the other hand, a value read using a pointer with $f = 1$ may have an incorrect type. Therefore, the type of the value to be read is checked. The cast operator (t) shown in the rule below represents this runtime type check.

$$\frac{o \text{ inside } H(b)}{H, HT;\ *(\mathrm{ptr}(b, o, 1)) : t \rightsquigarrow H, HT;\ (t)v : t} \quad \text{where } v \text{ is the } o\text{-th element of } H(b)$$

Update. Memory writing operation $*e : t = v$ is defined as follows: If the pointer operand is actually an integer, the program execution is terminated.

$$H, HT;\ *(\mathrm{num}(n)) = v \rightsquigarrow \mathrm{fail}$$

When the cast flag of the pointer is not set, the runtime system checks whether the offset is inside the boundary. If the check succeeds, the value of the second operand is stored.

$$\frac{o \text{ inside } H(B)}{H, HT;\ *(\mathrm{ptr}(b, o, 0)) = v \rightsquigarrow H[(b, o) := v], HT;\ v}$$

$$\frac{o \text{ outside } H(B)}{H, HT;\ *(\mathrm{ptr}(b, o, 0)) = v \rightsquigarrow \mathrm{fail}}$$

However, if the cast flag of the pointer operand is set, the second operand is converted as if it is cast into the element type of the first operand.

$$\frac{o \text{ outside } H(B)}{H, HT;\ *(\text{ptr}(b,o,1)) = v \rightsquigarrow \text{fail}}$$
$$\frac{o \text{ inside } H(B)}{H, HT;\ *(\text{ptr}(b,o,1)) = \text{num}(n) \rightsquigarrow H[(b,o) := \text{num}(n)], HT;\ v}$$
$$\frac{o \text{ inside } H(B) \qquad HT(b') \text{ pointer} = HT(b)}{H, HT;\ *(\text{ptr}(b,o,1)) = \text{ptr}(b',o',f') \rightsquigarrow H[(b,o) := \text{ptr}(b',o',0)], HT;\ v}$$
$$\frac{o \text{ inside } H(B) \qquad HT(b') \text{ pointer} \neq HT(b)}{H, HT;\ *(\text{ptr}(b,o,1)) = \text{ptr}(b',o',f') \rightsquigarrow H[(b,o) := \text{ptr}(b',o',1)], HT;\ v}$$

2.4 Integer and Pointer Arithmetic

Addition is defined as follows:

Adding two integers:

$$\begin{aligned}
\text{num}(n_1) : \text{int} + \text{num}(n_2) : \text{int} &\rightsquigarrow \text{num}(n_1 + n_2) : \text{int}\\
\text{ptr}(b_1, o_1, f_1) : \text{int} + \text{num}(n_2) : \text{int} &\rightsquigarrow \text{num}(b_1 + o_1 + n_2) : \text{int}\\
\text{num}(n_1) : \text{int} + \text{ptr}(b_2, o_2, f_2) : \text{int} &\rightsquigarrow \text{num}(n_1 + b_2 + o_2) : \text{int}\\
\text{ptr}(b_1, o_1, f_1) : \text{int} + \text{ptr}(b_2, o_2, f_2) : \text{int} &\rightsquigarrow \text{num}(b_1 + o_1 + b_2 + o_2) : \text{int}
\end{aligned}$$

When two integers are added, the arguments must be treated as numbers. We define the *integer interpretation* of the value $\text{ptr}(b, o, f)$ to be $(b + o)$, which equals the usual representation of pointer values in conventional compilers.[2] Both arguments are converted to their integer interpretations and then added. The result will not be a pointer.[3]

[2] We take care of this equivalence, because this is the easiest way to retain the property that any different pointers have the different integer values, which many programs rely on (although this is not guaranteed by ANSI-C). See Section 6.2 for some discussion.

[3] Alternatively, we can keep the base value of one operand when the other operand is statically known to be non-pointer (e.g., a constant value). By this trick our compiler can support some non-ANSI-C hacks, e.g., `(int *)((int)p & ~3)` to get aligned pointers. In this paper, however, we does not take this into account because we cannot define the semantic consistently, e.g., it is unclear which of two base values should be kept when two pointer values are added. We will consider such tricks in our future version of compiler as a kind of “dirty” compatibility features.

Adding a pointer and an integer:

$$\begin{gathered}
\mathrm{ptr}(b_1, o_1, f_1) : t \text{ pointer} \mathbin{\textbf{+}} \mathrm{num}(n_2) : \text{int} \rightsquigarrow \mathrm{ptr}(b_1, o_1 + n_2, f_1) : t \text{ pointer} \\
\mathrm{ptr}(b_1, o_1, f_1) : t \text{ pointer} \mathbin{\textbf{+}} \mathrm{ptr}(b_2, o_2, f_2) : \text{int} \rightsquigarrow \mathrm{ptr}(b_1, o_1 + b_2 + o_2, f_1) : t \text{ pointer} \\
\mathrm{num}(n_1) : t \text{ pointer} \mathbin{\textbf{+}} \mathrm{num}(n_2) : \text{int} \rightsquigarrow \mathrm{num}(n_1 + n_2) : t \text{ pointer} \\
\mathrm{num}(n_1) : t \text{ pointer} \mathbin{\textbf{+}} \mathrm{ptr}(b_2, o_2, f_2) : \text{int} \rightsquigarrow \mathrm{num}(n_1 + b_2 + o_2) : t \text{ pointer}
\end{gathered}$$

When an integer is added to a pointer, the base (b_1) and the flag (f_1) of the pointer operand are kept in the result. A pointer of the integer type loses its pointer information. All the rules of the operational semantics are listed in the Appendix.

3 Implementation

3.1 Encoded Pointers and Integers

As described in the previous section, pointers and integers in Fail-Safe C contain more information than those in usual implementations. Both pointers and integers are represented by two machine words. We use one word for the base (b) and another for the offset (o), and borrow the least significant bit from the base word for the cast flag (f), assuming that base values are at least 2-byte aligned (i.e., its least significant bit is always zero). Non-pointer values (num) are stored in offset fields and the base fields are kept to be zero which means null (invalid) base.

If the base of a pointer is non-zero, the value of the base must point to the top of a valid memory block. In addition, if the cast flag of a pointer is unset, it indicates that 1) the pointer refers to the region of a correct type, and that 2) the offset of the pointer is correctly aligned according to the static type of the pointer.

Figure 2 illustrates those representations. In each case, the value (base + offset) in a fat pointer is equal to its integer interpretation. Those representations are first proposed in our earlier papers [12,13]. In our present scheme, however, we refine them by requiring the pointer whose cast flag is off to be aligned so that alignment checks can also be eliminated.

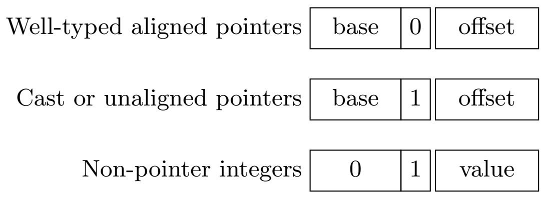
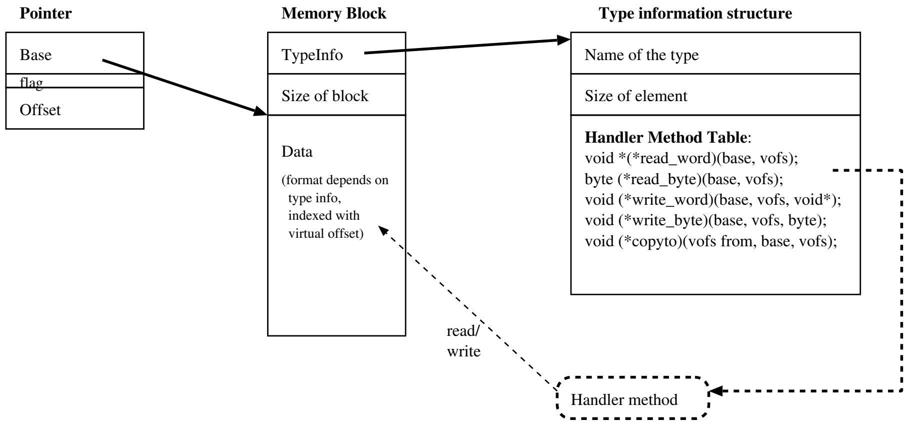


Fig. 2. Representations of fat pointers


Pointer
Base
flag
Offset
Memory Block
TypeInfo
Size of block
Data
(format depends on type info, indexed with virtual offset)
read/ write
Type information structure
Name of the type
Size of element
Handler Method Table:
void *(*read_word)(base, vofs);
byte (*read_byte)(base, vofs);
void (*write_word)(base, vofs, void*);
void (*write_byte)(base, vofs, byte);
void (*copyto)(vofs from, base, vofs);
Handler method

Size of block: Byte count of the data block (= max. virtual offset + 1)

Fig. 3. Structure of memory header and type information

The optimizer may reduce a fat integer to a simple, conventional representation unless doing so changes the semantics of a program. For example, the results of arithmetic operations are always non-pointers, and if a value is used only as an operand to integer arithmetics, it does not need to be a fat integer. In addition, the cast flag can often be proved to be a constant and thus can be omitted. For example, the flags of fat integers are always set and thus can be omitted, because there is no object which may be pointed to by integers without casting. Those properties can be used for optimization to reduce the runtime overhead of our implementation scheme.

3.2 Memory Blocks

Every memory access operation in Fail-Safe C must ensure that the offset and the type of a pointer are valid. To check this property at runtime, the system must know the boundary and the type of the contents of the referred regions. The runtime of Fail-Safe C keeps track of them by using custom memory management routines.

A *memory block* is the atomic unit of memory management and boundary detection. Each block consists of a *block header* and a *data block*. A block header contains information on its size and its dynamic type, which we call *data representation type*. Figure 3 illustrates a pointer and a memory block it refers to.

The actual layout and representation of the data stored in a block may depend on its data representation type. For example, we use a simple array structure identical to those of usual compilers for data of type `double`, and a packed array of two-word encoded pointers for data of pointer types (e.g. `char *`).

Virtual Offset. Although the actual representation of data in a memory block is different from that of conventional compilers, the difference should be hidden from user programs. (If a program can access the base field of a pointer by tweaking the offset of a cast pointer, the safety invariants on which the whole runtime system relies may not hold.) For this purpose, we introduce a notion of *virtual offset* to handle all memory accesses by user programs. Virtual offsets are the offsets visible to user programs. More specifically, the virtual offset of an element e in a memory block B is the difference of e's address from B's base address *in conventional compilers*. For example, the virtual offset of the second element of a pointer array should be *four* in 32 bit architectures, even if it is actually stored at the 8–15th bytes of the memory block.

Type Information and Cast Operation. The TypeInfo field of a memory block is used to support cast operations. The field points to a type information structure, which is generated one per each type appeared in the programs. Whenever a pointer is cast to another type, the runtime compares the typeinfo of the region referred to by the pointer, with the target type of the cast operation. If the types do not match, the runtime set the cast flag of the resulting pointer.

The TypeInfo is also utilized when a cast pointer is used. Although the ANSI-C standard does not support memory accesses via cast pointers, an ill-typed memory access is sometimes safe, e.g., when reading the first byte of a pointer. Actually, a usual C programmer is very skillful in using it and likes to use it. We have decided to allow ill-typed memory access as far as possible unless it collapses runtime memory structures since it appears frequently in usual application programs.

However, it is not always safe to allow arbitrary updates to memory regions. For example, neither overwriting the first byte of an array of pointers by an integer, nor overwriting any byte of the regions pointed to by function pointers is safe. Furthermore, actual operations needed for memory accesses depend on the data representation type of memory regions. This implies that the type check code must know the actual layout of the memory regions and must change its behavior according to the type. We solve this problem by a method quite similar to the so-called virtual function table in C++ implementations.

Every type information structure has a handler method table that contains function pointers to access handler methods for read and write operations. Handler methods have interfaces which are independent of actual data representations in memory blocks. If a pointer with its cast flag set is dereferenced, the read handler method associated with the referred region is called. The method converts the referred contents of the region to a generic fat value and returns it. If a value is written through a cast pointer, the handler checks the runtime type to set the cast flag appropriately. Thus using so-called "method indirections", we can safely allow programs to access the memory via ill-typed pointers (e.g., reading four `chars` through a pointer of type `int *`), even if we adopt different representations for various types as a means of optimization.

Deallocation. Liveness of memory blocks is maintained by garbage collection to prevent the creation of dangling pointers. The library function `free()` does not actually release any memory block, but marks the memory block as "already released" by changing its size field to zero. This disallows further access to that block, yet preventing dangling pointers from capturing another objects allocated at the same location.

3.3 Memory Operations

Dereference of a pointer is not trivial. We need to know if the pointer refers to a valid region and if the type of the target value is correct. The Fail-Safe C compiler translates it to the following sequence of operations.

1. Check if the pointer is `NULL` (i.e., base $= 0$). If so, signal a runtime error.
2. If the cast flag is not set, compare the offset with the size of the memory region referred to. If it is inside the boundary, read the content in the memory. Otherwise, signal a runtime error.
3. If the cast flag is set, get the pointer to the handler method from the type information block of the referred region, and invoke it with the pointer as an argument. The value returned from the handler method is converted to the expected result type.

If a pointer is non-null, our invariants on the pointer value shown in Section 2.2 ensures that the value in the base field always points to a correct memory block. Therefore, the data representation type and the size of the referred block is always accessible even if the pointer has been cast.

In step 2, if the cast flag of a non-`NULL` pointer is not set, the invariants ensures that the referred region has the data representation type expected by the static type of the pointer. Thus exactly one storage format can be assumed. However, if the cast flag is set, the actual representation of the referred block may differ from statically expected one. In this case, the code sequence delegates the actual memory read operation to the handler method associated to the referred block.

Store operations to the memory are performed in almost the same sequence. If a pointer has ever been cast, its handler method performs appropriate cast operations to keep the invariant conditions on its stored value.

3.4 `Malloc()` and Other Allocation Functions

Our guarantee of type safety heavily relies on the type information associated with each memory block. However, the interface of the `malloc()` function in the standard C library does not take any type information. Thus, we have to recover the missing information. Our approach is to "guess" the intended type by analyzing a cast expression whose operand is the return value of the `malloc()` function, and we use an additional argument for the function to pass the type information. If the compiler fails to guess the type, the `malloc()` library function prepares a "large-enough" region, and marks that region as "type undecided". The type information is post-filled upon the first write access to that region, using the write handler method mechanism described before.

We apply this approach not only to the malloc() function but to a more general set of functions. We assume that each function returning `void *` need a hint about its return type. For each, we add a hidden type parameter to its interface. If such a "polymorphic" function returns a value which is returned from another polymorphic function, we pass the received type hint to the subsidiary function (see Fig. 4 for example). In this way, various `malloc()`-wrappers can be supported naturally without rewriting source code.

```
void *xalloc(size_t size) {
  void *p = malloc(size);
  if (p) return p;
  abort();          /* allocation failure */
}
```

(a) original source.

```
fat_pointer *xalloc(typeinfo_t __rettype, size_t size) {
  fat_pointer *p = malloc(__rettype, size);
  if (p) return p;
  abort();
}
```

(b) translated program (example).

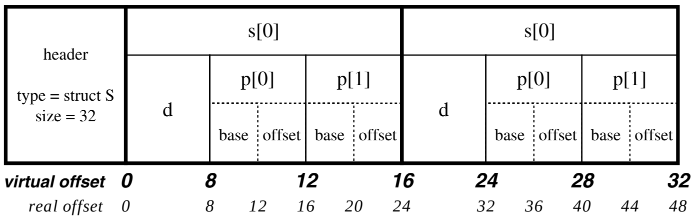
Fig. 4. An example of type hint passing

3.5 Structures and Unions

When a structure or an array of structures appears in a program, the compiler unfolds the nested structures, and lay each element of the structure continually (except required padding for alignments) on memory. Only one header is added for each array of structures (see Fig. 5). The virtual offset is counted on the basis of the virtual size of each member. The general invariants on pointers to scalar also apply to pointers to structures: a pointer to the inside of an array of structures has the cast flag off only if its type is a pointer to that structure type and its offset is the multiple of the size of the structure.

In this approach, pointers to members of the structures will have the cast flag on. Therefore, taking the address of an internal member (e.g., `&(a->elem)`) may have some performance penalties. We are currently implementing a pointer range analysis which can reduce this performance penalty. Direct member accesses (e.g., `a.elem`) and member accesses through pointers to structures (e.g., `a->elem`) have no additional performance penalties compared with scalar accesses.

```
struct S {
  double d;
  char *p[2];
} s[2];
```



(We assume the field d never holds any pointer value.)

Fig. 5. An example of the representation of structures

Unions are decomposed to the types of the members, and operations are converted as if there is an explicit cast operation between those types. If a region of union type is initialized (i.e., written for the first time), the TypeInfo field of the region is set to the type of the referenced member of union. Thus the pointer to the union type always has cast flag on. If the value of a pointer to a member of a union is created, the creation is treated as a cast operation: the cast flag is cleared if the referenced member (type) matches the firstly written member (type). If the type does not match, the handler methods associated with the region absorb the difference between structure representations.

For more rigorous and clearer presentation, we plan to formalize the above treatment of structures and unions.

3.6 Function Pointers and Varargs

Pointers to functions are often cast to other types and then called. For example, programmers usually write comparator functions used in `qsort` as functions taking two pointers to elements, and cast them to type `int (*)(void *, void *)`. The `qsort` function calls such a function with two `void *` arguments, and the arguments are implicitly cast to the expected type. Fail-Safe C supports idiomatic operations of this kind.

Functions whose addresses are taken by the `&` operator are represented by small constant memory blocks. Two entry points are generated by the compiler for each function whose address is taken, and their addresses are stored in the block. The first entry has a normal interface, and used in all cases except for calls through cast pointers. The second entry has an interface which is common to all functions: all arguments are packed into one array of fat values, and results are also encoded to fat values. If a function is invoked through a cast pointer ($f = 1$), the callee converts all arguments to the universal representation (fat integers), calls the second entry, and converts the result back into the expected type. Variable-number arguments (varargs) can also be supported using this scheme. All optional arguments are passed as uniform argument arrays.

3.7 Local Variables

Taking the address of a local variable could be dangerous, as it may result in a dangling pointer if the pointer escapes from the scope of the function. Therefore, if the address of a local variable is taken, that variable should be allocated in a heap rather than on the stack. Appropriate allocation code is inserted by our compiler. Alternatively, an optimizer can perform escape analysis to allocate some of those object blocks on stack. A local variable whose address is not taken can be treated in the standard way: they can be stored either on the stack or in a register.

3.8 External Libraries

Since system calls and lots of C libraries do not respect our representations of pointers and integers, those functions are implemented via wrapper interfaces. A common sequence for such wrappers are:

1. convert scalar arguments,
2. check the boundary of array arguments with predefined constraints,
3. copy the contents of array arguments to a temporally allocated array,
4. call the original function,
5. write back any modification to the array arguments,
6. convert the return value and return it to the caller.

Steps 3. and 5. can be omitted if the internal representation of data is the same as those of a native implementation (e.g., if the dynamic type of an argument is `double[]`). Since the C-level types of library functions do not have enough information to generate those wrappers, some manual intervention is needed.

Some functions are too complex to be implemented in the above common way. For example, the meaning and type of the third argument of the Unix system call `ioctl()` depends on the first and second arguments. So the wrapper must parse those arguments to check the correctness of the third argument. The well-known `printf()` function is another example of such a function. In addition, functions like `gets()` or `scanf()` have possibly unlimited side effects—it may write even over the whole memory if the input does not contain the expected delimiter character. In those functions, boundary checks can be performed only during the operations. Functions such as `strcat()` and `strchr()` are other examples where wrappers cannot perform boundary checks in advance. We provide hand-written custom wrappers for those functions—or, alternatively, we can just compile them from their source code by using our Fail-Safe C compiler.

4 Preliminary Experiments

4.1 Preliminary Implementation

We implemented a preliminary compiler system of Fail-safe C. To estimate the performance of our system, we conducted small-scale experiments. The compiler

Table 1. Result of experiments

	Program	Native	Safe version	Ratio
Pentium III 1 GHz	Fib	0.648	0.736	1.14
	Int qsort	0.310	1.243	4.01
	Poly qsort	0.821	3.422	4.17
	Program	Native	Safe version	Ratio
UltraSPARC II 400 MHz	Fib	1.301	1.314	1.01
	Int qsort	0.467	2.075	4.44
	Poly qsort	1.993	16.278	8.17

[unit: s (except for ratios)]

consists of a C-to-C source code translator, implemented in Objective Caml, and runtime libraries implemented in the native C language. The translator reads a program, converts representations of pointers or integers, and inserts appropriate check code, most of which are inlined. The converted code is then passed to a native C compiler and compiled into a machine language. In this experiment, garbage collector is not implemented yet.

The translator removes “useless” conversion of integers between native and two-word representations, that is, when the two round-trip conversions appear in sequence in the resulting program. No other optimizations are performed at the source-to-source translation stage, intended to know the *worst-case* estimation of the runtime overhead of our scheme.

4.2 Test Programs

We chose the following three micro-benchmarks as test programs, since these programs represent typical kinds of operations in usual C programs. For each test, we compared the execution time of the translated program to that of the program compiled directly by gcc (version 2.95.2 with `-O6`).

In the first test, we want to measure overheads due to fat integers (arithmetic operations, and increase in function arguments and return values). The second, called int-qsort, sorts an array filled with one-million random integers in ascending order using the quick sort algorithm. This shows the overhead of fat pointer operations, such as boundary and cast flag checks. The last, called poly-qsort, is a polymorphic version of the above quick sort test. It takes a comparator function, a data swap function, and the element size of a target array as additional parameters, thus being able to sort an array of any element type. We use the same input data in int-qsort.

All tests were performed on two different platforms: one is an x86 PC system with a 1 GHz Pentium-III CPU, and the other is Sun’s Ultra Enterprise 4500 with 400 MHz UltraSPARC II CPUs.

4.3 Analysis of the Results

The results are shown in Table 1. The overhead of “fib” was between 1% and 14%. In this test, the argument and the return value of the fib function are increased to two words, and before each addition which originally appears in the source code, two word integers are coerced to one word by adding base and offset. Thus, the one step calculation of fib performs three additions and two invocation of function with a two-word argument, which were originally one addition and two invocations with an one-word argument. It seems that the overhead of arithmetic and function invocations introduced by two-word integer representation is relatively small.

In int-qsort, the overhead of safe pointer operations makes the program execution 3–3.5 times slower (including integer overhead). This result is not quite satisfactory, but we believe that the overhead can be considered as a trade-off of efficiency for security in certain application domains. Moreover, we expect that implementing optimizations such as common subexpression elimination([9], for example) and redundant boundary check elimination in our compiler system will reduce this overhead.

Finally, the execution time of poly-qsort in Fail-Safe C is around 4–8 times as large as the original. This is caused by 1) the additional overhead of runtime type checking, and 2) our current poor implementation of function pointers.

We are aware that the programs used in this experiments are very small. We will perform more extensive tests with larger programs as soon as the full implementation is completed.

5 Related Work

There are many related research activities. They can be roughly classified into two categories.

The focus of the first category is to detect various error states at runtime which are common to many programs. For example, Purify [5] and Sabar-C (CodeCenter) [8] dynamically detect memory access errors caused by array boundary overflow. In addition, StackGuard [3] and many other recent implementations focus on preventing well-known security holes such as stack buffer overflow used by many recent security attacks. However, all of them detect only some common patterns of memory misuse, so complete safety is not guaranteed. Loginov et. al. [10] proposes a method to keep pointer safety by adding a 4-bit tag to every octet in the working memory. Backward-compatible bounds checking by Jones and Kelly [7] have modified gcc compiler to insert bounds checking code which uses a table of live objects. By their approach, it is impossible to access a memory which is exterior of any objects (e.g., function return addresses in the stack), but it can still reads and modifies any data in the memory by forging pointer offsets. Jones and Kelly claims their method detects pointer offset forging, but it does not seem to work when on-memory pointers are overwritten by integers.

Safe-C [1] can detect all error caused by early deallocation of memory regions. However, their paper does not mention about cast operations and, as far as we know, supporting unlimited cast operations with Safe-C seems non-trivial, for the same reason as that of Jones and Kelly's work. Patil and Fischer [14] proposed an interesting method to detect memory misuses. Their work perform boundary checking in a separate guard process and use program slice techniques to reduce runtime overhead. However, it has some limitation on the source and destination types of cast operations.

Another category works on safe imperative languages that *resemble* to the C language. The major examples of such languages are CCured and Cyclone. To conform common C programs to Cyclone [4,6], it is reported to require rewriting about 10% of program code (which we consider not so small). Extremely speaking, Java and (the type-safe portion of) C# can also be considered to be examples of such languages, but of course porting C programs to those languages is more burdensome.

CCured [11] supports cast over pointer types. The approach of CCured is to analyze a whole program to split the program into two parts: one is the "type-safe part" which does not use cast operations, and the other is the "type-unsafe part" which can be poisoned by cast operations. However, to our knowledge, they do not focus on source-level compatibility to existing programs, and in fact it supports only a subset of ANSI-C semantics. We are focusing on complete compatibility with the ANSI-C specification, and on the highest compatibility with existing programs. The main technical difference between CCured and our work is that our system allows to use optimized representation for data even if it is pointed from unsafe pointers. Because "wild pointers" in CCured have only a base field and an offset field, the representation of a wild pointer's target must be unique, and thus they cannot point to data in the "safe part" of a program. This means that all objects which can be traced transitively from one single wild pointer must adopt slow and fat representation. We suppose that such a kind of data structures can not be neglected for many programs in the real world, although this supposition must be verified by detailed analysis of existing programs.

6 Current Status and Future Work

6.1 Current Status

We are currently implementing a complete version of the Fail-Safe C compiler. We have already implemented type analysis and program flow analysis, and are currently working on data flow analysis for removing redundant use of fat pointers as much as possible. We will formalize the whole execution model of Fail-Safe C and finish the implementation soon. Then, we are planning to analyze various existing server programs to evaluate the source-level compatibility of our compiler with conventional compilers, and also plan to measure the overhead of our scheme as well as the possibility for further optimization.

6.2 Future Work: Beyond ANSI-C

In this paper, we focus on how to support all the ANSI-C features safely by using runtime checks. We are aware, however, that ANSI-C specification is too restrictive and only few of existing programs are fully ANSI-C compliant. The compiler which only supports fully ANSI-C compatible programs may not be useful for general purposes. For example, ANSI C does not define the behavior of a program which casts a pointer to an integer and take the hash index of it. In addition, the interface of well-known Unix system calls such as `getsockaddr` relies on pointer casting undefined in ANSI-C specification.

For these reasons, our Fail-Safe C system actually accepts some superset of ANSI C language, which is still sufficient to prevent serious misbehavior of programs. The implementation techniques we have described in this paper, especially those of memory block representations and handler methods, are carefully designed to allow extensions beyond the ANSI-C specification which accept many idioms common in existing Unix/C programs. For example, although the result of integer overflow is undefined in ANSI C, our system allows it because it does not cause any unsafe situation by itself.[4] For another example, reading an array of floats via an integer pointer, which our compiler supports by emulation using handler methods, is "safe" in terms of security. Supporting those operations increase the compatibility between Fail-Safe C and conventional implementations of C.

However, our current support for such "beyond ANSI-C" programs is not yet enough to accept many existing programs. For example, using the least significant bits of pointers for the purpose of tagging, which is common in interpreters of dynamically-typed languages, cannot be supported in our current schema, because setting the least significant bits by using integer arithmetic loses the base addresses of pointers.

Thus, we plan to examine existing programs and implement various compatibility features, some of which might be ad-hoc, to our system, provided that they do not affect base-line safety requirements. Of course, not all C programs can be supported on our system. Programs which really depend on the exact structure of memory and hardware, for example operating system kernels, and programs which generate execution code dynamically (e.g., A language runtime with just-in-time compilers) can not be supported at all and are out of the scope of our research. Also, as for programs which allocate memory as a bulk and perform their own memory management inside (e.g., the Apache web server), the protection of such memory regions would be weak (because the allocated bulk memory would become one memory region in our implementation, but execution code injection attack is still impossible) and severe performance penalty

[4] We think that defining the behavior of integer overflow in terms of the modulus of 2^{32} or 2^{64} is a reasonable choice for C language implementations. Recently reported security holes related to integer overflow are actually not caused directly by the overflow itself but caused by wrongly implemented array boundary check which goes wrong by integer overflow. As our system performs its own boundary check in a safe way, this does not affect security in Fail-Safe C.

would be caused (because almost all pointers would have cast flag set). One solution for those programs is to remove such custom memory allocators and use the allocator which our runtime provides, but it requires rewriting part of the programs.

At the same time, we may as well give a second, more careful thought to the support of those extended features from the viewpoint of debugging and "further safety". Although some programs really need such "beyond ANSI-C" features as above, those weird operations are simply a bug in many cases. It may sometimes (e.g., for debugging) make more sense to stop the program execution *as soon as* those suspicious operations are performed *before* it reaches to critically dangerous situations such as overrunning buffer boundaries or accessing forged pointers. We also plan to consider the balance between compatibility and strength of error-detection, and provide various options for programmers.

7 Conclusion

We proposed a method for fail-safe implementation of the ANSI C language, and estimated its performance overhead via preliminary experiments. Even without optimizations, the execution time is limited within 1–8 times of the original. We are still developing a complete version of the compiler which can be used as a drop-in replacement for conventional unsafe compilers.

Acknowledgements. We are very grateful to Prof. George Necula and Prof. Benjamin Pierce for their discussions and comments on an early version of the paper. We are also thankful to the members of Yonezawa group, especially Prof. Kenjiro Taura, for many valuable suggestions.

References

1. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, 1994.
2. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, June 2000.
3. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.
4. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, June 2002.
5. Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*, pages 125–136, 1992.

6. Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.
7. Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.
8. Stephen Kaufer, Russell Lopez, and Sasha Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proc. 1998 Summer USENIX Conference*, pages 161–171, 1988.
9. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy Code Motion. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–234, June 1992.
10. Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2029:217–, 2001.
11. George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proc. The 29th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL2002)*, pages 128–139, January 2002.
12. Yutaka Oiwa, Eijiro Sumii, and Akinori Yonezawa. Implementing a fail-safe ANSI-C compiler. In *JSSST 2001*, Hakodate, Japan, 18 September 2001. Japan Society for Software Science and Technology. In Japanese.
13. Yutaka Oiwa, Eijiro Sumii, and Akinori Yonezawa. Implementing a fail-safe ANSI-C compiler. *Computer Software*, 19(3):39–44, May 2002. In Japanese.
14. Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, January 1997.
15. Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proc. '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195, 2000.
16. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, February 2000.

A The Semantic Rules

Notations

The meta-functions and judgments used in the rules of dynamic semantics have the following meanings:

- $\Gamma \vdash e : t$
  The expression e is typed to t under environment Γ.
- $H, HT;\ e : t \rightsquigarrow H', HT';\ e' : t$
  The expression $e : t$ reduces to $e' : t$ in one step under the heap H and the heap typing HT, and the heap and heap typing are updated to H' and HT' after reduction respectively.
  The static type t does not change during evaluation, and is omitted when it is unimportant.

- $H, HT;\ e : t \rightsquigarrow \mathrm{fail}$
  The reduction of the expression $e : t$ causes a detected runtime error under the heap H and the heap typing HT.
- $H[(b, o) := v]$
  The heap H with the o-th element of $H(b)$ updated to v.
- $H[b := l]$
  The heap H with a new element l added as $H(b)$.
- $HT[b := t]$
  The heap typing HT with a new element t added as $HT(b)$.
- $e[v/x]$
  The expression e with all free occurrences of the variable x substituted with the value v.
- $v \times s$
  A vector of length s with all elements initialized by v.

Static Semantics

We allow both integers and pointers to have *fat pointer* values.[5]

$$\frac{}{\Gamma \vdash \mathrm{num}(n) : t} \qquad \frac{}{\Gamma \vdash \mathrm{ptr}(b, o, f) : t}$$

Typing rules for memory operations are as follows:

$$\frac{\Gamma \vdash e : t \text{ pointer}}{\Gamma \vdash *e : t} \qquad \frac{\Gamma \vdash e_1 : t \text{ pointer} \quad \Gamma \vdash e_2 : t}{\Gamma \vdash *e_1 = e_2 : t}$$

We allow cast operations between arbitrary types.

$$\frac{\Gamma \vdash e : t'}{\Gamma \vdash (t)\, e : t}$$

The second operand of an addition must be an integer. The first operand may be either a pointer or integer, and the return value has the same type as the first operand.[6]

$$\frac{\Gamma \vdash e_1 : \mathrm{int} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : \mathrm{int}} \qquad \frac{\Gamma \vdash e_1 : t \text{ pointer} \quad \Gamma \vdash e_2 : \mathrm{int}}{\Gamma \vdash e_1 + e_2 : t \text{ pointer}}$$

The typing rules for the other expressions are obvious.

$$\frac{\Gamma \vdash e : \mathrm{int}}{\Gamma \vdash \mathrm{new}\langle t\rangle(e) : t \text{ pointer}}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma,\ x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \mathrm{let}\ x : t_1 = e_1 \ \mathrm{in}\ e_2 : t_2}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

[5] Constants which appear in a source program will never have a pointer value, but our typing rule allows them because the transition rules cause such terms during evaluation.

[6] ANSI C allows expressions of the form "integer + pointer", but it can be obviously canonicalized to this form by commutativity.

One-Step Reduction of Innermost Expressions

The rules for dereference, update, cast and additions are already shown in Section 2.

Let

$$H, HT;\ \mathrm{let}\ x : t' = v : t'\ \mathrm{in}\ e : t \rightsquigarrow H, HT;\ e[v/x] : t$$

Allocation. The `new` expression allocates a new block in the heap, and updates H and HT. The complex part of the side condition in the following rule assures the "freshness" of the address of the allocated block, that is, if two different pointers point to two valid locations in the heap, the integer interpretation of the pointers must also differ. The size of the allocation is converted to a simple integer, and the allocation fails if the requested size is not positive.

$$\frac{\forall b' \in H.\ \forall o'\ \text{inside}\ H(b').\ (b' + o') \notin [b, b+s-1] \qquad s > 0}{H, HT;\ \mathrm{new}\langle t\rangle(\mathrm{num}(s)) \rightsquigarrow H[b := \mathrm{num}(0) \times s], HT[b := t];\ \mathrm{ptr}(b, 0, 0) : t\ \text{pointer}}$$

$$\frac{s \le 0}{H, HT;\ \mathrm{new}\langle t\rangle(\mathrm{num}(s)) \rightsquigarrow \mathrm{fail}}$$

$$H, HT;\ \mathrm{new}\langle t\rangle(\mathrm{ptr}(b, o, f)) \rightsquigarrow H, HT;\ \mathrm{new}\langle t\rangle(\mathrm{num}(b + o))$$

One-Step Reduction of Expressions

An evaluation context C determines the order of reductions.

$$\begin{aligned} C ::= &\ [] \mid *C \mid (t)C \mid C + e \mid v + C \mid *C = e \mid *v = C \\ &\mid \mathrm{let}\ x = C\ \mathrm{in}\ e \mid \mathrm{let}\ x = v\ \mathrm{in}\ C \mid \mathrm{new}\langle t\rangle(C) \end{aligned}$$

A one-step reduction $\longrightarrow$ is defined by using innermost reductions $\rightsquigarrow$ and the evaluation context. The meaning of H, HT is the same as those of innermost reductions.

$$\frac{H, HT;\ e \rightsquigarrow H', HT';\ e'}{H, HT;\ C[e] \longrightarrow H', HT';\ C[e']} \qquad \frac{H, HT;\ e \rightsquigarrow \mathrm{fail}}{H, HT;\ C[e] \longrightarrow \mathrm{fail}}$$