Implementation of the Memory-safe Full ANSI-C Compiler

Yutaka Oiwa

Research Center for Information Security (RCIS)
National Institute of Advanced Industrial Science and Technology (AIST), Japan
y.oiwa@aist.go.jp

Abstract

This paper describes a completely memory-safe compiler for C language programs that is fully compatible with the ANSI C specification.

Programs written in C often suffer from nasty errors due to dangling pointers and buffer overflow. Such errors in Internet server programs are often exploited by malicious attackers to crack an entire system. The origin of these errors is usually corruption of in-memory data structures caused by out-of-bound array accesses. Usual C compilers do not provide any protection against such out-of-bound access, although many other languages such as Java and ML do provide such protection. There have been several proposals for preventing such memory corruption from various aspects: runtime buffer overrun detectors, designs for new C-like languages, and compilers for (subsets of) the C language. However, as far as we know, none of them have achieved full memory protection and full compatibility with the C language specification at the same time.

We propose the most powerful solution to this problem ever presented. We have developed *Fail-Safe C*, a memory-safe implementation of the full ANSI C language. It detects and disallows all unsafe operations, yet conforms to the full ANSI C standard (including casts and unions). This paper introduces several techniques—both compile-time and runtime—to reduce the overhead of runtime checks, while still maintaining 100% memory safety. This compiler lets programmers easily make their programs safe without heavy rewriting or porting of their code. It also supports many of the "dirty tricks" commonly used in many existing C programs, which do not strictly conform to the standard specification. In this paper, we demonstrate several real-world server programs that can be processed by our compiler and present technical details and benchmark results for it.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors – Compilers

General Terms Languages, Security

Keywords Memory Safety, C language

1. Introduction

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

Today, C remains one of the major languages for writing application programs, including those running on various Internet servers. As requirements for applications have become more complex, though, programs written in C have often been used to perform very frequent complex pointer manipulations. This has created serious security flaws. In particular, if in-memory data structures are destroyed by array buffer overflows or dangling pointers, the behavior of a running program becomes completely different from its text. In addition, by forging specially formed input data, malicious attackers can sometimes hijack the behavior of programs that contain such bugs. Most of the recently reported security holes have been due to such misbehavior.

Many countermeasures to this problem have been proposed. Buffer-overrun detection techniques, such as StackGuard [6], ProPolice [7], and dmalloc [21], are well matured and have been introduced into many commercial and open-source compilers, although they prevent only some of the possible attacks. There have been many proposals for C-like languages that satisfy type safety and memory safety, such as Cyclone [9, 11]. Necula et al. [16, 5] proposed a type-based program analysis for compiling a large subset of C languages and succeeded in compiling existing C benchmarks with a small amount of program modification. However, as far as we know, no one has achieved both *complete memory safety* (e.g., comparable to Java or many Lisp/ML dialects) and *full compatibility with the C language* at the same time.

To resolve the current situation, we have developed *Fail-Safe C*, a special implementation of the full ANSI C language that prevents all of the dangerous memory operations that lead to execution hijacking. Our compiler inserts check code into the program to prevent operations that destroy memory structures or execution states. If a buggy program attempts to access a data structure in a way that would lead to memory corruption, the runtime system of our compiler system cooperates with the inserted codes to report the error and terminate program execution. Use of our compiler system instead of usual C compilers enables safe execution of existing C programs.

Our compiler supports all programs written in conformity with the 1989 version of the ANSI C specification [10, 1, 14], including casts, unions, arbitrary pointer arithmetics, function pointers (cast and uncast), and variable-number arguments (varargs). Moreover, it also supports many of the "dirty tricks" popular with C

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06…$5.00

programmers in order to ensure it can accept most existing programs. Of course, our compiler cannot accept every existing program: programs that really need raw memory access (e.g., the core part of operating system kernels) conflict with our memory protection scheme. However, our scheme is powerful enough to accept many security-critical programs used on Internet servers, most of which do not need raw memory access. The current implementation of our compiler accepts several existing, realistic server programs such as OpenSSL, OpenSSH, and BIND9.

This paper describes several implementation techniques and designs introduced in Fail-Safe C. The rest of the paper is organized as follows: Section 2 describes the basic ideas of our system design to support memory-safe execution of C programs including cast and pointer arithmetics. Section 3 describes some details of the implementation, including tricks for better performance and support for various subtle features of C. Sections 4 and 5 present the current status of Fail-Safe C implementation and some benchmark results for performance evaluation. Section 6 compares the design of Fail-Safe C with those in some previous reports. Section 7 concludes by summarizing the main points and mentioning future work.

This work is based on the implementation method proposed in the author's Ph.D. dissertation [19] with various design and implementation improvements. For more specific details of our implementation, please refer to the dissertation as a full version of this paper.

Note Throughout this paper, we assume that the underlying architecture uses 32-bit pointers and integers for clarity. This is just for an example: our technique, as well as those in most other reports, can support various word widths such as 32 and 64. The term "*word*" refers to the natural size of pointers, i.e., 32 bits or 4 bytes. Example program code shown in this paper was written by hand for clear understanding of concepts: it is not textually equal to the program generated by the compiler.

2. Basic Designs

There are many language constructs in C that are not found in various safe languages (such as Java, C#, Lisps, and MLs) and make safe implementation of the language difficult. First, C supports flexible arithmetic operations on the pointers. In most safe languages, "references" to objects or arrays are designed so that they point only to a whole array or an object. Such languages allow references handled as only opaque data and do not allow arbitrary arithmetic operations on them. This is a common design choice to ease runtime checking of array boundaries: by allowing references that point only to a fixed location inside an object, the language runtime systems can easily access metadata associated with each object such as array sizes and perform required runtime validations. In C, however, pointers are allowed to point anywhere inside array objects, which makes boundary checking hard. Storing the object boundaries (e.g., using memory object tables) does not work well: pointers that have overflowed might even point to the interior of another object.

Cast operations and union types make safe implementation of the language very hard. C is a weakly statically typed language: unlike many other strongly statically typed languages, a pointer in C can be cast to another type so that it points to an object that does not have a data type expected by the static type system. This breaks many assumptions that are essential for ensuring the type safety of statically-typed languages. Many people might think that as long as cast operations exist, it is impossible to implement C in a memory-safe way. Furthermore, there are many more obstacles to safe implementation: function pointers (with cast operations), varargs (common in `printf`-family functions), untyped `malloc` functions, and guaranteed cast operations between pointers and integers.

Our goal in designing our compiler was to invent a set of new implementation techniques that can solve all of the above problems. We introduced the following set of techniques, amongst other things. They are described in more detail in subsections 2.1–2.7. Although some of them are mutually related, they are nevertheless described in sequential order.

Fat Pointer: A pointer that accepts arithmetic operations and still keeps information about the objects being pointed to.

Fat Integer: An integer that can hold a fat pointer value.

OO-based Typed Memory Block: A memory block implemented using object-oriented techniques that stores both its own types and sizes.

Virtual Offset: A special memory addressing technique that hides internal representation of any data types and provides a uniform view over objects of various types.

Access Methods: A set of methods associated with memory blocks; it supports dereferencing operations on cast pointers.

Cast Flag: A hint embedded in fat pointers for shortcutting memory access operations to reduce runtime overhead.

Safe Memory Management: A memory management method that prevents any temporal memory access errors such as dangling pointers and double block releases.

2.1 Fat pointer

To access various meta-information about blocks (e.g., the block size and content type) regardless of the pointer arithmetic, Fail-Safe C internally represents all pointers using two-word *fat-pointer* representations: a pair consisting of a base and an offset.

The base of a fat pointer always keeps the address of the top of a block, and the 32-bit offset keeps the relative position of the element referred to by the pointer from the block top. The special value 0 can be used as a base part representing "null pointers"; i.e., pointer values that do not point to any objects. The offset, which is initially 0, represents the byte offset of the referred element from the block top. In all pointer arithmetic operations, only the offset is modified.

The fat pointer technique itself is not very original: fat pointers or "smart" pointers have already been introduced in various implementations. For example, CCured [16] introduces a three-word representation for "seq" pointers (one of the kinds of pointers in CCured that allow pointer arithmetic inside objects), and many people implement it by hand when they need to simulate internal pointers in existing safe languages. However, the fat pointer in our design has been extended from existing ones in the following ways:

- It holds a special flag called the "cast flag" embedded in a two-word representation.
- The offset holds "virtual offsets" instead of real memory address offsets or element indexes.

Both these features are described in more detail below.

```
int *p = &a;      →   base p_b = &a;
                      ofs p_o = 0;

p++;              →   p_o += 4;
```

2.2 Fat integer

In ANSI-C, integers equal to or larger than the pointer size are required to be able to hold any pointer value. Integer values that

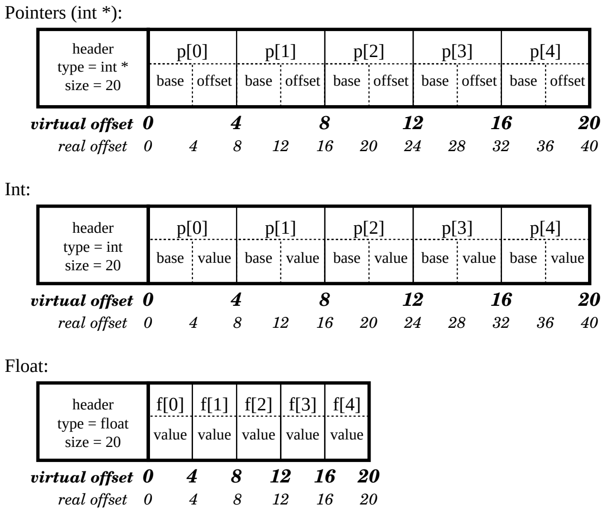

Pointers (int *):
header
type = int *
size = 20
p[0]
p[1]
p[2]
p[3]
p[4]
base
offset
virtual offset 0 4 8 12 16 20
real offset 0 4 8 12 16 20 24 28 32 36 40
Int:
header
type = int
size = 20
p[0]
p[1]
p[2]
p[3]
p[4]
base
value
virtual offset 0 4 8 12 16 20
real offset 0 4 8 12 16 20 24 28 32 36 40
Float:
header
type = float
size = 20
f[0]
f[1]
f[2]
f[3]
f[4]
value
virtual offset 0 4 8 12 16 20
real offset 0 4 8 12 16 20

Figure 1. Block structure for arrays of pointers and primitive types.

were originally pointers can be cast back to corresponding pointer types if the values were not modified while they were integers. To implement this behavior, the usual one-word representation of integers is of course insufficient because we cannot distinguish such integers (valid as pointers) from arbitrary integers. Therefore, we also use two-word representation for integers, which are called *fat integers*.

Conceptually, fat integers could use the same representation as fat pointers. However, to enable more efficient implementation of integer arithmetic operations, a fat integer in our system is internally handled as a pair consisting of the base and a value (or *virtual address*) defined to be equivalent to the sum of the base and the offset. All arithmetic operations on integers ignore the bases of operands and operate on only the value parts. An arithmetic result always has a base part of 0, corresponding to a null pointer. A cast operation between pointers and integers converts offsets to virtual addresses, and vice versa.

```
int i = (int)p;        →   base i_b = p_b;
                           int i_v = p_b + p_o;

int *p = (int *)i;     →   base p_b = i_b;
                           ofs p_o = i_v - i_b;

i *= 5;                →   i_b = 0; i_v *= 5;
```

(Cast flags are not shown in these examples.)

2.3 Typed Memory Blocks

In safe languages, every memory access operation must ensure that the offset and the type of a pointer are valid. If a statically typed language has no casts, or only a cast respecting subtype relations (as in objects in Objective Caml), no runtime type information is needed. However, as C does not impose such restrictions on cast operators, the system must know the boundary and the type of contents for every memory block in order to check this safety property at runtime. The runtime system of Fail-safe C keeps track of these by using custom memory management routines.

A *memory block* is an atomic unit of memory management and boundary overflow detection. Each block consists of a *block header* and a *data area*. A block header contains information about the block's size and its dynamic type, which we call the *data representation type*.

The actual layout and representation of the data stored in a block depend on its data representation type: a different representation is used for each representation type in our system. This allows the implementation to utilize several different representations for each type appearing in user programs. For example, if there is an array of pointers, it is stored as a collection of packed fat pointers (Figure 1). This is also true for arrays of fat integers. However, for an array of simple primitive types (which cannot hold pointers), such as characters or floats, we use a packed array without any base parts, to gain better performance for I/O operations (If the representation of `char` arrays is the same as the native one, the runtime system can use the internal address of data directly for file I/O operations). For each user-defined struct, the compiler automatically generates more complex runtime representations (an example is shown in Figure 2), and generates various related data structures as well.

In our current implementation, all unions are translated as a combination of casts and structs. For example, if there is a declaration

```
union {
  int a;
  char *b;
  double c;
} x;
```

it will be translated as if it were a struct holding only the first element and any necessary padding (if any):

```
union {
  int a;
  char __pad[(sizeof(double) - sizeof(int)];
} x;
```

A member access "`x.b`" is treated as an equivalent code

```
*((char **)&(x.a))
```

and translated to a code which uses cast pointer handling.

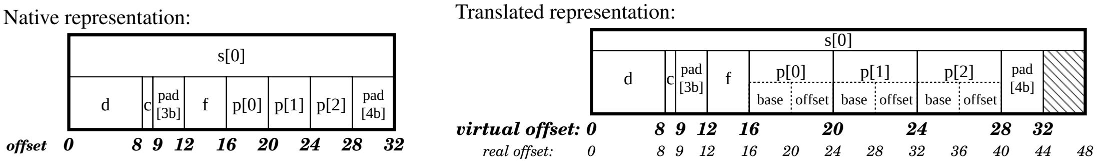
2.4 Virtual offsets

Several methods may be used to indicate a specific element in a memory block. The usual methods used in conventional language implementations use one of the memory addresses of elements, the index count of elements from the block top (sometimes called the word offset), or the difference between the memory address and block top address (the byte-offset). For most implementations, some or all of these will work.

In our system, the situation is more complicated because there is a cast operation that needs to be implemented safely and consistently. The method using real addresses or offsets of real addresses creates a safety problem (although these are used for many existing systems aimed at making C secure): if a pointer is cast to the `char *` type, it will point to every byte of the internal representations of several data including pointers. If such internal information about pointers (e.g., base parts of fat pointers) are compromised through these cast pointers, the safety of the system is no longer ensured. Several proposed systems, including Safe-C [2] and BCC [12], seem to suffer from this problem. CCured [16, 5] solves the problem by maintaining a bit-array for each memory block to indicate whether each word in the block can be used as a valid pointer to the block top; however, the handling is rather complex and unintuitive.

The element index does not have a similar problem for primitive types if alignment requirements are equal to the size of the corresponding type. However, this complicates the implementation of cast operations and makes it impossible to properly represent a cast pointer to data types having alignment requirements smaller than

Example of `struct { double d; char c; float f; char *p[3]; } s[1];`


Native representation:
s[0]
d
c
pad [3b]
f
p[0]
p[1]
p[2]
pad [4b]
offset 0 8 9 12 16 20 24 28 32
Translated representation:
s[0]
d
c
pad [3b]
f
p[0]
p[1]
p[2]
base
offset
base
offset
base
offset
pad [4b]
virtual offset: 0 8 9 12 16 20 24 28 32
real offset: 0 8 9 12 16 20 24 28 32 36 40 44 48

1. A 3-byte padding labeled “pad[3b]” aligns field f to a word boundary in both virtual and real addressing.
2. A 4-byte padding labeled “pad[4b]” aligns the whole struct to a double-word boundary (required by double-word aligned field d) in the native addressing.
3. A 4-byte padding at the last word of the translated representation aligns the whole structure to a double-word boundary in the real addressing. This padding is invisible to the user program.

Figure 2. Example of the representation of a struct.

the element size (e.g., structs); that is, the C specification allows pointers that are not aligned to elements.

Consequently, another method of addressing had to be invented for our system. The addressing used in Fail-Safe C, which is called *virtual offset*, corresponds to the *program-visible* size (hereinafter called the *virtual size*) of elements, not to the actual size of representations altered to implement security mechanisms. For example, the virtual size of a natural-sized integer in our system will be equal to the native word size—although these values use two-word representation internally—because its value range visible to the running user program will still correspond to one word. The virtual size of pointers will also be one word, and floating numbers and smaller integers (which cannot hold pointer values) will have virtual sizes equivalent to their real sizes. Figures 1 and 2 shows some examples of the differences between virtual and real offsets.

In other words, the virtual size of every type will be the real size of the equivalent data type in the native implementation of C. This definition of virtual offsets does not lead to the problems that arise with the other two methods: a cast pointer temporarily pointing to the middle of elements can be properly cast back to its original type and pointing to only the base part of fat pointers is impossible because there is no specific offset that points to only the base part of pointers.

Another important consequence of this representation is the possibility of consistent definition for memory access performed via cast pointers. Although the ANSI-C standard does not support memory access via cast pointers, an ill-typed memory access is sometimes safe (e.g., when the first byte of a pointer is read). Actually, C programmers are often skilled at using this sort of access and find it useful. Fail-Safe C allows the use of ill-typed memory access as far as possible unless it collapses runtime memory structures since such accesses appear frequently in most application programs. Because the virtual sizes used in Fail-Safe C correspond to real sizes in native implementation, the mapping from our representation of data to the corresponding native representation can be defined simply; in short, we can simulate “what will happen when the operation is performed in the native implementation”. For example, if there is an integer 0x12345678 (hexadecimal) and one reads a single byte from that address via a cast pointer, it will be read as either 0x78 or 0x12 (depending on the byte order). The same holds when there is a pointer to the offset 8 of the memory block at address 0x12345670.

2.5 Type information and access methods

As the actual representation of data in a memory block differs from that in conventional compilers, some methods to support memory access via a cast pointer must be provided for every combination of a pointer type and a block representation type. Our compiler uses an object-oriented implementation technique for this purpose.

In the header of each block, there is a *typeinfo* field that contains a pointer to a block containing several items of information about its representation type. One type-information block is generated for each representation type that appears in a user program. Furthermore, a method table similar to that usually used in a C++ implementation is stored in the type-information blocks.

Access methods stored in the method tables implement a generic interface for read/write block contents with various access sizes—such as a byte or word, regardless of the type of block. A read method receives a virtual offset and returns the corresponding content in a data area as a fat integer[1] if the given virtual offset falls inside the block boundary. The write method receives a virtual offset and a value to be written as a fat integer and performs an appropriate write operation depending on the actual data representation. These methods will signal a runtime error if the virtual offset is outside the block boundary.

Type information blocks and access methods for primitive types are implemented in the runtime library provided in the compiler distribution. For user-defined types such as structs or pointers, the compiler system automatically generates them for every type used in the program.

2.6 Faster memory access and cast flags

Given typed memory blocks, access methods, and fat pointers, there is always a simple way to access the contents of memory blocks, regardless of pointer casts. To access word data referred to by a non-null fat pointer, the program can always look up type information in the block header using the base part of the fat pointer, pick up the associated access method from the type information block, and invoke it. The called access method always knows the representation of the referred data block and returns the corresponding data as a fat integer. This can then be easily converted back to any type that the program wants.

However, as one can easily guess, this process with indirect function invocation is very slow compared with usual unsafe memory access in C, or even compared with memory access in other statically typed languages. Our experiments have shown that execution takes more than 10 times longer even for a very simple program when this access method is used for every memory access. Since this is an unacceptable performance penalty, we implemented a shortcut for memory access.

For every fat pointer, there is a one-bit flag called the “cast flag” embedded in an unused bit in the base part. This flag indicates

[1] Returned values will be narrow (native) integers if the access size is smaller than the word size.

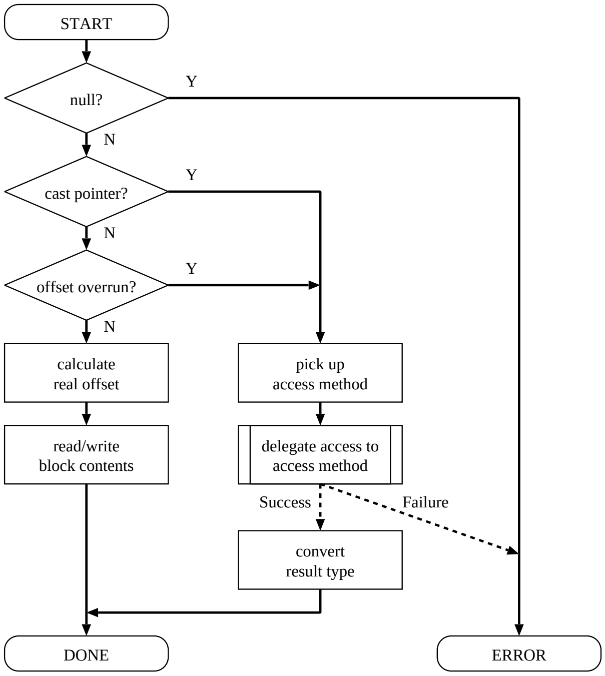


Figure 3. Basic procedure for memory access via pointers.

whether or not a pointer is cast. More precisely, when this flag is zero, and if the pointer is not null, the memory block referred to by the base part must have a correct representation type for the pointer's static type, and the offset part of the pointer must be a multiple of the (virtual) size of the element. This ensures that the following property holds during runtime: if there is a pointer with cast flag = 0, the pointer may be a null pointer, point to a valid element inside the memory block, or point outside the valid memory block.

The introduction of a cast flag enables memory access operations to be optimized as follows (Figure 3):

- check whether the pointer is null; if not,
- check whether that cast flag is set; if it is set, use an access method; otherwise,
- check whether the offset is smaller than the block size; then,
- convert the virtual offset into a real memory offset and access the memory directly.

```
int i = *p;   →
```

```
fat_int i; base i_b; int i_v;
     /* NULL check */
if ((p_b & ~CASTFLAG) == 0)
  raise_error();
     /* cast check */
if (p_b & CASTFLAG != 0 ||
     /* overflow check */
     get_header(p_b)->size <= p_o)
  /* invoke an access method */
  i = (*get_header(p_b)
       ->read_int_method)(p_b,p_o);
else
  /* read memory directly */
  i = *(fat_int *)(p_b + p_o * 2);
          /* real/virtual size ratio for int is 2 */
i_b = base_of_fat_int(i);
i_v = value_of_fat_int(i);
```

The cast flag of a pointer is recalculated each time when a pointer (or integer) is cast to another pointer type. It uses a representation type of the referred block stored in the block header.

```
int *p = (int *)q;   →
```

```
p_b = q_b & ~CASTFLAG;
     /* NULL check */
if (p_b == 0 ||
     /* type check */
     get_header(p_b)->type !=
        type_of_int_pointer ||
     /* alignment check */
     o_q % 4 != 0)
   p_b |= CASTFLAG;
p_o = q_o;
```

As most pointers in C programs are not cast, the introduction of cast flags greatly reduces the runtime overhead of real programs. In the real implementation of Fail-Safe C, the memory access is further optimized by a clever scheme, as described in Section 3.1.

2.7 Safe memory management

Another cause of memory errors in C programs is bad use of memory management routines. Unlike most safe language implementations, memory management in C is completely non-automatic: programmers must specify when heap-allocated memory blocks are deallocated. If they fail to specify the correct timing, there will be a dangling pointer pointing to the debris of an already-deallocated block, which could lead to further invalid memory accesses. In addition, local variables in C can be pointed to by pointers, but they are automatically released at the end of the function's execution regardless of such pointers.

We utilize a garbage-collection technique, as is used in almost all implementations of safe languages, to prevent fatal misbehavior related to the early deallocation of memory blocks. When a user program requests deallocation of a memory block, the runtime system will not immediately release the block, but only forbid further access to it .[2] The garbage collector will later check that there are no pointers pointing to the block, and it will release the memory block only after all dangling pointers pointing to the block have disappeared. It also collects any unreferenced "leaked" objects in memory. The current implementation uses Boehm-Weiser's conservative GC library [3, 4] as a backend library. In future, we may implement a type-exact garbage collector in order to avoid false pointers.

For the local variables, the compiler performs a simple check on the use of local variables and pointers, and if there is a pointer to any local variable, the compiler automatically moves that variable to the heap area. The memory blocks for such variables are allocated at the start of the function, and they are automatically "deallocated" at the end of the function. The area will be then under the management of garbage collectors, like other heap-allocated memory blocks.

3. Implementation Details

This section gives further details of the implementation techniques (tricks) introduced in the compiler to improve performance and compatibility.

3.1 Fast checking of cast flags

As described in the previous section, when a fat pointer is dereferenced, three properties must be checked before any direct access to

[2] This behavior differs slightly from that of most safe languages because user programs are supposed to call the `free()` function to declare explicitly that the memory blocks are no longer to be used.

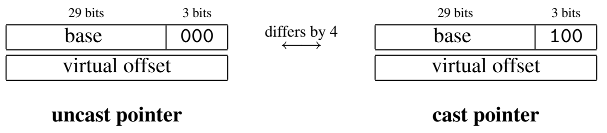


Figure 4. Bit allocation in fat pointers.

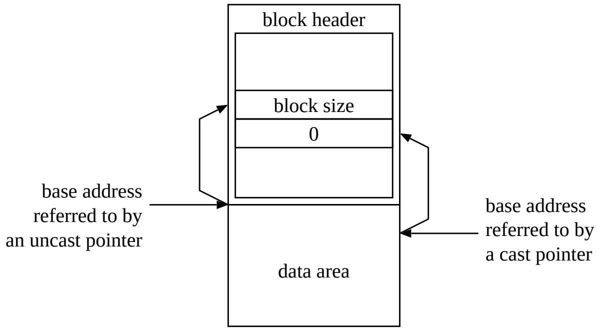


Figure 5. Fast cast-flag check.

the data area of the referred memory block: (1) pointer is not null, (2) the pointer is not cast, and (3) the pointer's virtual offset points to an interior part of the memory block (Figure 3). While (1) and (3) are common to almost all safe languages having flat array types (e.g., Java, ML, and Lisp), in our implementation we also needs (2), whose overhead is not negligible. A preliminary experiment has shown that checking that a pointer is not cast slows the execution of memory-heavy benchmarks by several percent. To avoid this overhead, our implementation uses a clever trick.

First, all blocks and block headers are double-word aligned so that every base address of a block will have 0 as the bit corresponding to the cast flag. Next, the cast flag in fat pointers is located to the "third bit", the one corresponds to the word size (Figure 4) so that the base part of a cast fat pointer will have an integer value which is the real block address plus 4. Finally, each block header has an extra word that always contains a zero at a location exactly one word after the field field containing the virtual block size used for boundary checking. As a consequence of these three properties, if the code refers to the size field of the header via some cast pointer through offset-calculation as if it were not cast, it will read the zero stored in the header block instead of the true size field (Figure 5).

In other circumstances, if a null pointer is dereferenced as if it were a valid pointer, offset checking code that attempts to read the size field will access the very end of the address space (because of an integer wraparound). In most operating systems, no memory is mapped to these addresses and a `SIGSEGV` signal will always be raised if they are accessed. This condition can be reliably detected by checking the address information passed to signal handlers. Thus, those checks can be merged into one offset check, which is generally necessary anyway, without damaging the safety properties. The resulting optimized memory access procedure is shown in Figure 6. Compared with Figure 3, there is only one check on its "fast path".

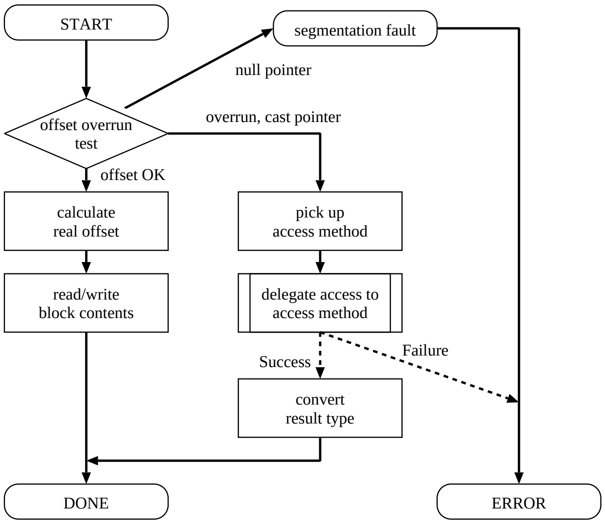


Figure 6. Procedure for memory access via pointers with fast access checking.

```
int i = *p;   →

fat_int i; base i_b; int i_v;
      /* single combined check */
if (get_header(p_b)->size <= p_o)
   /* slow path: invoke an access method */
   i = (*get_header(p_b)
          ->read_int_method)(p_b,p_o);
else /* fast path: read memory directly */
   i = *(fat_int *)(p_b + p_o * 2);
i_b = base_of_fat_int(i);
i_v = value_of_fat_int(i);
```

3.2 Separate Compilation

Separate compilation and reuse of modules are important features in any modern programming language. Virtually all languages currently in use have some provision for separate compilation of modules. In implementations of most statically typed languages, it is important that the type safety of the whole program is guaranteed when type-safe modules are linked together. However, in C, intermodule type safety is not guaranteed in usual implementations. Conventional C linkers hardly check inter-module consistency and simply unify sets of the same symbols. If a function or value in one module is referenced from another module as that of some other type, or if a type definition of structs or unions is inconsistent among modules, the type safety is easily broken in C.

Most existing works on safe C implementation offer no provision for link-time type safety or no separate compilation at all. CCured provides a source-level preprocessor that "links" several source files together before compilation so that any type inconsistency can be found at compile time, but it discards most of the benefits of separate compilation. Frankly, it is a good design choice for CCured, which depends heavily on static analysis even without separate compilation and requires whole-program analysis before compiling a single component module (because the result of whole-program analysis sometimes requires a change in the value representations in the module). However, for our compiler, we took a different approach.

In Fail-Safe C, there is virtually no requirement at all for whole-program analysis. Whether or not a pointer is cast outside one module, the value representation for any type does not change, unlike in CCured. Because virtually no performance overhead is imposed by the existence of a cast pointer thanks to the fast cast-flag checking described above, Fail-Safe C always generates code that can handle cast pointers appearing in any location.

Fail-Safe C implementation provides a special linker that checks inter-module type consistency at link time. The compiler stores information about every type definition appearing in a module in a generated object module file. The linker extracts the information from all modules to be linked and detects any mutual inconsistencies. It also generates all type-related metadata and various access methods associated with the user-defined types (such as structs) at link time to avoid duplicate code in the resulting executable files.

Our linker also implements some tricks for better compatibility: it has special support for undeclared functions and a workaround for the GNU autoconf system. Details of our linker are given elsewhere [20].

3.3 Determination of block types allocated by `malloc()`

The implementation of memory blocks in Fail-Safe C depends on the type information associated with each memory block. However, there are many situations where the block type is not known. For example, the interface for the `malloc()` function in the standard C library does not take any type information. Many existing systems assume that type inference for memory allocation is always possible or ensure this by introducing some explicitly typed memory allocation syntax (like C++'s `new` operator). In contrast, our system does not completely rely on static knowledge of types, but combines static (compile-time) and dynamic (runtime) approaches.

3.3.1 Compile-time type analysis

First, Fail-Safe C implements a combination of static type analysis and a dynamic type-passing mechanism to determine the type of memory allocation. Our system inserts an additional argument internally into any function that returns `void *` type (including `malloc()` and `calloc()`). At each invocation of such a function, the compiler performs a simple data-flow analysis to find out what type the return value will be cast to. If it succeeds in finding such a type, the generated code passes a pointer to the type information block of that type as an additional argument as a hint. Memory allocators such as `malloc()` and `calloc()` can use this hint to initialize a new memory block in the correct type.

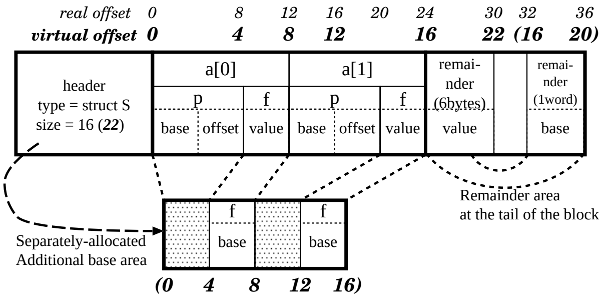
In a special case, when such return values are used as return values of another function returning `void *`, the compiler generates code that passes the hint received from the caller to the child function. This enables the implementation of wrapper functions (such as `allocx`, which is equal to `malloc` except that it aborts the program when an allocation fails) around memory allocation functions. This mechanism also works without whole program analysis, if separate compilation is needed.

If there is no single type that can be used as a hint (e.g., when such a return value is cast to two different types in an `if` statement), the compiler does not pass any hints to the callee, and the runtime system falls back to the runtime type analysis below.

3.3.2 Runtime type analysis (delayed type decision)

If the type cannot be reliably deduced through the analysis above, the Fail-Safe C system delays deciding the type of dynamically allocated blocks from allocation time to the first use.

If a block is allocated without a type hint, the system first assigns a special pseudo-type (called *type-undecided*) to the block. Because this pseudo-type is not equal to any real types, the first write access to this block will always be forwarded to access methods associated with the pseudo-type. Access methods for the "type-undecided" pseudo-type will then guess the block type based on the type used for the access.



Memory layout when a 22-byte heap block is allocated for a struct type with a size of 8 bytes containing a float element. The block contains 2 elements and 6 bytes of remainder area. Two block size values are stored in the header: 16 is the size without the remainder area (this one is used for boundary checks), and 22 is the size of the whole block.

Figure 7. Formats of remainder area and additional bases.

A type-undecided block has basically the same structure as ordinary blocks. The real size of the allocated buffer will be about twice the requested virtual size, as this is sufficient. The type information field in the header points to a specially defined type-information block. In addition, the size of the structured data area is initialized to zero. This causes all accesses to this block to be trapped and delegated to the associated access methods. The write access methods associated with type-undecided blocks initialize the data area according to the access type, which is passed as an additional argument to the access methods. After initialization, its size and the typeinfo field of the block's header is reinitialized to turn the block into a normal block. Finally, the method handles the write request from a caller by delegating it to the newly associated access methods.

As a last resort, if both the block type estimations fails, the memory block may have been assigned the wrong type In that case, cast pointers and access methods will maintain the compatibility and the program will continue running; the only effect is that this slows the execution.

3.4 Memory block extensions

Memory blocks allocated by users using `malloc()` in real programs are used in some weird ways in many real programs. For example, C programmers often allocate a memory area whose size is not a multiple of the size of its data type in order to implement a "variable-sized structure"[3]. They define a 1-element array as the last element of a struct and use it as an array of arbitrary size, which is determined at allocation time. In other cases, some heap-allocated objects can be first initialized as a simple char array and then later used in a different type, which confuses the type determination algorithm in the previous section.

To cope with such tricky programs, our implementation contains two general workarounds. First, the memory management routines maintain a "remainder area" in the heap block, which is not a multiple of the element size. Such a memory area is formatted in the same way as an array of characters and used by access methods to simulate a memory access between the end of all elements and the end boundary of the block. Second, if a non-null pointer is to be written in the array or in a field of a primitive type that does not normally hold a pointer value, the runtime system will allocate a separate cache called an "additional base area" to store

[3] This technique is officially supported in the newer version of the C specification (often called C99).

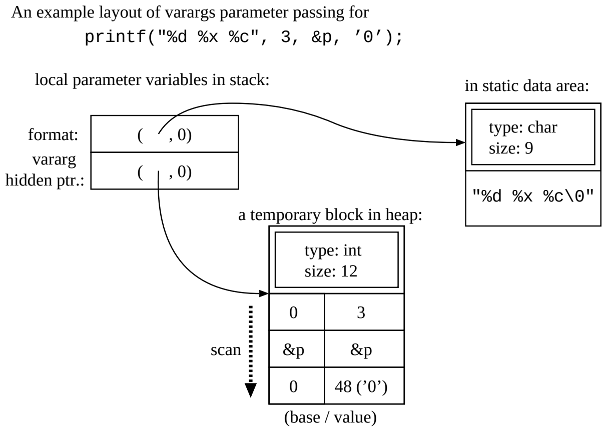

An example layout of varargs parameter passing for
printf("%d %x %c", 3, &p, '0');
local parameter variables in stack:
format:
(, 0)
vararg
hidden ptr.:
(, 0)
in static data area:
type: char
size: 9
"%d %x %c\0"
a temporary block in heap:
type: int
size: 12
0
3
&p
&p
0
48 ('0')
scan
(base / value)

Figure 8. Handling of varargs in Fail-Safe C.

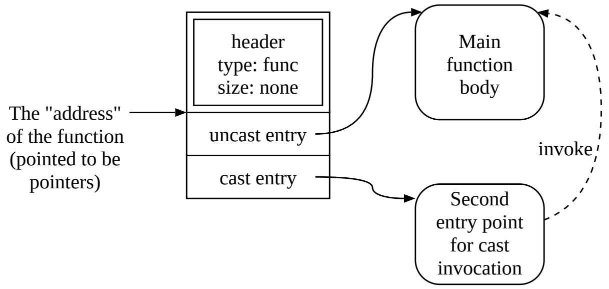


Figure 9. Data structure for functions in Fail-Safe C.

the base parts of the values. This area is also used as a last resort when type determination for heap blocks fails and guessed type was incorrect. The formats of additional base areas and remainder areas are shown in Figure 7.

3.5 Varargs and cast function pointers

Some C functions such as `printf` accept variable numbers of arguments. The ANSI-C specification even allows users to define such functions and handle arguments using macros defined in `stdarg.h`. In usual native compilers for most architectures, such arguments are put onto the stacks in the same way as for other fixed arguments. The corresponding access macros read the stack area directly to fetch variable arguments, which is obviously an unsafe operation.

Our system implements a similar behavior in a safe way (Figure 8). If functions with variable arguments are called, then any parameters that fall into the variable arguments will be automatically coerced and stored in a fat integer array in a heap, instead of the native stack[4]. The array is then passed to the function using hidden arguments and can be accessed through system-defined macros. Of course, block boundaries are checked during accesses to variable parameters, and if excess arguments are accessed, an error is generated.

The same technique is used when a pointer to a function is cast. For each function, the compiler generates a second entry point that receives all parameters as variable arguments, as well as a stub memory block which contains pointers for both the main and the second entry (Figure 9). A pointer to a function is represented by the address of the stub blocks internally. If an uncast pointer is used for an invocation, the caller extracts the address of the main entry point from the pointed stub block and call it directly. However, if a cast pointer is used for an invocation, the caller prepares all arguments as variable arguments and invokes the second entry point instead. The second entry point extracts the real arguments and passes them to the main entry of the function.

3.6 Runtime libraries

Almost all C programs require some library functions and system call functions to work correctly. The standard library requires many functions: the ANSI C and POSIX specifications of Unix system calls define more than 1000 functions. We have already implemented more than 500 functions, mostly by hand. Some functions, such as many string-related functions in `string.h`, can be implemented in a usual C language and compiled by Fail-Safe C.

However, some functions, especially ones that interact with underlying systems (including kernel-level system calls) cannot be implemented in that way. For those functions, we implemented a custom wrapper around the function. Each wrapper receives arguments encoded in our system's representation, converts all the arguments to the corresponding native representations, and calls the native function to perform system interaction. Returned values are also converted back to our representation. In addition, the wrappers are responsible for keeping the memory safety of the whole of our system and so on: before invoking a native function, they check boundary preconditions such as the length of an I/O operation and buffer lengths. We have implemented a preliminary wrapper generator to reduce some of the cumbersome work involved in implementing such wrappers.

As an implementation choice, we used the same technique for several complex user-level C library functions that interact with operating systems, such as file I/O (`stdio` functions), locales, and `setjmp`. When such functions are implemented, some native pointers, such as `FILE *` pointers, must be stored in data structures accessible from user programs. To prevent such pointers from being exploited, they are wrapped as a special memory block and associated with special access methods that deny all memory accesses.

4. Current status

Here, we briefly report on the current status of our implementation. We have implemented the whole proposed system. The current compiler accepts most features (including all essential features) of the ANSI-C specification on the Intel x86 architecture. Ports to other 32-bit architectures do not seem very hard, and 64-bit support is also theoretically possible (although many runtime routines must be rewritten accordingly).

Fail-Safe C system consists of the compiler and linker (written in Objective Caml), runtime and standard C libraries (written in our C compiler itself and in native C with a custom wrapper helper language), and some support scripts (in Perl). The compiler part was written as a source-to-source C translator; in other words, we used the native C compiler (Gnu C compiler) as a backend "rich assembler". The current implementation does little program analysis; but some local code optimizations such as redundant value elimination are implemented. Our system supports the same command-line syntax as conventional C compilers. Many library functions from ANSI C and POSIX are already implemented and are contained as part of the system.

The system has been published on our website [17] as open-source software since April 2008. We provide both a source distribution and an unofficial binary package for Debian GNU/Linux.

[4] This varargs block is heap-allocated because an address of this block may leak if the called function uses a `va_list` value returned from the `va_start` macro inappropriately.

Fail-Safe C can compile various server programs in real use on the Internet. The currently supported programs include:

- OpenSSL version 0.9.8j, the most widely used open-source cryptographic library and tools
- OpenSSH version 5.1p1, the most widely used remote secure login software
- ISC BIND (Berkley Internet Name Domain) server version 9.4.2, the most widely used Internet domain name server,
- thttpd version 2.25b, a compact web server,
- qmail version 1.03, a famous Internet mail server,
- postfix version 2.5.5, another recently famous Internet mail server,
- libtiff version 3.8.2 and libpng version 1.2.34, libraries for managing image files,
- zlib version 1.2.3, a file compression library.

During compilation, almost all of the main program code was left unchanged; most of the modifications required to accept those programs are fixes for small bugs in the programs themselves to adhere the stricter type and declaration checking of the Fail-Safe C. These modifications will be available on our website as patch files. This is very unlike any existing methods that provide complete memory safety: even CCured requires the modification of up to 1% of the original source code. We changed build procedures such as `Makefile`s accordingly.

During the implementation, we found a small off-by-one bug in thttpd. Luckily, although this bug was unsafe, it is almost never exploitable. We reported it to the original author. We have also tested our system against several real vulnerabilities in existing programs such as older versions of Sendmail and xv by extracting some of the buggy code as a test program, and we verified that our scheme can correctly handle and detect such memory corruption attempts.

5. Evaluation

We have applied Fail-Safe C to a few benchmark programs to evaluate the performance of the current implementation. The programs that we used were:

- ByteMARK benchmark version 2,
- the "Speed" subcommand available in OpenSSL.

We also include the results of three small micro-benchmarks here for evaluating the performance of primitive operations.

The performance of the current implementation is shown in Figure 10. The values in the figure are normalized to give the speed index of the program compiled by the native compiler the value 1.0 (smaller is better). We used a computer with dual-core AMD Opteron processors running at 2.4 GHz and a GNU C Compiler version 4.1.2 distributed with Debian GNU/Linux 4.0 (release etch). The points marked by ◇ show the performance of the assembly routines implemented in OpenSSL, for reference.

The micro-benchmark "micro fib 44" calculates an element of the Fibonacci sequence by simple recursion. It shows that there is little performance overhead (∼2%) on the handling of fat integers. The two tests "micro sumup", which sums up an array of random integers, and "micro qsort", which sorts an array of random integers, show performance on memory operations. They show that memory accesses with fat integers take about 1.7 to 2.2 times as long to perform as one compiled by a native compiler. However, when we force access methods to be used for all memory accesses (shown as "CAST"), the overhead grows to more than 10 times. This shows importance of cast flags and short-cut memory accesses in Fail-Safe C.

In the OpenSSL RSA tests, the performance depended on the length of the public key: in the range conventionally used (1024 to 4096), the program compiled by our compiler took about two to four times as long to run as one compiled by the conventional compiler. For AES tests, unfortunately, the slowdown was about five times. The reason for this is not yet completely determined, but we guess that the programming style of the OpenSSL implementation (implemented in a kind of object-oriented programming style in C) involves moderate use of cast pointers.

The Bytemark benchmark shows that performance varies greatly depending on the way that programs are written. In the Fourier test, our compiler imposed almost no performance penalty. The worst test was emfloat, which showed performance results about six times as long. The reason for this seems to be a limitation of the current implementation, which detects only some of the local variables that can be safely allocated in the stack: the emfloat test allocates many local structs whose addresses are passed around among several functions. Current function-local analysis does not find such variables as stack-allocatable. In future, we will implement a better analysis to solve this problem, which should work well with separate compilation.

On average, we can see that safety checks performed by our system increased the execution time of computation-bound programs by around two to four times in the ByteMARK test. For server programs with much heavier I/O overhead than these benchmarks, we would expect slightly better performance than indicated by these evaluation results.

6. Related Work

Canary-word techniques The "canary word" technique, which is a well-known way to avoid simple kinds of sequential-access buffer overflows, has been implemented for a long time. Protection on stack buffers was provided by StackGuard [6]. Recent versions of the Microsoft Visual C compiler include the `/GS` compile option, which has a similar function on the Windows operating system platform. The recent GNU C compiler also implements a similar buffer smashing protection (`-fstack-protector` option) based on ProPolice [7]. The benefits of the canary-based technique are its low overhead and high compatibility with existing systems. These systems modify only the structure of stack frames and the unreferenced area between global variables, both of which are not usually accessed directly by user programs. However, the limitation of this approach is also obvious: it can only prevent sequential-access buffer overflows that are used to directly attack execution-controlling data; it cannot prevent even a buffer overflow based on random access. If the execution-controlling data is overwritten directly without modifying the canary words (e.g., by random-access overflow and other exploits), the system is ineffective. For example, both StackGuard and GCC stack smashing protector are unable to prevent a security exploit found in the option parsing routine in the old Sendmail.

Safe memory management for C There have been some proposals for safe memory management of C programs. Loginov et al. [15] proposed a method of ensuring pointer safety by adding a 4-bit tag to every octet in the working memory. "Backward-compatible bounds checking" by Jones and Kelly [12] modifies the GNU C compiler (gcc) by inserting bounds-checking code that uses a table of live objects. Safe-C [2] can detect all errors caused by early deallocation of memory regions. However, as far as we know, these proposals are not only slower than our method, but also incomplete. They seem to have limitations regarding the source and

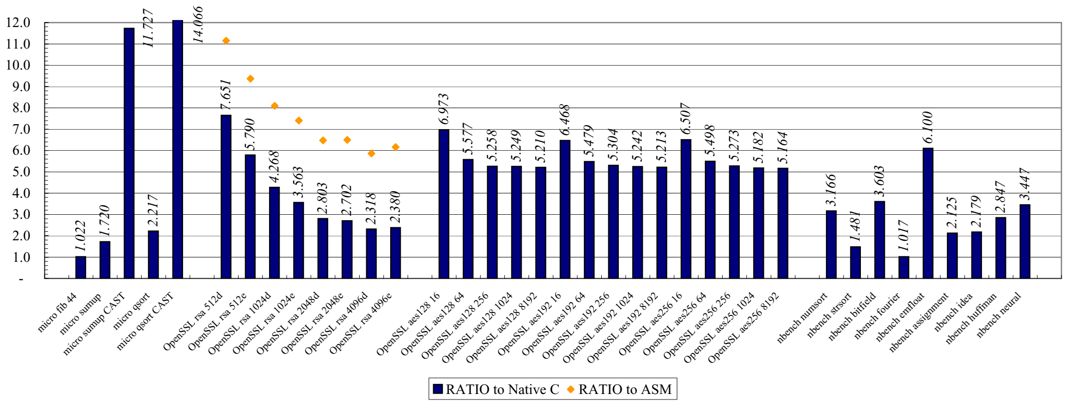


Figure 10. Performance of the current compiler.

destination types of cast operations or do not protect pointers stored in memory overwritten by integers via a cast.

Safe languages There are already plenty of languages (both statically and dynamically typed) that ensure complete memory safety. Some of these, such as ML and Lisp, are accepted by some programmers for writing programs in a memory-safe way. However, although these languages are good for new programs, it is hard to reuse existing C programs on those systems. The syntax of Java seems to have been designed intentionally to be similar to C, for acceptance in the real world. Thanks to this, porting C programs to Java is a little bit easier than porting them to other languages, but it still requires heavy rewriting of the code.

Some other safe imperative languages resemble C more closely. For example, Cyclone [9, 11] is designed to ease the porting of C programs so that they become type safe. For common C programs to conform to Cyclone, however, about 10% of the program code must still be rewritten [9, 11], which is a considerable task.

CCured Necula et al. have designed and implemented CCured [16, 5], a sound type system that can support C programs that include cast operations. The approach of CCured is to analyze the entire program and then split it into two parts: the "type-safe part" that does not use cast operations and the "type-unsafe part" that could be contaminated by cast operations. However, to the best of our knowledge, the designers did not focus on perfect source-level compatibility with existing programs, and the system does in fact support only a subset of the ANSI-C semantics. The reported amount of code that must be rewritten is less than 1% of the source code, which is much smaller than for other proposals such as Cyclone, but still a significant amount. Our work was designed with a greater focus on complete compatibility with the ANSI-C specification and on the highest possible compatibility with existing programs.

The main technical difference between CCured and our proposal is that CCured is based mainly on static analysis of cast operations, while ours uses dynamic handling as its main tool. CCured statically determines which variables might have a cast pointer, and "quarantines" the wild part from the pure part of the program. The pure part will then behave almost like a program of a pure statically typed language; e.g., there will be no type information inside. The weakness of this method is that the system cannot allow any pointers in the wild part to point to values in the pure part. In addition, as value types are completely determined statically, a pointer that could point to wild values must always point to wild values. This means that wild pointers have a "pollution" problem: if one pointer in a variable is found to have a cast in some case, all the data structures that might be pointed to by the same variable, and even all the data structures that could be indirectly traversed from the variable, must be in the wild part. Thus, the relative size of the program's wild part is likely to increase with program size.

In our scheme, on the other hand, a cast pointer does not infect any other data because each memory block has a representation type: even if there is a cast pointer pointing to a memory block, the pointers in that block can still be typed (not cast).

Another problem with CCured is conflicts between the system library and the pointer-type pollution described above. As system libraries are compiled beforehand, a library must have a single static type. However, in CCured, one wild pointer may pollute other values by forcing them to be the wild type, including data to be passed to the system library. If a library has already been compiled as a non-wild type, the program cannot be linked safely. This makes it harder to compile the large programs used in the real world.

Extensions to our work Kamijima and Sumii [13] have implemented a C-to-Java translator which supports pointer arithmetic and arbitrary pointer cast based on our scheme. They have introduced local static to reduce additional overhead imposed by the representation of Fail-Safe C's data structure on Java.

Furuse proposed VITC [8], an extension of our scheme with analysis and enforcement of information flow restriction. As our compiler enforces basic memory safety and runtime conformity to the defined language semantics, it can be combined with various static/dynamic analysis to ensure stronger safety/security properties on the C language.

7. Conclusion

We have designed a completely memory-safe implementation of the full ANSI-C language that can support all of the features of ANSI-C, including casts and unions. We have introduced several techniques to support C language features that most safe languages do not have. We also exploited several implementation tricks to reduce the runtime overhead as much as possible.

We have implemented both the compiler system and the runtime library, which contains over 500 standard library functions defined in the ANSI-C and POSIX specifications. The system accepts many existing, well-used server programs such as OpenSSL, OpenSSH, and BIND9.

A performance evaluation showed that, on average, the safe programs compiled by Fail-Safe C take two to four times as long to compute as unsafe programs compiled by native compilers.

7.1 Future work

Compatibility with more existing programs: Due to the nature of the memory-safety provided by our system, the current compiler does not directly support programs that use custom memory management, including Apache httpd and many implementations of programming languages. Such programs may either run very slowly (due to the heavy use of cast pointers and access methods) or simply not work correctly (if integer arithmetics are used in pointers).

We are considering implementing a compatibility library that supports an interface for several custom memory managements routines, e.g., the Apache APR and Boehm-Weiser's GC library. We expect the introduction of such libraries to enable our system to support a broader range of existing programs.

Optimization: As an instant safety measure for existing programs, we think that the current performance is acceptable but not completely satisfactory. We will investigate several compiler optimization techniques to further reduce runtime overheads. We observed that programs written in some safe languages with sophisticated implementations (such as Java and Objective Caml), usually ran at about (very roughly) half the speed of heavily tuned C programs. We have currently set this value as a future goal for our optimization efforts.

However, we must be aware that not all existing optimizations are directly applicable to our system. Needless to say, the optimizations to be applied must be sound. Moreover, many optimizations assume that if the boundary checks fail, the program execution does not continue, (or at least that the current scope of execution is terminated e.g., by escaping via exceptions). This not always true in our system: as can be seen in Figure 6, there is a control path that leads from the "failed" branch of a boundary check and merges back to the main path. We intend to implement optimizations carefully without sacrificing any safety.

Extensions: An extension of our compiler to accept programs written in the 1999 revision of C (C99, ISO/IEC 9899 : 1999) will not be difficult: however, C99 defines that a byte-to-byte copy of a valid pointer will become a valid pointer, which is not supported in current Fail-Safe C. We think that this will not cause any critical incompatibility to any existing programs, since this definition might be intended to ensure that pointers copied by `memcpy` will be valid pointers, which *is* supported in Fail-Safe C.

To support C++ language, we have designed an extension to the cast flags of Fail-Safe C to support class inheritance, nominal subtype relations on pointers and virtual method invocations [18]. The extended scheme uses 2-bit cast flags to distinguish uncast pointers (which allow pointer arithemtic), and pointers to subtype objects (which do not allow pointer arithmetic), from arbitrary-cast pointers. However, supporting full C++ language will require huge amount of implementation efforts on various language features unrelated to memory safety (such as method and operator overloading, templates, and huge standard libraries).

Acknowledgments

The author is grateful to Dr. Eijiro Sumii, Prof. Akinori Yonezawa and Prof. Hideki Imai for valuable advices and supports on this project. The author also thanks anonymous referees for valuable and encouraging comments.

Implementation of a part of Fail-Safe C runtime system and standard library is performed jointly with Lepidum, Inc. under the project "New-generation Security R & D Program" from the Ministry of Economy, Trade and Industry (METI) in Japan.

References

[1] American National Standard Institute. American national standard for information systems — programming language – C. ANSI X3.159-1989.

[2] 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.

[3] Hans Boehm. A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[4] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software: Practice & Experience*, pages 807–820, September 1988.

[5] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real workd. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 232–244, June 2003.

[6] 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.

[7] Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stack-smashing attack detection. *IPSJ SIG Notes*, 2001(75):181–188, 2001.

[8] Jun Furuse. VITC: Safe C code compilation against attacks. In *4th Workshop on Dependable Software*, 2006. In Japanese.

[9] 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.

[10] International Organization for Standards and International Electrotechnical Commission. Programming languages — C. ISO/IEC Standard ISO/IEC 9899:1990.

[11] 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.

[12] 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.

[13] Yuhki Kamijima and Eijiro Sumii. Safe implementation of C pointer arithmetics by translation to Java. *JSSST*, 26(1):139–154, 2009. In japanese.

[14] Brian W. Kernighan and Dennis M. Ritchie. *The Programming Language C.* Prentice Hall, second edition, 1988.

[15] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. *Lecture Notes in Computer Science*, 2029:217–, 2001.

[16] 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.

[17] Yutaka Oiwa. Fail-Safe C webpage. `https://staff.aist.go.jp/y.oiwa/FailSafeC/index-en.html`.

[18] Yutaka Oiwa. An extension to Fail-Safe C to support object-oriented languages. In *Symposium on Programming and Programming Languages*, March 2005.

[19] Yutaka Oiwa. *Implementation of a Fail-Safe ANSI C Compiler.* PhD thesis, University of Tokyo, 2005.

[20] Yutaka Oiwa. Type-safe linking of C programs. In *Symposium on Programming and Programming Languages*, March 2007.

[21] Gray Watson. Dmalloc – debug malloc library. `http://www.dmalloc.com/`.