Implementation of a Fail-Safe ANSI C Compiler
安全な ANSI C コンパイラの実装手法

Doctoral Dissertation
博 士 論 文

Yutaka Oiwa
大 岩　寛

Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo on December 16, 2004
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Abstract

Programs written in the C language 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, and this has become a problem affecting society as a whole. The root of these errors is usually corruption of on-memory data structures caused by out-of-bound array accesses. The C language does not provide any protection against such out-of-bound access, although recent languages such as Java, C#, Lisp and ML provide such protection. Nevertheless, the C language itself should not be blamed for this shortcoming—it was designed to provide a replacement for assembly languages (i.e., to provide flexible direct memory access through a light-weight high-level language). In other words, lack of array boundary protection is "by design." In addition, the C language was designed more than thirty years ago when there was not enough computer power to perform a memory boundary check for every memory access. The real problem is the use of the C language for current casual programming, which does not usually require such direct memory accesses. We cannot realistically discard the C language right away, though, because there are many legacy programs written in the C language and many legacy *programmers* accustomed to the C language and its programming style.

To alleviate this dilemma, many approaches to safe implementation of the C language have been proposed and put into use. To my knowledge, however, none of these support all the features of the ANSI C standard *and* prevent all unsafe operations. Some, such as StackGuard by Cowan, perform an ad hoc runtime check which can detect only specific kinds of error. Others, such as Safe C, accept only a small subset of the ANSI C standard. CCured, by Necula, comes closest to providing a solution in my opinion, but is not yet perfect.

This thesis proposes the most powerful solution to this problem so far. *Fail-Safe C* is a memory-safe implementation of the *full* ANSI C language. More precisely, it detects and disallows all unsafe operations, yet conforms to the full ANSI C standard (including casts and unions) and even supports many of the "dirty tricks" common in many existing programs which do not strictly conform to the standard. In this work, I also propose several techniques—regarding both compile-time and runtime—to reduce the overhead of runtime checks. By using the Fail-Safe C compiler, programmers can easily make their programs safe without heavy rewriting or porting of their code. In the thesis, I also discuss a demonstration of

how exploitation of existing security holes in well-known programs can be prevented.

The key ideas underlying Fail-Safe C are

1. a special memory block representation which supports run-time checking of block boundaries and types,

2. object-oriented representations of memory blocks with access handler methods associated with each block; these support safe execution of untyped operations such as pointer casts,

3. a special notion of memory addressing, called *virtual offset*, which contributes to the safety of cast operations and solves compatibility issues for many legacy programs,

4. a sophisticated representation of pointers (and integers), which records whether a pointer was cast, to manage both the safety of cast operations and the efficiency of normal pointer operations.

Whenever values in a program are used as a pointer to access memory data (except when the Fail-Safe C compiler deduces that it is safe to omit the checks), these values are checked against the boundary and type information kept in the referred memory block. If the pointer refers to memory beyond the block boundary, a run-time error is signaled and the program execution is safely stopped. If the type of the pointer conflicts with the type of the referred block, the memory access is processed via access handler methods to maintain the safety of the program execution. Otherwise, the memory block is accessed directly to ensure high execution performance. The cast information on the pointers is carefully maintained by the compiler to accelerate the type check of the pointers. In addition, the virtual offset notion hides all tricks from the running program; programs will find no differences between the usual compiler and the Fail-Safe C compiler, except that the program is immediately killed when an unsafe event occurs. This makes it possible to run many programs which include safe “dirty-tricks” without modifying their source code, and ensures the safety of such programs.

論文概要

C 言語で書かれたプログラムは、迷子ポインタやバッファ溢れなどによる厄介なバグの影響を受けがちであることはよく知られている。とりわけ、インターネット上のサーバプログラムにおけるそのようなバグは、悪意の攻撃者によってシステム全体を乗っ取るための攻撃の対象となりがちで、最近では社会的な問題にすらなっている。このような厄介なバグは元をたどれば、メモリ上の配列の境界を越えたアクセスにより、データ構造が破壊されることである。最近の言語、例えば Java、C#、Lisp、ML などの言語はこのような境界を越えたアクセスに対して保護機構を用意しているが、C 言語にはそのような機構はない。しかし、これは C 言語のデザイン上の欠陥とは言えない。なぜなら、C 言語は元々アセンブラ言語の置き換えとして、つまりは柔軟で直接的なメモリ操作を高級言語で記述するためにデザインされたものだからである。言い替えれば、このような保護機構の欠如は「わざと」導入されたものである。また、C 言語がデザインされた 30 年前には、当時の計算機能力に対して、このような保護機構を導入するのが現実的でなかったという点もある。過ちとされるべきはむしろ、そのような C 言語を現代の日常のプログラミング言語として、実際には直接的なメモリアクセスが必要とされない場合にも用いていることにある。けれども今日において、C 言語を直ちに放棄してしまうことは現実的ではない。C 言語で書かれた既存のプログラムは多く存在し、また C 言語やそのプログラミングスタイルに慣れ親しんだ「既存のプログラマ」も数多いからである。

このようなジレンマを解決するために、C 言語を安全に実装する多くの試みが提案され実際に実装されてきた。しかし、我々の知る限りそれらのすべては、危険な操作の全てを拒否し、**同時に**全ての ANSI C のプログラムを処理できるという目標を達成していない。Cowan による StackGuard に代表される実装のグループは、場当たり的な検査手法でプログラムに出現する特定の

形の誤りを検出するだけのものであるし、他方 SafeC に代表されるグループは、C 言語の仕様の一部分のみを入力として受け付けるものである。Necula によって提案されている CCured が、我々の知る限りでは現時点でもっとも目標に近いものであるが、これも完璧であるとはいえない。

本論文は、この問題に対するもっとも強力な解を提案する。本論文で述べられている Fail-Safe C は、メモリ安全な ANSI C の完全な実装である。この実装は、全ての危険な操作を禁止しつつ、キャストや共用体を含む全ての ANSI C 標準に準拠し、かつ ANSI C の範囲を越えたプログラムに頻出するいわゆる「汚いトリック」の多くをも許容する。同時に、本実装は、コンパイル時と実行時双方で行なわれるさまざまな最適化によって、実行時検査の負荷の削減をはかっている。Fail-Safe C コンパイラを用いることで、プログラマは簡単に、自らの書いたプログラムに変更を加えることなしに、また移植作業をすることなしに、安全に実行することが可能となる。論文中では、実在する有名なプログラムに存在するセキュリティー上の脆弱性を用いて、実際に Fail-Safe C を適用して安全性を保証する実験を例示している。

この論文で述べられているいくつかの重要なアイディアは以下の通りである。

1. メモリブロックの特殊な表現により、動的な境界検査と型検査を実現すること、

2. オブジェクト指向の概念を用いてメモリブロックを表現し、全てのメモリブロックにアクセスメソッドを付加することにより、ポインタのキャストなどの静的型によらないアクセスの安全な実行をサポートすること、

3. 「virtual offset」と名付けたメモリのアドレスづけの特殊な方法により、既存のプログラムの互換性の向上とキャスト操作の安全性を同時に実現していること、

4. そして、ポインタがキャストされているかどうかを自らに記録するような、ポインタ (と整数) の賢い表現により、安全にキャストを実装すると同時に通常のポインタの高速な使用を実現したこと。

Fail-Safe C の環境下では、プログラム中の値がポインタとして参照に用いられるたびに、参照先ブロックのサイズと型との整合性を検査される (コンパイ

ラが検査を省いても安全であることを確実に判定できた場合を除く)。ポインタが参照先ブロックのサイズを超過したメモリを参照している場合、実行時エラーが報告されプログラムは直ちに停止される。ポインタの型と参照先ブロックの型が整合しない場合は、アクセスハンドラメソッドが参照に用いられ、プログラムの実行の安全性を保証する。どちらでもない場合は、プログラムが直接メモリを参照することで、高速な実行を実現する。ポインタがキャストされたか否かの情報は、コンパイラによって正確に維持され、ポインタの型整合の判定を高速に行なえるようにしている。また、virtual offset の概念は、先に述べた一連の動作をプログラムから隠し、「舞台裏でこっそり行なわれるもの」にする。つまり、実行中のプログラムは、Fail-Safe C の監視下で実行されているということを認知することは、安全でないプログラムが突然終了させられることを除いてはできない。このことは、さまざまな「汚いトリック」を用いたプログラムがそのままプログラムを変更せずに動かせることを可能にし、また同時にそのようなプログラムが安全に動作することを示唆している。

Acknowledgements

I express my deepest gratitude to Dr. Eijiro Sumii, one of the best friends and research partners one could hope to have. His sharp but constructive suggestions have made the design of the Fail-Safe C system very solid regarding both the theoretical aspects and the implementation details.

I am very thankful to Dr. Tatsurou Sekiguchi for sharing his very deep knowledge regarding compiler construction techniques. He is without question most knowledgeable of my partners regarding conventional compilers, and he has been contributed greatly to the design and implementation of the generic part of my compiler, such as the handling of the intermediate representation of programs and various internal transformations.

I am deeply grateful to my thesis supervisor Professor Akinori Yonezawa for his continuous strong support in this research. He has provided me with many great opportunities for presenting this work to top-level researchers and discussing it with them.

I thank Profs. Naoki Kobayashi, Kenjiro Taura, and Hidehiko Masuhara, for both valuable technical suggestions but also for invaluable support during the difficult points of my research life. Without their continuous encouragement, I might not have been able to continue my efforts to complete this work.

I am also thankful for various suggestions given to me by Prof. George Necula, Prof. Benjamin Pierce, Dr. Yoshihiro Oyama, Mr. Toshiyuki Maeda, Prof. Ken Wakita, Dr. Akira Tanaka, Mr. Norifumi Gotoh and many others.

Finally, I express my heartfelt appreciation to my parents for supporting and encouraging me throughout my research endeavors.

Part of this research has been supported by research fellowships of the Japan Society for the Promotion of Science for Young Scientists.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

This thesis describes a method for safe execution of C programs which can be applied to all programs written in conformity with the ANSI C specification [33, 2, 38].

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, the C language 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, the C language 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 the C language have frequently been used to perform complex pointer manipulations very frequently. This has created serious security flaws. In particular, destroying on-memory data structures through 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 hijack the behavior of programs containing such bugs. Most of recently reported security holes have been due to such misbehavior.

To resolve the current situation, I have developed a special implementation of the ANSI C language, called *Fail-Safe C*, which prevents all of the dangerous memory operations that lead to execution hijacking. The Fail-Safe C 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 will lead to memory corruption, the runtime system of the Fail-Safe C system cooperates with inserted codes to report the error and terminate program execution. Use of the Fail-Safe C system instead of the usual C compilers thus

enables safe execution of existing C programs.

1.2 Design goals

The design goals set for Fail-Safe C were as follows.

(1) Complete safety protection

A program compiled with Fail-Safe C should never be affected by any memory errors. In other words, the program should run only in the way the program is written. This may seem an obvious requirement that hardly bears mentioning. However, many security holes allow exploitation where outside program code is injected into programs instructing them to execute themselves in a way contrary to how they were originally written.

Most of the previous research has aimed at preventing exploitation of only certain subsets of the existing security holes. This has been only a partial security solution, because if the proposed systems are applied to the majority of running systems, attackers (who are motivated by several external incentive such as a desire for money, information, and so on) will simply begin to exploit other kinds of security holes which these systems cannot block. In contrast, Fail-Safe C provides complete protection against exploitation based on memory corruption, which includes sequential buffer overflow as well as general memory boundary overflow, double-deallocation, misuse of cast operations, and all other possibilities. A Fail-Safe C user can expect the same level of security as would be the case for a program written in Java or ML while being able to continue using C language.

(2) Full conformance to the ANSI-C specification

There are already plenty of safe languages with which secure programs. Some of these—for example, ML, Lisp, or Haskell—use syntaxes and philosophies completely different from imperative languages, while others, like Java, use syntaxes that slightly resembles that of C languages. There are also several languages designed to be similar to the C language to make porting existing C programs to those languages easier. Moreover, there are several safe implementations for the proper subset of the C language. as I have personally experienced, porting from C languages with mosr of these systems still requires a considerable effort. The amount of the modifications required to port existing C programs varies among the languages, but the fact remains that these languages did not successfully replace programs written in the C language.

To overcome this problem, Fail-Safe C was designed to accept *unmodified* C programs as input. Since it is difficult to define what C language programmers expect, I used the official ISO/ANSI specification for the C language [33, 2], often called ANSI-C or the second edition of Kernighan-Ritchie book [38], as the

reference point in the first stage. Full-support of ANSI-C implies several complicating matters: support is necessary for a very wide set of cast operations between pointer types, *bi*directional casting between pointers and integers (including in the left direction!), a variable number of arguments (varargs), and so on. It is tough to comply to this specification while still providing a keeping 100% safety guarantee.

(3) Possible support for many existing techniques

The above suggests that ANSI C is too permissive. At the same time, ANSI C is so restrictive that most existing programs do not strictly comply with the ANSI C specification. Actual programs written in C language assume many more properties than those specified in the ANSI-C specification. For example, many programs expect that the pointer of different types to be interchangeably usable in many contexts without fear of representation incompatibilities. Moreover, it is often assumed that the pointers to functions receiving different types of pointers will be compatible. This kind of cast function pointer often appears in an argument of higher-order functions like `qsort` (See Figure 1.1 for an example). Another instance of techniques beyond the ANSI-C specification is a technique to implement variable-sized structures (Figure 1.2). This technique assumes that the memory space is "flat" in some sense and that the memory area allocated by `malloc` and other functions can be used in any form the programmer chooses. It is not always possible to support all techniques used in existing programs, but, supporting only strictly ANSI-C compliant programs is likely to be insufficient.

(4) Lowest possible execution overhead

Provided that all three of the above requirements are satisfied, the execution performance should be as good as possible. The implementation of the Fail-Safe C system combines several existing implementation techniques for both dynamically-typed languages and statically-typed languages, and enhances and extends these techniques with several new implementation tricks to enable the best possible execution performance.

In particular, the much of the design effort was aimed at providing support for cast operations and other type-unsafe operations without sacrificing the execution performance of type-safe operations. The implementation of the type-safe portion of operations was designed to be very similar to that of strongly- and statically-typed languages.

1.3 Very brief introduction to the Fail-Safe C system

Briefly, the key concepts of the Fail-Safe C system are as follows.

- Introduce size-managed, *typed memory blocks* to support reliable detection of boundary overflows at runtime. Each memory blocks appears as a portion

The following example is taken from the source code of the Apache web server (version 1.3.9).

An excerpt from `src/modules/standard/mod_autoindex.c`:

```
/*
 * Compare two file entries according to the sort criteria.  The return
 * is essentially a signum function value.
 */

static int dsortf(struct ent **e1, struct ent **e2)
{
    ... /* compare directory entries pointed by e1 and e2 */
}

static int index_directory(request_rec *r,
                           autoindex_config_rec *autoindex_conf)
{
    ...
        qsort((void *) ar, num_ent, sizeof(struct ent *),
              (int (*)(const void *, const void *)) dsortf);
    ...
}
```

The type of the `qsort` function in the standard library is the following:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

A pointer to the function `dsortf`, which have a type different from the required type, is cast and then passed as a fourth argument to `qsort`.

Figure 1.1: An example of function pointer casts.

The following example is taken from the source code of the GNU privacy guard (gnupg, version 1.0.1), a program which encrypts and signs digital contents.

A type definition in `g10/packet.h`:

```
typedef struct {
    byte version;
    byte cipher_algo;    /* cipher algorithm used */
    STRING2KEY s2k;
    byte seskeylen;   /* keylength in byte or 0 for no seskey */
    byte seskey[1];
} PKT_symkey_enc;
```

An excerpt of the function `parse_symkeyenc` in `g10/parse-packet.c`:

```
static int
parse_symkeyenc( IOBUF inp, int pkttype, unsigned long pktlen, PACKET *packet )
{
    PKT_symkey_enc *k;
    ...
    seskeylen = pktlen - minlen;
    k = packet->pkt.symkey_enc = m_alloc_clear( sizeof *packet->pkt.symkey_enc
                                                + seskeylen - 1 );
    k->version = version;
    k->cipher_algo = cipher_algo;
    k->s2k.mode = s2kmode;
    k->s2k.hash_algo = hash_algo;
    if( s2kmode == 1 || s2kmode == 3 ) {
        for(i=0; i < 8 && pktlen; i++, pktlen-- )
            k->s2k.salt[i] = iobuf_get_noeof(inp);
    }
    if( s2kmode == 3 ) {
        k->s2k.count = iobuf_get(inp); pktlen--;
    }
    k->seskeylen = seskeylen;
    for(i=0; i < seskeylen && pktlen; i++, pktlen-- )
        k->seskey[i] = iobuf_get_noeof(inp);
    ...
}
```

The array field `seskey` only have one byte in the declaration. However, the argument to `m_alloc_clear` specifies `seskeylen-1` additional bytes to allocate, and the elements of the `seskey` field up to (`seskeylen-1`)-th element is used to store session keys.

Figure 1.2: An example of a variable-sized structure technique.

of the usual flat memory space to user programs, but internally manages various forms of additional information to manage safety conditions.

- Represent every pointer as a pair consisting of a base and an offset, to support pointer arithmetic (*fat pointers*). Integers are also represented in two words for ANSI-C compatibility. These values also appear to user programs to be the one-word values.

- Attach a set of methods which perform basic read/write operations for every memory block (*access methods*). In other words, memory blocks are abstracted in the sense of object-oriented design. This enables the use of different internal representations for each block, while still enabling compatibility (or cast support).

- Reduce the overhead introduced through above abstraction by directly accessing block contents via pointers when the pointer is not cast. To achieve this, a one-bit flag is appended to every pointer to record whether the pointer is cast (*cast flags*).

The first two concepts mainly contribute to basic safety and compatibility. As every pointer contains a base part apart from the pointer arithmetic, the boundary of referred memory blocks can be checked however the offset is altered. Memory blocks hold the two-word fat pointers and integers, but still "pretend" to user programs that they are holding the usual one-word values. This pretense implies an internal translation of the offsets in memory blocks, because the change in representation alters the size of objects in blocks. This translation is formalized as a concept of *virtual offsets*.

The third and the fourth concept contribute to performance optimization. To satisfy the fourth goal given in the previous section (especially that there be little additional overhead for cast-free programs), it is desirable to use various memory block representations designed for each specific type in the programs. Access methods enable such heterogeneous representation of memory blocks while preserving compatibility, and cast flags enable the efficient implementation of cast-free memory operations.

Details will be given in Chapter 3 (and in Appendix A).

1.4 Clalifications: matters not handled by Fail-Safe C

Although Fail-Safe C is a powerful solution to security problems, it does not solve all types of safety problems, for obvious reasons. For example, if a program *intentionally* sends user passwords to a third party, the compiler has no way to prevent this. The intended purpose of the Fail-Safe C system is clarified in the following.

1. The definition of *fail-safety*

If a program intentionally dereferences a NULL pointer, it is impossible to define any meaningful "correct" behavior for the program, except to abandon an execution. Fail-Safe C does not and cannot provide a system which does not fail—instead, it provides a system which always remains *safe* even when programs *fail*. Under Fail-Safe C, when a memory-related security attack has been launched, the program is halt. It may suspend an important network service or commercial transaction, it may abort a transaction, or it may require a human intervention for the recovery of the whole system. However, Fail-Safe C does not allow attackers to hijack the execution of programs, does not allow embedding of a rootkit (which can be used for further invasion such as the creation of backdoors, or to read of eavesdrop on confidential data) via buffer overrun. In most Internet server programs, users of Fail-Safe C system can resume a service by simply re-booting the processes, without fear of severe sustained damage. Alternatively, a typical fault-tolerant system may save all of the services, with protection against invasions provided by the Fail-Safe C system.[1]

2. Security holes without memory corruptions

   Although the majority of security attacks are based on memory corruption, there are other instances of security holes. One example is an incorrect sanitizing of certain special characters in user inputs. For example, if a program running with some privileges passes a user-inputted string to Unix shells without sanitizing, attackers can gain access to the system resources by embedding some of the shell's special characters (such as >, <, ; or |). Many similar instances are found in a huge number of web programs, such as incorrect handling of URL-encoded strings or cross-site scripting problems.

   These problems, which are bugs based on the correct behavior of programs, cannot be dealt by Fail-Safe C. The program compiled by Fail-Safe C runs the algorithm written by the programmer correctly and, faithfully, reproduces the bugs. These are outside of the scope of this thesis.

   There are many proposed methods for analyzing and preventing such bugs. Fail-Safe C can work with these methods, most of which assume some kind of safe language or the safe implementation of languages as their basis. If these methods are directly applied to the C language, the properties which these methods assume are not assured if buffer overflow or other low-level memory corruption occurs. Fail-Safe C can overcome this limitation: if these

[1] The word *fail-safe* is borrowed from the engineering field of critical systems. Some systems have a natural direction for handling in emergency situations that prevents further damage. A fail-safe system is defined as one which may fail, but whose failure always occurs in the direction that does not leads to catastrophic failures. For example, a train signal system that has been designed mechanically to show only red signals in the event of failure is fail-safe. However, an airplane controller that turned off all engines in the event of a failure would be the opposite of a fail-safe system.

methods are (correctly) applied on Fail-Safe C, it can to ensure that the desired safety properties hold completely during the entire program execution.

1.5 Outline

Chapter 1 is this introduction. Chapter 2 discusses various topics regarding recent security holes and related research. Chapters 3 and 4 explain the concept of Fail-Safe C and applied safety management methods. In Chapter 5, some benchmarking results are shown, and some interesting instances of the detection of unsafe program behavior are described. Chapter 6 concludes this dissertation, and discusses possible paths this research may take in the future.

The appendix contains supporting information: Appendix A contains detailed description of the current Fail-Safe C implementation. It also describes additional ways to enhance higher performance and real-world compatibility. Some perspectives for future work in this research area are discussed in Appendix B.

1.6 Term definitions and prerequisites

Throughout this dissertation, the term *word size* refers to the size (the number of bytes in the representation) of the pointers. On some architectures, the size of `int` type might not be equal to the word size. Terms such as *word alignment*, *word boundary*, and so on refer to this word size.

The system assumes the following conditions for the underlying hardware architecture and C language environment used as a back-end code generator. These conditions are satisfied in most modern architectures, including i386-Linux and SPARC-Solaris.

- Signed integer arithmetic is based on two's complement.
- All integer and pointer sizes must be some power of 2.
- The size of one byte must be 8 bits.
- Pointers and `int` type must be at least 32 bits.
- Pointers must be word-aligned. Hardware protection for this restriction is allowed, but not required.
- All pointer types must have the same size and representation.
- There must be an integer type whose size is equal to the word size.
- The natural alignment of integers larger than or equal to the word size must be at least the same as the word alignment.

- At least the word-sized access to the memory must be done atomically.[2]
- Byte order can be either little endian or big endian.
- Memory addressing must be flat in some sense: at least the pointer arithmetic and integer arithmetic must be compatible in the usual sense.

The current implementation does not care about integer and floating number types that are larger than twice the word size (including `long double`). Extending the implementation to support these types is straight forward.

Fail-Safe C has been designed so that it does not depend on any specific setting for word size (especially 32 bits and 64 bits) and alignment requirements as long as the above conditions are met. However, the current implementation still has a non-substantial dependence on the 32-bit architecture in some cases (for example, term selection for field names: byte – half-word – word – double-word for 1, 2, 4, and 8 bytes, respectively). Many of the figures in this dissertation are drawn assuming a 32-bit architecture in either big or little endian byte-ordering to avoid extra complexity. For example, since Figure 3.4 is drawn assuming a big-endian 32-bit architecture, the given values of offsets and padding sizes will differ from those for a 64-bit architecture, or the order of base and offset fields will be swapped in little-endian architectures.

[2] Currently not strictly required, but needed for future support of multi-threading.

Chapter 2

Background

2.1 Typical causes of memory-related security holes

Several kinds of "typical" memory-related program bugs can create exploitable security holes. The following is a list of well-known patterns of vulnerabilities. Of course, the complete set of exploitable vulnerabilities are not limited to what is listed here.

1. Sequential-access buffer overrun

   Bugs of this kind are generally noticeable, appear frequently, and are easily exploited by attackers.

   A long input data sent to a victim program by attackers will be written to an array. If the length of the input exceeds the length expected by the programmer, and if the programmer forgot or failed to check the length properly, the data will not fit into the target array and will flood over it. The overflowed data are then written to the memory area immediately beyond the array. If important data are written in such areas, these data are compromised.

   The simplest (easiest for an attack) cases of buffer overrun are overflows of local variables. If the array being attacked is a local variable in a function, it will be located inside a native stack, and return addresses of the currently-running functions will be stored in the area after such a local variable. If the return address is overwritten by a buffer overflow, the execution does not properly return to the caller of the function, but is transferred to an address arbitrarily chosen by attackers; i.e., the entire execution can be hijacked. Buffers in a dynamically-allocated heap area are slightly harder to use for such exploitation, but there are many known security holes in such buffers; e.g., a security hole found in Sun's implementation of cachefsd [16, 22].

   Unfortunately, managing buffer boundaries properly at all required locations in programs is very tricky in C language. Worse, this kind of error has been ignored for many years, which means exploitable security holes of this type

(An excerpt from `vdcomp.c` in Xv version 3.10a)

```
char                inname[1024],outname[1024];

...

int get_files(host)
int host;
{
  short   shortint;
  typedef long    off_t;

  if (inname[0] == ' ') {
    printf("\nEnter name of file to be decompressed: ");
    gets (inname);
  }
  ...
}
```

Figure 2.1: An example of loose handling of an input buffer using `gets()`

have spread quietly among existing programs. This historical recklessness regarding buffer overflow problems can be seen even in the interface designs of many library routines which have *a priori* problems of this kind (e.g., `gets()` in a standard library). These functions are always vulnerable to a large data input because the interface lacks a maximal allowed input length. (See Figure 2.1 for an example.)

2. Random-access buffer overflow

   This is another kind of buffer overflow, but is slightly more complicated. The target of this attack is an array indexed by some integer values. By crafting exploitable inputs, attackers overwrite a single (or a small number of) word(s) in the memory in victim programs by instructing victims to write to an index outside of the array boundary. If the contents of the overwritten memory are used to control the behavior of programs (e.g., return addresses), the execution will be hijacked. Attack attempts through this kind of flaw are slightly more difficult than those through sequential-overflows, but these attacks more powerful because the overwritten data is not limited to data adjacent to the victim arrays and the bugs are more difficult to find.

   This kind of security holes is often related to the overflow of integers. Careless programmers often forget about the nature of integers in a computer, in that integers have a limited range of values and wraparound to either 0 or a negative value when they exceed the value range. Even if the algorithm of a program is correct under theoretically infinite value range, the program may fail in an actual environment. An example of this kind of bug, found in the

Sendmail mail server, is described in more detail in Section 5.1.1 (Page 55).

Note that this kind of bug is sometimes called an "integer overflow security hole". This is inappropriate: the integer overflow itself is not a security threat at all;[1] in fact, Fail-Safe C as well as the implementations of many other safe languages (e.g., Objective Caml [56] and Java [26]) *do not* prevent integer overflows. The real cause of vulnerability is an inappropriate implementation of boundary checking, which is triggered by integer overflow problems. Thus, it should be called a "buffer overflow vulnerability caused by integer overflow". Fail-Safe C correctly detects this kind of bug.

3. Format-string vulnerability

   A library function `printf` and related functions take an argument encoded to a string which describes both the number and the types of input data as well as the desired output format. The string is usually called a "format string". For example, a string `"%s"` specifies that a string (a pointer to a character array) is expected as an argument, `"%d"` specifies that an integer is expected, and `"%s: %d"` specifies that a string and an integer are expected. The user can implement custom functions taking arguments similar to those functions by using functions like "`vprintf`" and "`vfprintf`".

   The format-string vulnerability is caused by misuse of these functions. If a format string does not contain any conversion specifiers denoted by "`%`", these functions output exactly the same string. Thus, these functions can also be used to output simple fixed messages. For example, the invocation `printf("Hello\n");` works in the same way as the invocation of the simpler function, `fputs(stdout, "Hello\n");`. However, if the strings to be output are externally supplied, this method should not be used (like "`printf(s);`"), but the correct conversion specifiers should be used instead ("`printf("%s", s);`"). If the first form is used, it will misbehave when the string contains the `%` character. As no real arguments corresponding to the conversion specifier are supplied to these functions, these functions will read unexpected memory locations to fetch arguments. In addition, the output size can be made arbitrarily long to cause buffer overflow when functions `sprintf` and `vsprintf` are used. Furthermore, there is a "`%n`" specifier which requests that a number of output characters be *written* to the address specified as an argument, and this can be used for an attack in a way similar to how the random-access buffer overflow is used.

[1] Of course, integer overflow behavior is not intended by programmers in most cases, and is usually the cause of a bug. For debugging purposes, it may be desirable to also prevent integer overflow. Recently the GNU compiler (gcc) optionally detects overflow conditions in integer arithmetic. Fail-Safe C can also be modified to detect such errors, but note that overflow on unsigned integers is defined to be handled on a "modulo upper-bound" basis under ANSI-C specification; thus, it is valid for user programs to utilize such integer overflows behavior (for example, a loop `for (i = 1; i != 0; i <<= 1) {...}` to scan all bits in unsigned integers).

An instance of this kind of security hole has been found in ISC DHCPD, a server program for dynamic configuration of IP addresses in a local LAN [15, 13, 65]. Many security holes of this kind have also been found in several other programs.

4. Early memory deallocation, or deallocation of already deallocated blocks.

   It is a common mistake to use the contents of memory blocks after they have been deallocated by a free() standard library function [24], or to request deallocation of an already deallocated block [14, 17, 62]. Errors of this kind are generally hard to find because the behavior after such errors usually changes greatly depending on the states of the memory management routines, which depend on almost all previous execution statuses. However, many attacks occur for both types of errors. For the first type, an attacker cunningly leads victim programs into allocating a new block in the same memory location as for previously deallocated memory blocks, and into writing an attack data to the location where the victim program thinks another kind of data is stored. An attack exploiting the second type of error is more complicated and difficult, but there is a known exploitation technique (published in a mailing list) which cunningly leads the memory manager in the standard library to misbehave in a predictable way [23].

The recent trend in security attacks seems that attacks exploiting complicated security holes, such as buffer overflow caused by integer overflows or double- deallocation are increasing nowadays, mainly because many simple buffer over- run problems have been identified and solved.

2.2 Existing countermeasures to security holes

As the fear of security vulnerabilities continues to grow, several countermeasures to prevent security compromises have been proposed. In this section, some of these systems are discussed. First three subsections discusses the systems which can be applied to existing C programs. These system prevent a limited kind of security holes, or have some loopholes in security, though. The last three subsections discuss various existing language systems which provide a complete guarantee of memory safety will be discussed. Most of these systems, though, do not support existing C programs.

2.2.1 Buffer-overflow detection using Canary words

The “canary word” technique is a well-known technique to avoid simple kinds of sequential-access buffer overflows.[2] Figure 2.2 illustrates the basic usage of

[2]The name “canary” is taken from the caged canaries once brought into mines by miners to detect poisonous gases or a lack of oxygen. Being more sensitive to such conditions, canaries were affected before the humans, thus giving the miners a chance to escape.

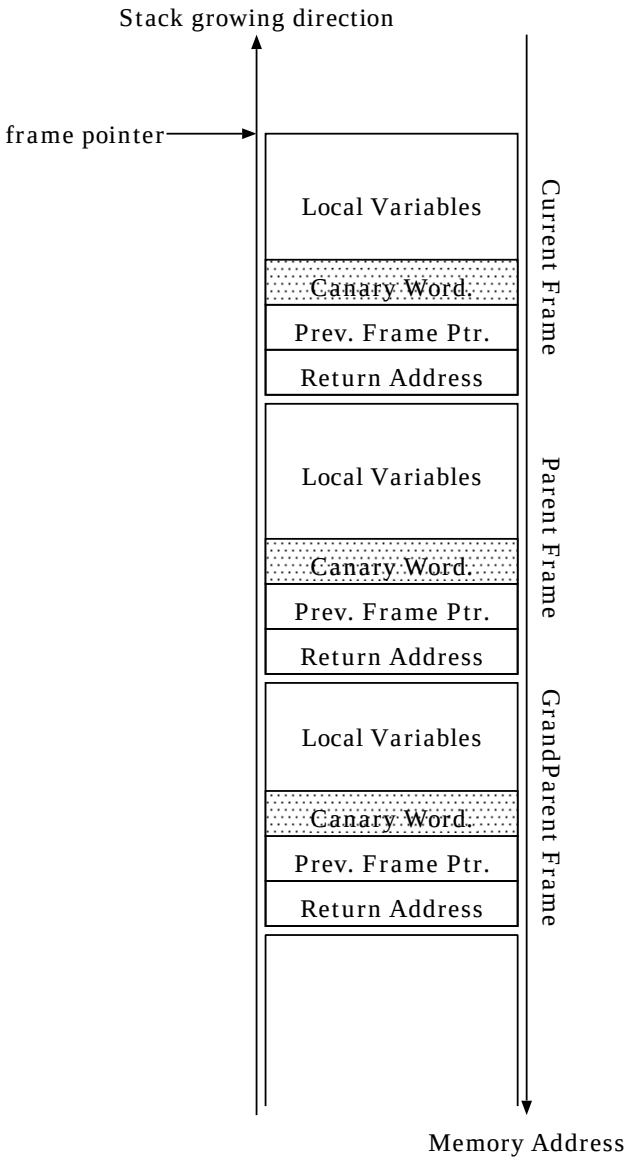


Figure 2.2: Buffer-overrun protection using canary-words

canary-based protections. A randomly-generated integer value, called a canary word, is inserted into the every stack frame between local variables and execution-controlling data such as return addresses and saved frame pointers. If any local variables in the stack suffers from a sequential buffer overflow, and if the important execution-controlling data are affected, the canary word is also overwritten. The epilogue code of each function checks the value of canary words before using the execution-controlling data to transfer execution to its callers. If the canary word is modified, the program execution is halted by the system reporting a buffer overflow condition. The randomness of the canary words is important because if an attacker can guess the original canary value, they can prevent buffer-overflow detection by overwriting the canary with the known original value.

This idea has been implemented for a long time. Protection on stack buffers is provided by StackGuard [20] and many recent implementations. Recent versions of the Microsoft Visual C compiler have includes the `/GS` compile option which has a similar function on the Windows operating system platform [11]. Gray Watson has implemented "dmalloc, debug malloc library" [73], which is a drop-in replacement for memory management routines in the system library that provides canary-based boundary protection for heap-allocated data (along with several forms of debug support for memory problems such as memory leaks).

The benefits of the canary-based technique are its low overhead and high compatibility with existing systems. These systems only modify the structure of stack frames and at the unreferenced area between global variables, both of which are not usually accessed directly by user programs. Furthermore, the runtime cost of introducing canary words is only a few words for each stack frame and up to tens of additional instructions for the prologue and epilogue code of each function.

The limitation of this approach is obvious: it can only prevent sequential-access buffer overflow that is used to directly attack execution-controlling data, and cannot prevent even buffer overflow based on random accesses. 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.

2.2.2 Unexecutable stack area

Until recently, all addresses in the virtual memory space accessible from processes were marked "executable" by many operating systems. This setting has been effectively used to exploit many existing security flaws. An attacker attempting to exploit a stack buffer overflow security hole will send a malicious input string with a program code to be executed at the top part of a string, put data to be written over execution-controlling data after that, and send the attack string to victim programs. The data that replaces the execution-controlling data instructs the victim program to transfer its execution to the code embedded at the top of the attack string, which is in the stack area. In this way, a successful attack can instruct the victim to execute virtually any code the attacker chooses. A variant of this type of attack is to

place an attack code within the environmental variables of a Unix system, which are usually placed at the bottom of the execution stacks.

To prevent this kind of attacks, many operating systems now forbid execution of program code in the stack area. For example, the Solaris operating system by Sun Microsystems forbids stack execution by default from version 9 onwards [68]. Implementation of this feature is difficult in the Intel IA32 architecture, though, because of a shortcoming in the page-based protection design of this architecture. However, both AMD and Intel have recently extended the CPU architecture to support such protection (called NX bits [1, 19]) and Windows XP SP2 has introduced a feature to enable such protection [3].

Still, this protection improves security only slightly. The use of stack-placed execution code for execution hijacking is done only because it is convenient, *not* because it is *required* for attacks. If stack execution is forbidden, attackers will simply start to use a different method. There are many means of attack without using stack execution (for example, that described in [28]).

2.2.3 Memory management using a live-object table

Several implementations check for buffer overflows and other forms of memory access dynamically by using a table of live objects maintained during program execution. Loginov et. al. [41] proposed a method to ensure pointer safety by adding a 4-bit tag to every octet in the working memory. "Backward-compatible bounds checking" by Jones and Kelly [36] modifies the GNU C compiler (gcc) to insert bounds-checking code that uses a table of live objects. Their approach makes it impossible to access a memory which is exterior to any objects (e.g., function return addresses in the stack), but any data in the memory can still read and modified by forging pointer offsets. Jones and Kelly claim their method detects pointer offset forging, but it does not seem to work when pointers stored on memory are overwritten by integers.

Safe-C [5] can detect all errors caused by early deallocation of memory regions. However, they do not mention anything about cast operations and it seems to be not trivial to extend their work to support unlimited cast operations. for the same reason as that of Jones and Kelly's work. Patil and Fischer [53] proposed an interesting method to detect memory misuses. In their method, boundary checking is done in a separate guard process program slice techniques are used to reduce the runtime overhead. However, there are limitations regarding the source and destination types of cast operations.

2.2.4 Various safe languages

First, of course, there are already plenty of languages which ensure complete memory safety. Plenty of dynamically-typed languages (Common Lisp, Scheme, SmallTalk and so on) have been implemented securely. These languages have been used to implement many real-world services. A number of general-purpose script-

ing languages (e.g., Perl, Python and Ruby), as well as many domain-specific languages (such as PHP) are used for web services on the Internet. As long as they are correctly implemented, the implementation of these languages is memory safe because of the nature of the design principle for dynamically-typed languages.

There are also many safe implementations of statically-typed languages. For example, Haskell ([25] for example) and ML and its variants (for example, Standard ML [66, 58] and Objective Caml [56]) have been used to implement various large applications. The design and implementation of these languages not only provide protection against runtime memory corruption, but also help programmers reduce the occurence of conceptual errors in programs through the highly sophisticated design of the type systems. The syntaxes of these languages are designed according to a concept completely different idea from that underlying imperative languages such as C.

There are also imperative strongly-typed safe languages. Among these available safe languages, Java [26] is used for the vast majority of applications. It is used for many stand-alone programs (such as Eclipse), and for many web-based applications on both the host-side (Java Servlets) and the client-side (Java Applets).

These languages have many advantages over the C language for writing new programs. The design of many current languages is more advanced than that of the C language, whose design was essentially fixed in more than 30 years ago intentionally at low level. Recent improvements in the implementation of languages, along with the rapid increase in computing power, now make it practical to run production-level programs in those languages. Regrettably, though, these languages can contribute only a little to countermeasures to block security holes. Many programmers are reluctant to change over to those new languages, and the cost of porting existing C programs to other languages is significant.

2.2.5 Variants of safe C-like languages

Some other safe imperative languages resemble the C language. For example, Cyclone [27, 35] 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 be rewritten, which is a considerable task, At the extreme, Java and (the type-safe portion of) C# can also be considered examples of such languages, but of course porting C programs to these languages is more burdensome.

2.2.6 CCured

Necula et al. has designed and implemented CCured [49, 18], a sound type system which can support C programs including cast operations. The approach of CCured is to analyze the entire program and then split the program into two parts: the “type-safe part” which does not use cast operations, and the “type-unsafe part” which can be contaminated by cast operations. However, to the best of my knowledge, the designers did not focus on perfect source-level compatibility with existing

programs, and in fact the system supports only a subset of the ANSI-C semantics. The reported amount of required rewriting code is less than 1% of the source code, which is much smaller than Cyclone, but still a significant amount. Fail-Safe C 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 Fail-Safe C is that CCured is mainly based on static analysis of cast operations, while Fail-Safe C treats dynamic handling as its main tool. This difference in the main design concept leads to several differences between these two systems. A more detailed comparison made in Section 6.2 after the methods of Fail-Safe C are described in detail.

Chapter 3

Basic Concepts

This chapter gives an overview of the design concepts of the Fail-Safe C system.

3.1 Value representation

3.1.1 Fat pointer and cast flag

To access various information about blocks (e.g., block size and content type) regardless of the pointer arithmetic, Fail-Safe C internally represents all pointers using *fat-pointer* representations: the pair consisting of a base and an offset. The base parts of a fat pointer always keeps the address of the top of a block and the offset part keeps the relative position of the element referred to by the pointer from the top of the block. Values stored in the offset parts are *virtual offsets*, which will be described below (Section 3.2). A special value of 0 can be used as a base part representing "null pointers"; i.e., pointer values which do not point to any objects.

In addition, a Boolean flag, called a *cast flag*, is added to every pointer. This flag indicates whether the pointer can be used for normal accesses or the access methods must be used for memory accesses. The value of the fat pointers will hereafter be given as $(b, o)^f$, where b is the base part, o is the offset part, and f is the cast flag. The cast flag embedded into the base part is placed to the bit which correspond to the word size (i.e., third lowest bit in 32-bit architectures) (Figure 3.2), to enable fast checking of cast flags described in Section 4.2.

During program execution, the following conditions are maintained for all cast flags in the pointers.

- The cast flag of a pointer must be set to 1 if the type of the memory block referred to by the pointer different from that expected from the pointer type (i.e., when the pointer is cast). (As an exception, the cast flag of null pointers can be 0[1].)

[1] The reasons to allow null pointer with cast flag of 0 are

1. Static initializers for zeros (or null pointers) in an array can be omitted in the C language. If

Pointer creation:

$$\&x \longrightarrow (b_x, 0)^0 \qquad \text{where } b_x \text{ is the base address of } x$$

Pointer arithmetics:

$$\underbrace{(b,o)^f}_{T*} \pm \underbrace{y}_{\text{integer}} \longrightarrow (b, o \pm y \cdot s)^f \quad \text{where } s \text{ is the size of type } T$$

Pointer casts:

$$(T')(b,o)^f \longrightarrow (b,o)^{f'} \qquad (f' \text{ is recalculated from } b \text{ and } o \text{ for type } T')$$

Figure 3.1: Arithmetic and cast on fat pointers

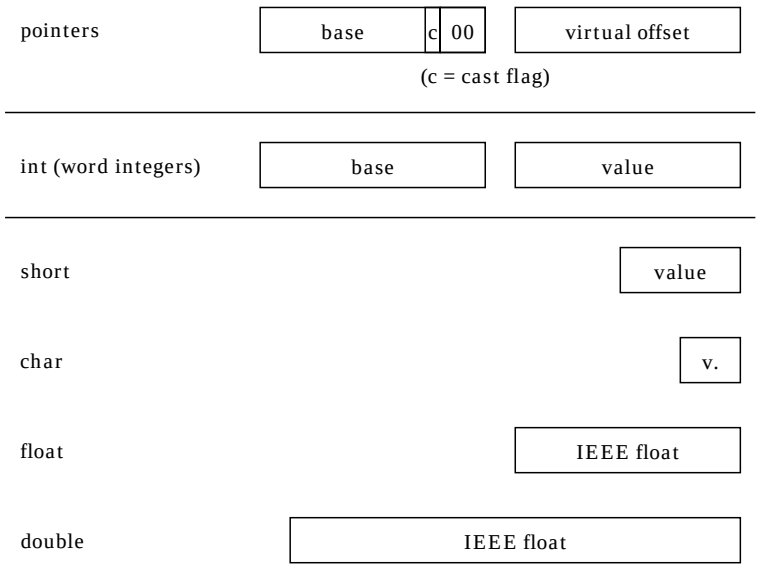


Figure 3.2: Representations of pointers, integers, and floating numbers

- The cast flag of a pointer must also be set to one if the offset field of the pointer is not a multiple of the virtual size of the element type.

Note that the second condition, as well as the first condition, is required for correct access to data inside memory blocks. To meet above conditions, a cast flag is recalculated whenever a value is cast to a pointer type (including pointer-to-pointer casts). Speaking abstractly, the cast flag does not need to be modified when pointer arithmetic operations are performed, because these operations always add or subtract offsets which are multiples of the element size. However, integer overflow conditions might violate the second condition on pointer arithmetics in an actual implementation, when the size of the element type are not a power of 2 (See Section A.2.3.2 for further details). Figure 3.1 summarizes the behavior of fat pointers and cast flags in related operations.

The main reason for introducing a cast flag is to improve performance: although the access methods associated with each memory block can support all kinds of memory accesses perfectly, for either cast pointers and non-cast pointers, these also introduce a heavy execution overhead (equal to about 10 times of the execution time). The cast flags serve as a binary switch embedded in every pointer to selectively provide shortcuts around slow access methods. Introducing cast flags means that all memory-related operations not related to a cast pointer can be served in a similar way and with the same order of execution overhead for several statically-typed safe languages (e.g., Java and ML). A slightly similar concept that mixes both type-universal accesses and type-specific accesses in one system is used by CCured [49, 18]. However, the choices of two semantics in CCured is static (compile time), while in Fail-Safe C is dynamic (execution time) for every pointer. In CCured, if a pointer is statically determined as "possibly cast" (called "wild" in CCured), all values possibly indirectly referred from the pointer must also be determined as "wild" data. This "infection" effect of a cast does not occur in Fail-Safe C. Thanks to the cast flag and the existence of access methods, the bad effect caused by a cast operation only infects the pointer itself, not all subsidiary data. Data referred from the pointer can still use the usual efficient representations of data, and this enables faster operations.

3.1.2 Fat integers

ANSI-C requires that integers whose size is larger than or equal to the size of pointers should be able to hold any pointer values. Integer values which were originally pointers can be cast back to corresponding pointer types if the value

---

the null pointers must have cast flag set to 1, all elements of uninitialized fat pointer arrays must be translated to explicit initializers ($(4, 0)$ on 32-bit architectures).

2. Future versions of Fail-Safe C will implement an analysis to find some pointer variables which do not point to differently typed objects. Because null pointers frequently appear even on those pointers, they should be included into the set of "well-typed" pointers. Otherwise, the effectiveness of these analysis will decrease.

Integer creation:

$$i \text{ (constant)} \longrightarrow [0, i]$$

Integer arithmetics:

$$[b, v] \odot [b', v'] \longrightarrow [0, v \odot v'] \qquad (\odot \text{ may be any operator})$$

Casts between integers and pointers:

$$(\texttt{int})(b, o)^f \longrightarrow [b, b + o]$$

$$(T*)[b, v] \longrightarrow (b, v - b)^f \qquad (f \text{ is recalculated from } b \text{ and } v - b \text{ for type } T*)$$

Figure 3.3: Arithmetics and cast on fat integers

is not modified while they are integers. To implement this behavior, the usual one-word representation of the integers is of course insufficient because we cannot distinguish those integers (valid as pointers) from arbitrary integers. Therefore, Fail-Safe C uses two-word representation for integers also, which are called *fat integers*.

Conceptually, the same representation as for the fat pointers can be used for fat integers. However, to enable more efficient implementation of integer arithmetic operations, the current design of Fail-Safe C uses the representation that slightly differs from that of the fat pointers. A fat integer in Fail-Safe C is internally handled as a pair consisting of the base and a value (or *virtual address*), hereafter written as $[b, v]$. The virtual address is defined to be equivalent to the sum of the base and the offset. This mapping provides an injective map from the in-boundary fat pointer values to integers[2] (ignoring cast flags); i.e., the virtual addresses of two different elements in any memory block (including the elements of different blocks) are different.

All arithmetic operations on integers ignore the bases of operands and only operate on the value parts. Arithmetic result always have base part of 0, corresponding to a null pointer. A cast operation between pointers and integers converts the representations according to the above-defined mapping. Figure 3.3 summarizes the behavior of fat integers on related operations.

Integer types that are narrower than the pointer size (e.g. `char` and `short` in typical 32-bit architectures) cannot have any pointer value: thus, the representations the same as native ones are used in Fail-Safe C.

[2]The mapping from all fat pointers (including out-of-boundary pointers) to a virtual address is surjective rather than injective.

3.2 Typed 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 contents for every memory block. The runtime 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 in Fail-Safe C. Each block consists of a *block header* and a *data area*. A block header contains information on 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 may depend on its data representation type: a different representation can be used for each representation type. This allows the implementation to utilize several different representations for each types appearing in user programs. Basically, an array of values in the same representation as scalar values is used in the data area. For example, the system basically uses a simple array structure identical to that of conventional compilers for data of type `double`, and a packed array of two-word encoded pointers for data of pointer types (e.g. `char *`). The actual representations used in the memory blocks of various data representation types will be described in detail in Section A.1.1.

3.2.1 Virtual offsets

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

The situation is more complicated in Fail-Safe C, though, because there is a cast operation which 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 the C language secure): if a pointer is cast to `char *` type, the pointer will point to every byte of the internal representations of several data including pointers. If the internal information of pointers required to check the boundary condition of memory blocks (i.e., the 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 and BCC seems to suffer from this problem. CCured [49, 18] solves the problem by maintaining a bit-array for each memory block indicating whether each word in the block can be used as a valid pointer to the top of block; however, the handling is rather complex and not intuitive.

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 also makes it impossible

to properly represent a cast pointer to data types having alignment requirements smaller than the element size (e.g. structs); i.e., the specification allows pointers not aligned to elements.

As a consequence, another method of addressing had to be created for Fail-Safe C. The addressing used in the Fail-Safe C system is called *virtual offset*, which corresponds to a *program-visible* size (hereafter called the *virtual size*) of elements, not the actual size of representations altered to implement security mechanisms. For example, the virtual size of a natural-sized integer in Fail-Safe C will be equal to the native word size—although these values uses 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 will have virtual sizes equivalent to the real sizes. In other words, the virtual size of every type will be the real size of the equivalent data type in the native implementation of the C language. This definition of virtual offsets does not lead to the problems that arise with the other two methods: a cast pointer temporarily points to the middle of elements can be properly cast back to its original type, and specifying only the base part of fat pointers is not possible because there is no way to point to only the base part of pointers.

Another important consequence of this representation is the possibility of consistent definition for memory accesses performed via cast pointers. 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, C programmers are often skilled at using this sort of access and find it useful. The Fail-Safe C system allows 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 in Fail-Safe C correspond to real sizes in native implementation, the semantic mapping from a Fail-Safe C representation of data to the corresponding native representation can be defined. For example, the four bytes read from inside one integers (assuming a 32-bit architecture) via a cast pointer can be defined in a way that the concatenation of four 8-bit values (as binary numbers) constitutes a 32-bit value equivalent to the original integer, as is usual in the native implementation.

3.2.2 Access methods

As the actual representation of data in a memory block differs from that of 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. Fail-Safe C uses an object-oriented implementation technique for this purpose.

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

implementation is stored in the type-information blocks.

Methods stored in method tables (*access methods*) implements a generic interface for read/write block contents in various sizes—such as byte or word, regardless of the type of block. A read method receives a virtual offset and returns a corresponding content in a data area as a fat integer[3] if 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. These methods will signal a runtime error if the virtual offset is outside the block boundary.

3.2.3 Memory operations

The dereferencing of a pointer is not trivial. We need to know if a pointer refers to a valid region and if the type of the target value is correct. The basic memory dereferencing method used in the Fail-Safe C system is as follows.

1. Check if the pointer is `NULL` (i.e., base $= 0$). If so, generate a runtime error.
2. If the cast flag is not set, compare the offset with the size of the referred memory block. If it is inside the boundary, read the content in the memory. Otherwise, generate 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 block, 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 invariant conditions regarding the pointer value shown in Section 3.1.2 ensure 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 for 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 block may differ from the statically expected one. In this case, the code sequence delegates the actual memory read operation to the handler method associated with the block.

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

The actual implementation, however, is more complicated to enable higher performance and higher compatibility. The diversion from the simple semantics described here is discussed in Sections 4.1.2, Section 4.1.1, and Section 4.2.

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

3.3 Memory management

Fail-Safe C utilizes 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 forbids further access to the block.[4] The garbage collector will later checks if there are no pointers pointing to the block, and then releases the memory block.

3.3.1 Temporal properties of local variables

A pointer to a local variable is slightly problematic for the Fail-Safe C system. Such a pointer may escape its scope by being assigned to a global variable or other external data structure, and continue to exist even after the execution of the function containing the local variable ends. However, such a local variable is usually allocated in the call stack and will disappear unconditionally when the function execution finishes. The solution applied by the Fail-Safe C system is simple: all local variables whose address is taken are allocated in the heap area, and the garbage collector will take care of deallocation. A code allocating memory blocks for these variables is inserted at the top of functions along with the value initializations.

3.4 Structures and unions

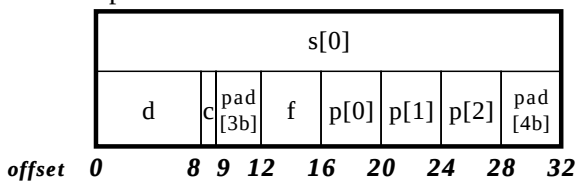
A struct(ure) value in ANSI-C language is a kind of first-class value: although it has an internal structure, the value can be assigned to variables like other scalar values, or it can be passed to functions as an argument or returned as a result, none of which can be done for arrays. Fail-Safe C maps each struct declared in a user program to another struct that has basically a same set of elements, each of which is translated to its corresponding value representation. As the representations inside translated structs become non-uniform and change for every structs declared, the type information and access methods have to be automatically generated during compilation. An example of a translated structures is shown in Figure 3.4. The size of a translated representation for the a struct will not exceed twice the size of the corresponding native struct (including any padding), because each element of the native struct can be placed at the offset which is twice of the original offset in the translated representation in the worst case. This upper bound is important for the handling of heap-allocated structure, described in Section 4.3.

On the memory blocks of struct types, only one block header is added to the top of a block, but not for each elements of the blocks. As a consequence of this and the rule of cast flags explained in Section 3.1.1, every pointer pointing to an

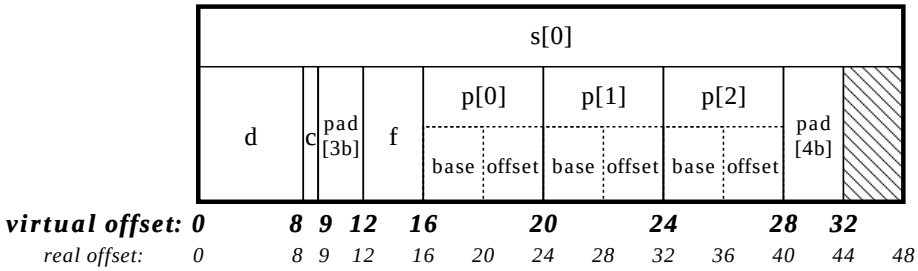
[4] This behavior differs slightly from that of most safe languages because user programs are supposed to call `free()` function to declare explicitly that memory blocks are no longer intended to be used.

```
struct {
  double d;
  char c;
  float f;
  char *p[3];
} s[2];
```

Native representation:



Translated representation:



1. The representation shown is for a big-endian 32-bit machine which requires double-word alignment for `double` type.

2. A 3-byte padding labeled “pad[3b]” aligns field *f* to the a word boundary in both virtual/real addressing.

3. A 4-byte paddings labelled “pad[4b]” aligns the whole structure to a double-word boundary (which is required by field *d*) in the virtual addressing.

4. A 4-byte paddings 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 3.4: An example of the representation of a struct

individual element of a struct must have its cast flag set, even if there is no cast in the original program. The accesses through these pointers will be handled by access methods, which creates some runtime overhead. As current Fail-Safe C implements all accesses to arrays through pointer arithmetic, all accesses to array-type elements inside structs are done through access methods. This is a current limitation of Fail-Safe C. The main reason for not adding headers corresponding to the elements is that if will leads to two different pointer representations both of which point to the same element of a struct (one through the outer header to the struct, and one through the inner header to the element). Because the inner header requires a memory space and some types have larger real sizes than virtual sizes, the virtual addresses, or the integer equivalents, of these two pointers will be differ, which will complicate the semantics and confuse both the programs and users. For C programs, finding a complete solution to this problem is likely to be difficult. I plan to reduce this unwanted overhead through program analysis and better handling of array accesses.

Unions in C language are treated as a kind of implicit cast operation in Fail-Safe C. For example, a program

```
struct S1 { int x; char *y; };
struct S2 { int x; double y; };

union U1 { struct S1 s1; struct S2 s2; };

union U1 u1 = { 1, 1.0 };

int main(void) {
    u1.s1.x; u1.s1.y;
    u1.s2.x; u1.s2.y;
}
```

is translated into a program equivalent to

```
struct S1 { int x; char *y; };  /* size = 8 */
struct S2 { int x; double y; }; /* size = 16 */

struct U1 { struct S1 s1; char __pad[8]; };
     /* __pad required for making size correct */

struct U1 u1 = { {1, 1.0}, {0} };

int main(void) {
    ((struct S1 *)(&u1))->x;
    ((struct S1 *)(&u1))->y;
    ((struct S2 *)(&u1))->x;
    ((struct S2 *)(&u1))->y;
}
```

at an early stage of compilation.[5] Access methods perform the conversion neces-

[5]The translation is performed after adding padding for every vacant byte in structures, to avoid problems arise from alignment incompatibility.

sary to support the various (sometimes peculiar) operations performed on union values.

3.5 Functions

User-defined functions are translated into functions taking and returning values in the translated representations. Direct invocations of user-defined functions (and library functions) are simply translated into function invocations for the translated functions. Section A.2.3 provides a detailed description of the translations of function bodies in the current implementation of Fail-Safe C.

There are two topics which requires additional handling—variable arguments and function pointers.

3.5.1 Variable arguments

Variable arguments, or varargs, are a feature of the C language which allows the number of arguments for a function (including a user-defined function) to change for every invocation of the function. The most widely used instance of vararg functions might be the `printf()` function in the standard library. In the usual implementation of the C language, varargs are typically implemented in the following ways[6] (Figure 3.5).

- The caller puts the arguments in the reverse order of the parameter list onto the stack. This means that the fixed arguments, which appear before variable arguments in the parameter list, are placed at the top of the arguments in the stack, in a fixed location relative to the frame pointer.
- The called function accesses fixed arguments through addressing relative to the frame pointer. This works whatever the number of arguments are pushed by the caller.
- The function calculates the address of the first variable argument, either from the address of the last fixed argument or using implementation-provided loopholes. For example, the GNU C compiler (gcc) provides a special pseudo-function `__builtin_va_nextarg` for this purpose.
- If more variable arguments are required, the addresses of these are calculated using the address of the previous variable argument.

Of course, this native method of vararg handling is unsafe and not directly applicable to Fail-Safe C. However, Fail-Safe C should behave similarly regarding the use

[6]The implementation of varargs depends heavily on the underlying architecture and the ABI definitions. For example, on the SPARC32 architecture arguments are passed in registers as long as the number of hardware registers permits. The called varargs function first puts all register-passed arguments at the top of the stack by itself to construct the stack format described here.

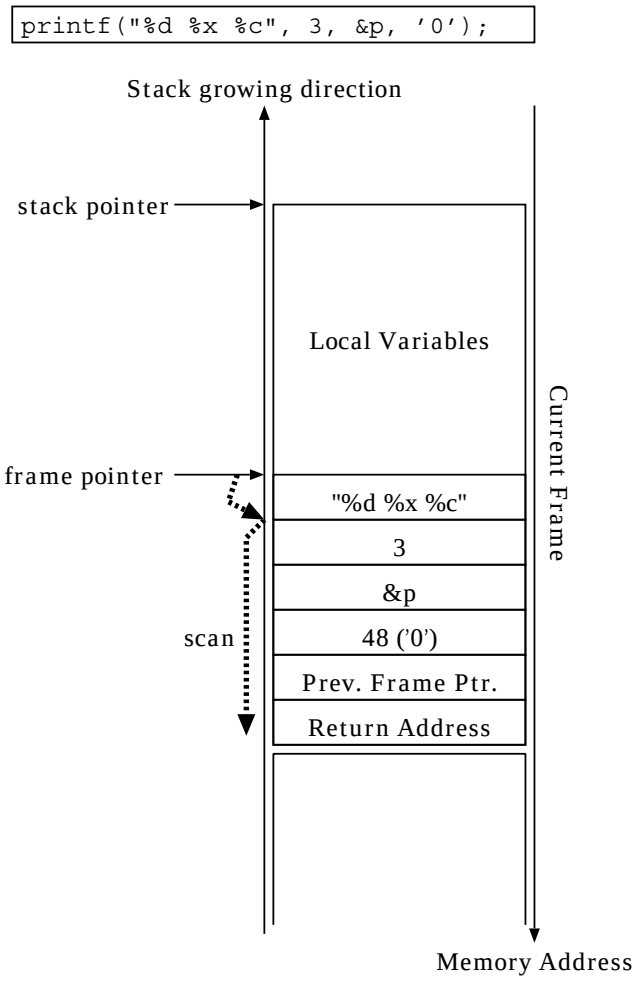


Figure 3.5: Handling of varargs in a native compiler

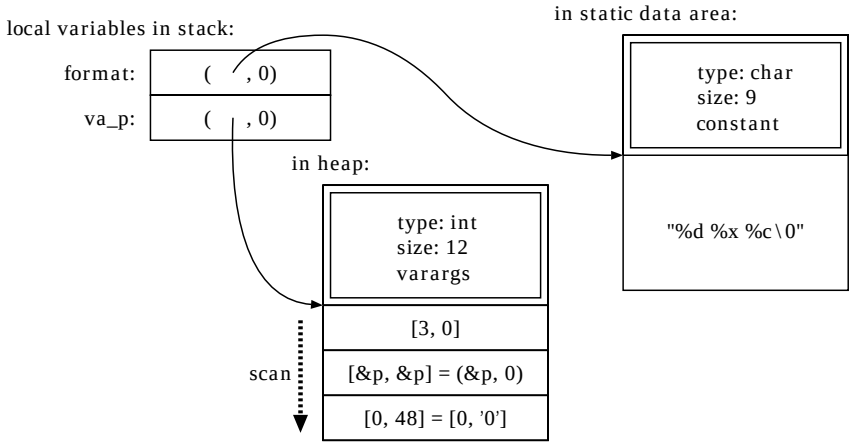


Figure 3.6: Handling of varargs in Fail-Safe C

of varargs because many existing programs depend on the behavior of the above implementation to some extent. (For example, many programs print the value of a pointer by using `printf` with integer conversion specifiers like "`%08x`", not using a proper specifier for pointers "`%p`".)

The Fail-Safe C implementation of varargs is as follows (Figure 3.6): all vararg arguments are stored in a temporarily allocated block of fat integers from the first one to the last. The address of the block is passed to functions as a hidden, additional parameter. The function will then take varargs from the block, sequentially from the top. Comparing Figure 3.4 and 3.5, we can see that there is a natural correspondence between the semantics in the two implementations. If the arguments passed are redundant, the rest of the arguments will be silently ignored, similar to with the native semantics. If an argument is insufficient, fetching the missing varargs will cause a runtime error, in the same way as access violations do in normal memory blocks.

3.5.2 Function pointers

The invocation of a function via pointers is complicated, again because of the existence of a cast. If a function pointer is not cast, simply invoking the referred function as usual is sufficient. However, if a pointer is cast, the referred function may expect incompatible arguments[7], or the pointer may not even point to a function.

Fail-Safe C solves this problem by again using an implementation technique

[7]Even if the interface is fortunately "compatible" in the native semantics, it may become incompatible in Fail-Safe C. For example, pointers to different types have incompatible representations in Fail-Safe C. The sample code shown in Figure 1.1 is an instance affected by this incompatibility.

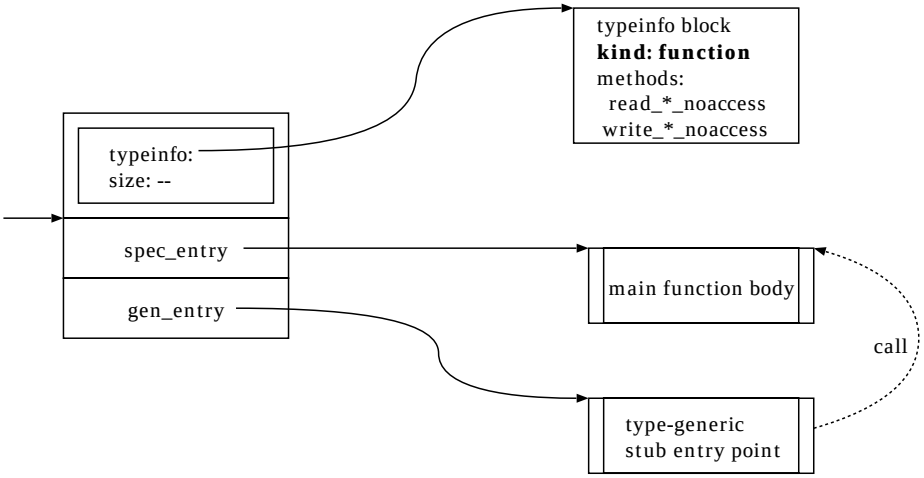


Figure 3.7: The structure of function stub blocks.

borrowed from object-oriented languages. In addition to the usual entry points used for direct invocation of functions, Fail-Safe C generates a *generic entry point* for each function, which uses a common interface unified for all functions. Generic entry points receive all arguments in the form of varargs, as described above. There is also a memory block generated for each function, called a *function stub block*. It contains two pointers to the both entry points of functions, and is tagged with a special mark as a block corresponding to a function. Figure 3.7 shows the structure of a function stub block.

If a pointer to be invoked is cast, the caller checks the special mark on the referred block, takes the address of the generic entry point, and passes all arguments as varargs. A generic entry point then takes arguments from the vararg block, converts representations, and then passes them to the usual entry point of the function. If the pointer is not cast, the caller can instead take the address of the usual entry point and call it directly.

3.6 Theoretical aspects of the system design

In the final section of this chapter, some concepts underlying the system design of the Fail-Safe C are explained.

3.6.1 Invariant conditions and safety

As explained in Section 3.1.1 and the following sections, valid fat pointers and fat integers are defined as follows:

Definition 3.1 A fat pointer $(b, o)^f$ is valid as a pointer to type T when

1. the base b is an address of a valid memory block (a global variable, a function block or a heap object), or 0, and
2. if the cast flag f is 0,
   (a) the object at the address b has dynamic type T when b is not 0, and
   (b) the offset o is a multiple of the size of type T.

Definition 3.2 A fat integer $[b, v]$ is valid as a value of wide integer type value when the base b is an address of a valid memory block or 0.

The key point of Definition 3.1 is that the cast flag f dynamically chooses one of two well-known strategies to confirm the safety of programming languages. If f is 1, a pointer is similar to a reference in dynamically-typed languages (Lisp, Scheme, etc.). In dynamically-typed languages, any reference can point to any valid objects in a heap area, but all dereferencing operations must first check the type of the referred object. In contrast, a pointer with cast flag 0 is similar to a reference in statically-typed languages (ML, Haskell, etc.). In these languages, every reference must point to an object of the corresponding types, but dereferencing operations can blindly assume that the static type of the pointer are reliable. Setting the cast flag f of all pointers to 1 causes the whole system to degenerate to one similar to those of a dynamically-typed language, possibly becoming much slower than the current system. In contrast, forcing all cast flags to be 0 makes the whole system very similar to that of statically-typed language, where pointer cast operations are forbidden. Table 3.1 summarizes the differences between dynamically-typed and statically-typed languages and Fail-Safe C. The fat integers are, conceptually, simply `void*` pointers with a lightly different representation.

Thus, we should be able to derive the proof of safety from usual proof of safety for typed safe languages with reference cells once a complete dynamic semantics is written down for Fail-Safe C. The usual proof of the safety for the typed safe languages with reference cells—for example, the one shown in Chapters 13 and 14 of [55][8]—follows the following steps.

1. Define a well-typed condition of a store, or the state of memory locations, based on the definition of the well-typedness of values recursively applied to the element in the memory state according to store types.
2. Prove the preservation property, which is defined to preserve well-typedness of store types, as well as the types of evaluating terms and others.
3. Prove the progress, assuming the well-typedness of the current store.

[8]This reference concerns functional languages, but the basic principle of the proofs can also be applied for imperative languages.

Table 3.1: Comparison of several aspects of dynamically-typed languages, statically-typed languages and Fail-Safe C

	dynamically-typed languages	statically-typed languages	Fail-Safe C (f: cast flag)
Pointers may point to invalid address	no	no	no
Pointers may point to null address	yes[a]	yes[a]	yes
Pointers may point to object of unexpected type	yes	no	when $f = 1$
Pointers may point to object of expected type	yes[b]	yes	yes
Dereference possible without type checking	no	yes	when $f = 0$
Dereference possible after type checking	yes	yes[c]	yes
Runtime type information required	yes	no	yes

[a] If the language provides such feature.
[b] If any "expected type" is definable.
[c] If runtime type information is available.

The well-typed condition of a store can be simply derived from the usual recursive structure of definitions and our definition of the well-typedness of fat pointers. Structs can basically be treated like a record. The proofs of preservation and progress basically inherit the original structures. Obviously, the main difference in these proofs will be in the handling of cast pointers. For the preservation property, the read from store via a cast pointer will evaluate to a value which is explicitly coerced into the expected type (see step 3 in Section 3.2.3) if the evaluation is to succeed without errors, which satisfies the requirement. For the progress property, the important point of proof will be that if a read operation refers to a memory block of a different type, the result of a one-step evaluation should be defined for all possible types in the program *if the referring pointer has a cast flag set*, as in the definition of dynamic semantics for untyped languages (this can lead to an explicit error condition, though). The reduction of non-cast pointers dereferencing can be a partial function, as is usual in statically-typed languages, and it corresponds to the implementation of direct memory accesses.

The complete proof of safety will be derived in future work.

3.6.2 Partial compatibility with native compilers

The second issue of discussion is the compatibility with the semantics of native compilers.

One design principle of Fail-Safe C is to always maintain a one-way mapping between the state of the program running on Fail-Safe C to the corresponding state of the program running on the native system. As implied by the cast operation definitions given in Sections 3.1.1 and 3.1.2 and the virtual offsets in Section 3.2.1, and many other descriptions, the intended mapping can be defined as the following erase operator:

Definition 3.3 A base-erasing function erase(), or $|\cdot|$, for scalar values and struct values can be defined as follows:

- erase for pointers:

$$|(\texttt{NULL}, x)^f| = x$$
$$|(b, o)^f| = b + o$$

- erase for integers:

$$|[\texttt{NULL}, v]| = v$$
$$|[b, v]| = v$$

- erase for objects:

$$|\{p_1, p_2, \ldots, p_n\}| = \{|p_1|, |p_2|, \ldots, |p_n|\}$$

After a similar definition provided for the program state and other things has appeared in the proofs, the following rough sketch of a commutative diagram can be imagined for the single-step evaluation of Fail-Safe C (step_{FSC}) and the native semantics (step_{C}):

$$\begin{array}{ccc} \Sigma = (H,S,P) & \xrightarrow{\text{erase}} & |\Sigma| = (|H|,|S|,P) \\ \Big\downarrow{\scriptstyle \text{step}_{\text{FSC}}} & & \Big\downarrow{\scriptstyle \text{step}_{\text{C}}} \\ \Sigma' = (H',S',P') & \xrightarrow{\text{erase}} & |\Sigma'| = (|H'|,|S'|,P') \end{array}$$

(H: state of heap store, S: state of local variables, P: evaluating program)

If this diagram holds, it roughly means the translated program will behave in the same way as the corresponding native program does. More precisely, the following property can be proven:

Partial Compatibility: the program behaves in the same way as usual programs, if the Fail-Safe C system does not generate a runtime error.

$$|\,\text{step}_{\text{FSC}}(\Sigma)| = \text{step}_{\text{C}}\,|\Sigma| \qquad \text{if } \text{step}_{\text{FSC}}(\Sigma) \neq \text{error}$$

The definition of step_{C} can be simple; for example, using the usual flat model of a byte array (a partial map from the integer address to the byte value) to express memory states. In the actual proof, there may be some kind of universal/existential qualifiers around the above equation to handle indeterminism in some operations (e.g., the addresses of allocated memory area). The main difficulty regarding these proofs will be the handling of indeterminism appearing in both sets of semantics.

3.6.3 Completeness (full compatibility)

The final thing to prove is that a the correct ANSI-C program does not fail under Fail-Safe C. However, it is difficult to formally define formally what is a "correct" ANSI-C program. For example, if the pointers are represented simply by integers corresponding to memory addresses, completeness does not hold. A counterexample is a small piece of program

```
char a[1];
char b[1];

char test(void) {
 char *p = &a;
 char *q = p + ((int)b - (int)a);
 return *p;
}
```

which works with the simple native semantics (because `q` will have the valid address of `b`), but fails in Fail-Safe C (because `q` points to a memory block of `a`, and

the address of `b` is outside that region). Several attempts have been made to formally define the semantics of the C language, however, but none has been entirely satisfactory. For example, Papaspyrou [51] does not provide a definition for cast operations, thus which is insufficient for a proof regarding the semantics of Fail-Safe C. Norrish [50] formalized the semantics of the C language in the form of input for the HOL theorem prover, but this also seems to lack any formalization of cast operations. It assumes that every values of every types has an equivalent representation as a byte array, thus the same problem will arise as with the simple definition given above.

The most natural modeling of ANSI-C semantics is likely to be one using a partial map from a memory address to a byte value as a memory model, except that every word in memory (and every integer) remembers whether a value points to a specific memory region and if so which region. This will resemble a degenerated Fail-Safe C system in which all memory blocks and all pointers use fat integers as a representation. In Fail-Safe C, there is one-to-one mapping between fat pointers and fat integers, except for cast flags, and all memory blocks will behave in the same way as fat integer blocks when access methods are used, Therefore, the correspondence between the degenerated system and the full Fail-Safe C system can be easily traced.[9]

3.6.4 Future extension: certifying/certified compilation

Provided that the safety properties described in the previous sections are proven, the Fail-Safe C system can contribute to the safety of the entire operating system. If all programs are guaranteed to be compiled with Fail-Safe C and other safe languages, the underlying operating system need not rely on a hardware-based memory-protection mechanism. (Such mechanisms are currently used on most of modern operating systems.)

For example, the SPIN microkernel system [7] uses Modula-3 language [30] and a custom C-like language called Cove to ensure the safety of memory access and system interfaces without the help of memory management units. Kernel-mode Linux [42] enables any kind of user programs to run in a kernel mode of a Linux system, assuming that the program safety is ensured by some means such as binary verification using Typed Assembly Language (TAL) [45, 46, 47]. Fail-Safe C may allow these systems to become inter-operable with general C programs.

To support dynamic loading of binary programs on these systems, the system must have some mechanism to guarantee that the loaded program is certainly compiled by safe compilers. As such binaries are generated by software, digital signing of the binaries will not work well, because it is easy to sign a forged binary program with the same key that safe compilers use. Instead, most of these systems

[9] Obviously, the semantics of the degenerated system are not strictly equivalent to ANSI-C, but they seem to include ANSI-C, which is sufficient for the completeness proof. In addition, the Fail-Safe C does not detect some undefined behaviors in ANSI-C; for example, creation of an out-of-bounds pointer without it ever being used.

use load-time program verification to ensure that the program meets required static safety preconditions (usually well-typedness) and have correctly embedded run-time checks required in addition to static preconditions. To use Fail-Safe C on these systems, the program compiled by the Fail-Safe C compiler must be verifiable in some way. To make load-time verification of complex programs generated by compilers practical, the compilers should add additional information that works as an "oracle" of verification. This technique is called *certifying compilation*, and a kind of Proof Carrying Code [48, 4, 29] may be useful for the Fail-Safe C system. Another possibility might be an extended version of TAL, but a large extension will probably needed to certify Fail-Safe C programs under TAL.

Another kind of certification technique can also be usefully applied with Fail-Safe C system. *Certified compilation* ensures that the code generated from a user program by compilers has the same operational behavior as one predefined by static and dynamic semantics. Because the program code generated by the Fail-Safe C compiler is complicated, such certification can be a valuable way to enhance the effectiveness of the safety proof discussed above.

Chapter 4

Advanced Features

This section describes some additional ideas implemented in Fail-Safe C to improve compatibility and execution performance.

4.1 Features on memory block

4.1.1 Additional base storage area

There is a small chance that fat pointers are written to the fields in memory blocks which contain neither a fat pointer nor a fat integer. Typical cause of this might be either the use of unions or the lazy type-decision which will be described in Section 4.3. If such a situation happens, written fat pointer will lose its base part and converted into a null pointer, which might cause a runtime error later.

To remedy this problem, the Fail-Safe C system allocates an additional base storage area for once a pointer value is written over any narrow values (Figure 4.1), and stores the base parts into it. The real size of the storage is the virtual size of the structured data area, rounded down for word alignment. Each word in this area corresponds to each (virtual) word at the same virtual offset in the structured data area. If some words in the structured data area already hold fat pointers or fat integers, the corresponding slots of the additional base area will not be used (Figure 4.2). Base address storages are neither modified nor read when memory blocks are accessed via non-cast pointers.

The handling of the remainder data are has one small, almost negligible shortcoming. If a non-null fat pointer is written over some narrow data, and then a part of the corresponding word is overwritten via well-typed pointers, then the base part written to the additional base area at the first step is not cleared, although theoretically the word should not be treated as a valid pointer. This behavior does not break the safety of the system, and thus the current implementation of Fail-Safe C ignores this for the sake of execution performance.[1]

[1] If users want this problem to be fixed for debugging, all direct write accesses for blocks with additional base area can be prevented by changing the *fastaccess-limit* of a block to zero when an

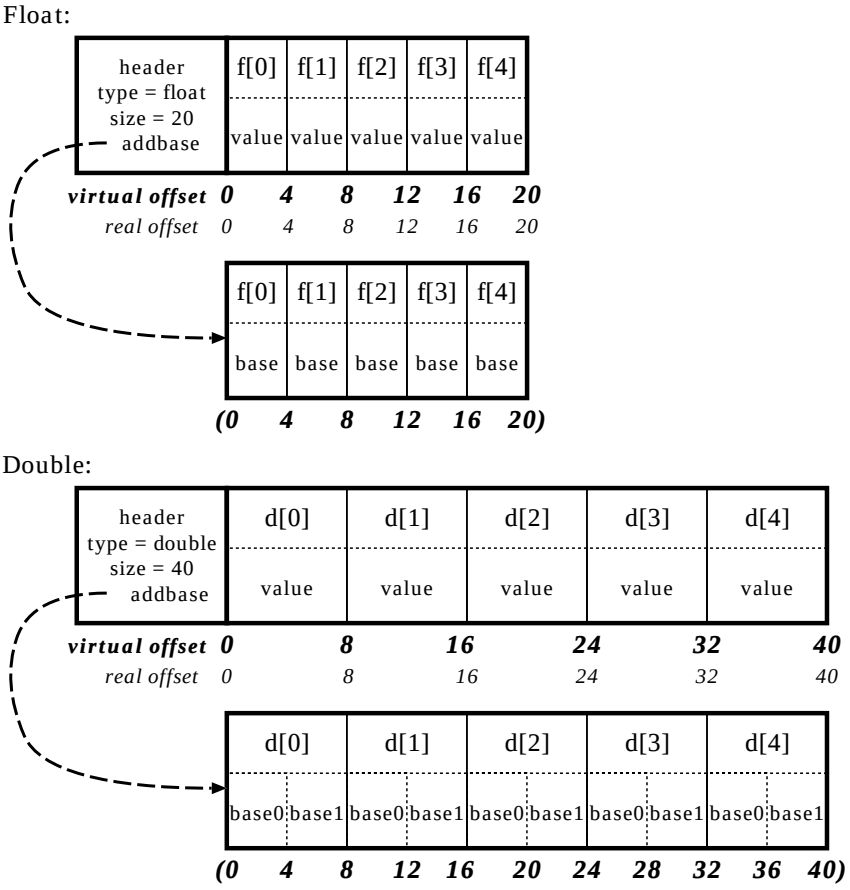


Figure 4.1: The representation of additional base area for primitive types

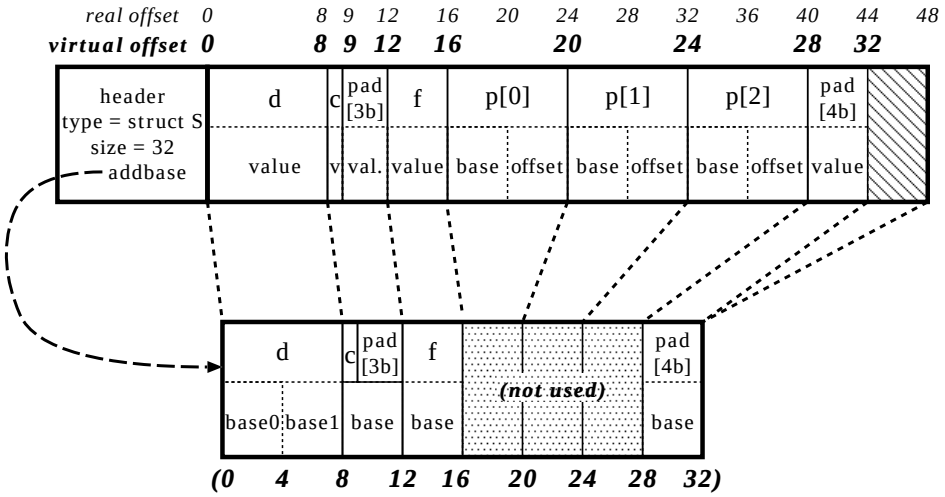


Figure 4.2: The representation of additional base area for (non-continuous) structs

4.1.2 Remainder data area

Sometimes C programmers allocate an memory area whose size is not a multiple of the size of its data type, to implement a "variable-sized structure" (described in Section 1.2(3)). In such case, Fail-Safe C allocates a "remainder area" to handle memory operations on these surplus memory area.[2]

The data format in a remainder area depends on the data representation format of the main part of the block: if the representation is equivalent to the native representation (hereafter called *continuous* data representation), the format of the remainder data will also be a flat, native-compatible representation. In other words, the main data area and the remainder data area are continuously represented in the native-compatible format.[3] An additional base storage area is used when fat values are stored into remainder data area (Figure 4.3).

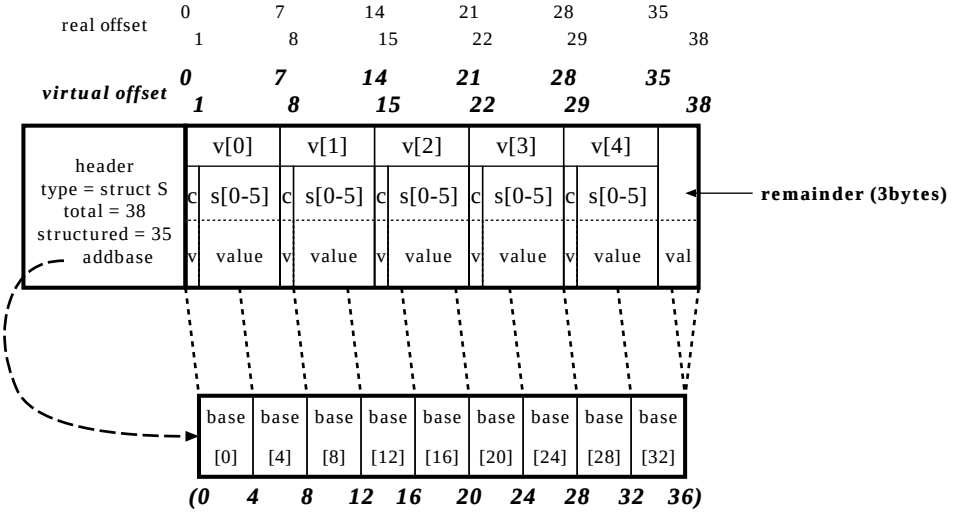
In contrast, if the representation in is not continuous, a "separate" format is used for remainder area: the value part of data are laid out sequentially, then the base part of values follows. If the size of remainder area is not multiple of machine word size, the number of base addresses are truncated down. I chose this separate format for a remainder data area because the most common use of those indivisible

additional base area is allocated for the block. This, however, sacrifices the execution performance in a large amount.

[2]There will be no remainder area for any statically allocated data blocks, because such a data structure cannot be represented statically in the syntax of the C language.

[3]The main reason for choosing this format is that a size of the main data area of continuous types may be indivisible by the word size. A word in additional base area might corresponds to the word which lays over both main data area and the remainder data area (the word base[32] in the upper case of Figure 4.3).

```
struct S { /* continuous */
 char c;
 char s[6];
};
struct S *v = malloc(38);
```




```
struct S { /* non-continuous */
 char *p;
 float f;
};
struct S *a = malloc(22);
```

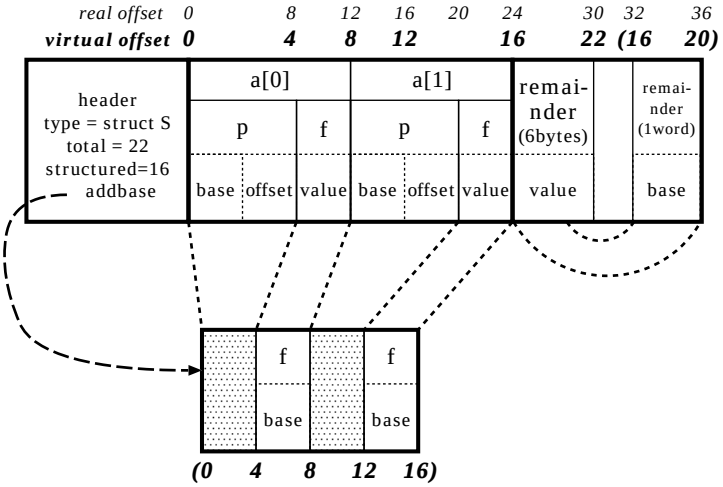


Figure 4.3: Formats of remainder area

data size is to put data buffer (usually in `char` type) after dynamically-allocated data structures. Thus, the format of this area is optimized for raw data storage instead of pointer storage.[4] Furthermore, if the all elements of a data block are fat values, allocating an additional base storage are only for the remainder area is superfluous.

4.2 Fast checking of cast flags

When a fat pointer is dereferenced, three properties must be checked before directly accessing a data area of the referred memory block: (1) that pointer is not null, (2) that the pointer is not cast, and (3) that the virtual offset of the pointer points to an interior part of the memory block (Figure 4.4). While (1) and (3) are common to almost all safe languages having flat array types (e.g., Java, ML, and Lisp), Fail-Safe C also needs (2), whose overhead of is not negligible. To avoid this overhead, the implementation uses a clever trick.

First, every block and block header are double-word aligned so that every base address of a block will have 0 on the bit corresponding to the cast flag. Next, the cast flag in fat pointers are located to a bit corresponds to the word size (Section 3.1.1), so that the base part of a cast fat pointer will have the integer value which is larger than the corresponding block address by the word size, exactly. Finally, each block header has an extra word which always contains a zero at just one word after the location of *fastaccess-limit*. Then, as a consequence of the three properties, if a code refers to the *fastaccess-limit* field of the header from 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 fastaccess-limit field (Figure 4.5).

In other circumstances, if a null pointer is dereferenced as if it were a valid pointer, a offset checking code which attempts to read the *fastaccess-limit* field will access to very end of the address space (because of an integer wraparound). In most operating systems, no memories are 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 the checks can be merged into one offset check, which is necessary anyway in a general situation, without damaging safety properties. An experiment has shown that this reduces the program execution time in memory-heavy benchmarks by roughly 4% to 18% (Section 5.3).

4.3 Determining types of blocks

The implementation of memory blocks in Fail-Safe C depends on the type information associated with each memory block. However, there are many situations

[4]The newer specification of C language [34] (usually called C99) supports explicit declaration for variable-size fields in the tail of structures. In future extension of Fail-Safe C to C99, the data format for remainder data area might be changed to reflect the declared data type for that area.

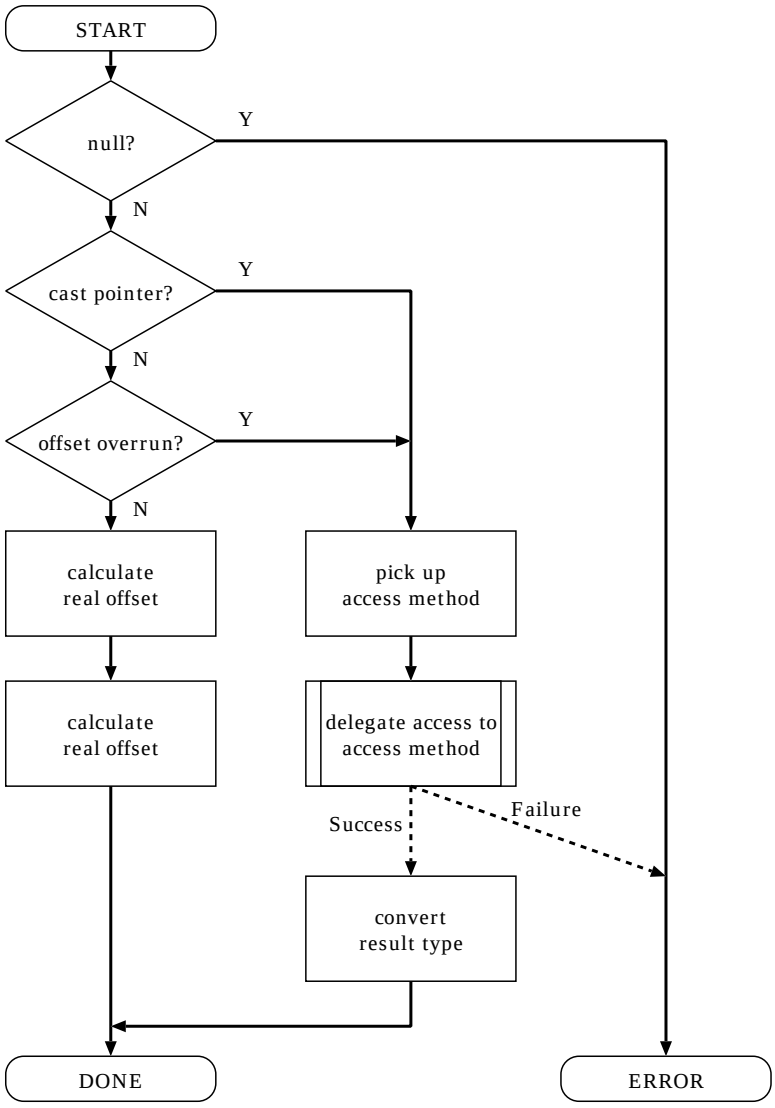


Figure 4.4: Unoptimized procedure for memory access via pointers

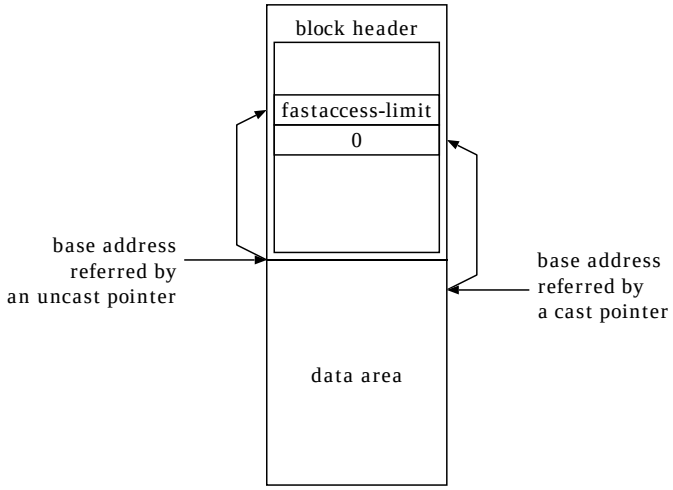


Figure 4.5: Fast cast-flag check.

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, Fail-Safe C does not completely rely on a static knowledge of types. Fail-Safe C delays deciding the type of dynamically-allocated blocks if the type cannot be reliably deduced.

If an untyped block is allocated, the system will first assign a special pseudo-type (called *type-undecided*) to the block. Because this pseudo-type is not equal to any real types, the first write accesses 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. For a last resort, if the block type estimation fails, cast pointers and access methods will maintain the compatibility and let program continue running, where it only slows the execution.

A type-undecided blocks has basically the same structure as te usual blocks. The real size of the allocated buffer will be about twice the requested virtual size, as this is sufficient (see Section 3.4). More precisely, it will be $\lceil ws \cdot (\lceil s/ws \rceil + \lfloor s/ws \rfloor) \rceil$ where s is the requested virtual size and ws is the word size. In some cases the allocate memory area will be excessive, especially when the type is determined to be a continuous type. As a special handling, if the determined type is continuous, the runtime system will reuse unused area as an additional base area of the block.

The type information field in the header points to a specially-defined type-information block. In addition, the size of structured data area (*structured-limit*) is initialized to zero. This causes all accesses to this block to be trapped and dele-

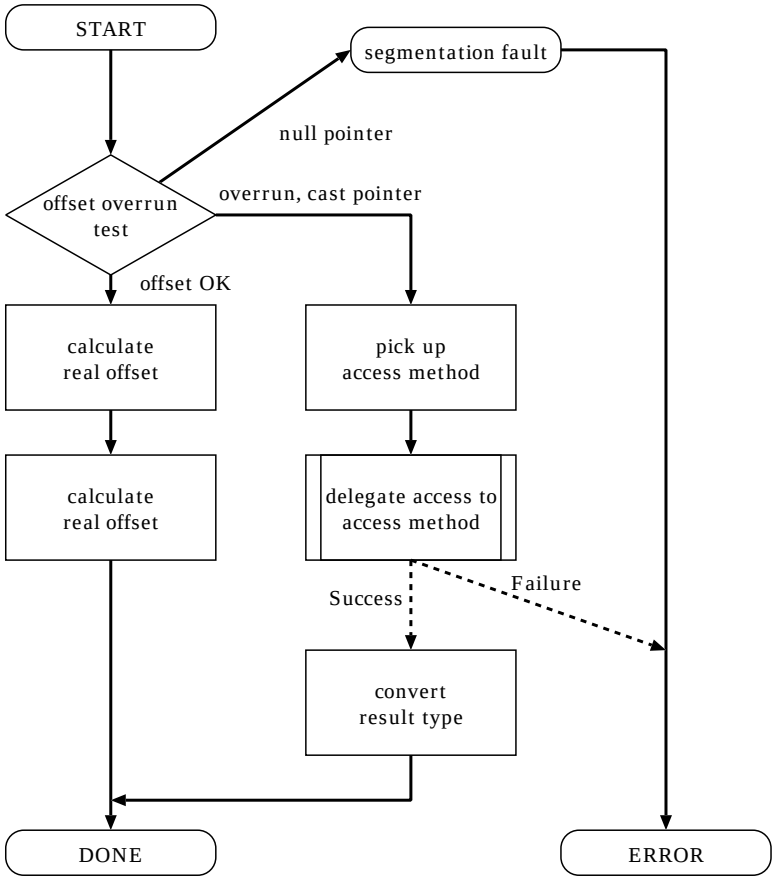


Figure 4.6: Procedure for memory access via pointers with fast access check

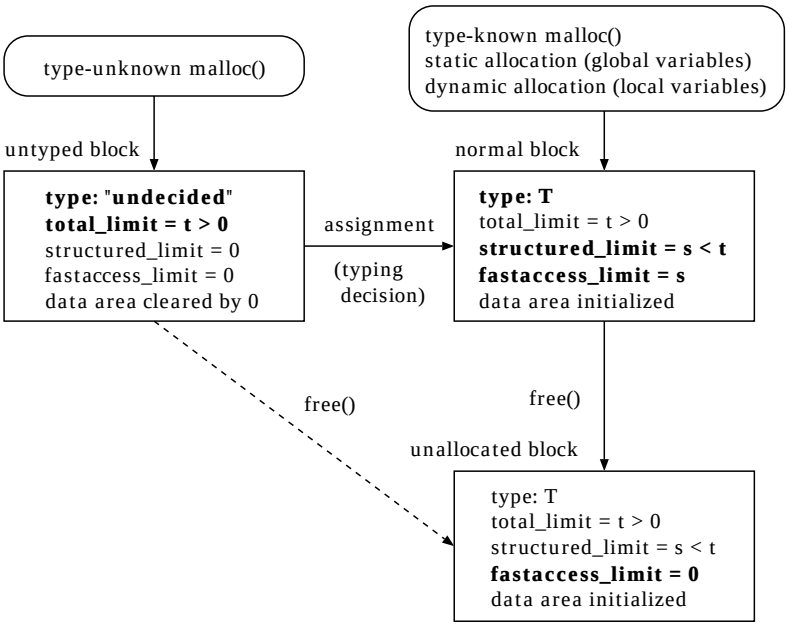


Figure 4.7: State diagram for blocks

gated 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 an additional argument to the methods (See Section A.1.2). After initialization, it limit values and typeinfo field of the block's header are reinitialized to make the block a normal block. Finally, the method handles the write request from a caller by delegating it to the newly-associated access methods (Figure 4.7).

Obviously, it is usually unsafe to change the block type and its limit values during program execution. If two or more pointers points to one block, changing its block type will cause type inconsistency. However, regarding type-undecided blocks, this whole process is a safe operation, because the "type-undecided" pseudo-type does not appear in the program as a static type, thus all pointers referring to a type-undecided block must have cast flag = 1.

There is a partly-unresolved problems related to type-undecided blocks. This delayed-typing mechanism leads to the generation of too many pointers with the cast flag set to 1, because there is no chance to remove the cast flag from a pointer which has pointed to the block being initialized. The cast flag is retained as set until the pointer reaches to some explicit cast operation in the user programs. Currently, the Fail-Safe C compiler inserts ad hoc checks and additional operations to remove redundant cast flags (the same as those in cast operations) before every invocation of access methods in generated code. In addition to this, the compiler tries to generate program code which let several distinct pointers in a function to share

the base part of a fat pointer, to make this optimization more effective. However, because the compiler uses a static-single-assignment form for the intermediate representation of programs. not all instances of the same pointer will always have a redundant cast flag removed, and the extent of the effect of redundant flag removal may depend on the internal representation of programs in the compiler. Regardless, the ad hoc nature of these checks does not have affect safety.[5] A Possible alternative solution is to find pointers which may point to type-undecided blocks through an analysis (e.g. type analysis), and then insert checks at more appropriate points.

I also plan to implement an algorithm to guess the intended type of a block by analyzing a cast expression whose operand is the return value of the `malloc()` function.[6] The guessed type is passed as a hidden argument to the function. Furthermore, not only `malloc()` is made special: all functions returning a value of "`void *`" type can be specially handled. Inside such user-defined functions, the passed type information may be either ignored, or passed to another function returning `void *` type (including `malloc`). This extension is designed to support frequently-implemented small wrappers to `malloc`, that serve in the same way as `malloc`, but if allocation fails these terminate the program instead of returning `NULL` to callers.

4.4 Interfacing with external libraries

Almost all C programs uses externally defined routines to accomplish their task. These routines include system calls for low-level interaction with operating systems, standard library routines for file input/output, mathematical operations and memory allocation, or other high-level libraries such as GUI, database access, or network communications. Fail-Safe C must support communication with these external routines.

One possible way to provide this functionality is to compile these libraries with the Fail-Safe C compiler along with user programs. However, this method has three drawbacks:

- Source codes (which run with user-level privilege) are needed to compile the library with Fail-Safe C. This cannot be done for either closed-source libraries or system calls.
- The generated code incurs performance overhead due to the additional safety checking done by the Fail-Safe C system. It might be beneficial to optimize frequently called routines, though, to reduce execution overhead.

[5] (Future static analysis (Section A.5.1) must take this optimization into account to maintain safety.

[6] The extension can be implemented alone, but because program analysis required for local optimization (Section A.5.1) subsumes that for this extension, I plan to implement the extension at the same time as other local optimizations.

Thus Fail-Safe C takes another approach. A set of standard library routines which can be called from the program code generated by Fail-Safe C is implemented in native C language. These routines are usually called wrapper routines, because they often uses corresponding functions in the native version of the library internally; i.e., they "wrap" the original function by adding interface code before and after it.

4.4.1 Generic structure of wrappers

Wrapper routines have two main purposes. The first one is to ensure the safety condition required by Fail-Safe C is satisfied even after the invocation of native routines. For example, calling the `read` system call with an insufficient buffer instantly breaks any data structure on the memory beyond the buffer. To ensure safe execution of a program on Fail-Safe C, the wrapper routine must check that the length of the operation, which is passed to the wrapper as another argument, must be smaller than the available number of bytes in the memory block containing the buffer. Sometimes there is no condition that can guarantee safe execution of a native routine in any case: for example, the `gets` library function may fail no matter how large a buffer is provided to the function. Such a case cannot be handled through a simple wrapper function.

The second purpose is to convert data formats between Fail-Safe C and native routines. Because the representations of data in Fail-Safe C differ from those expected by native library routines, the data in a Fail-Safe C program must be converted by wrapper routines before being passed to native libraries. The data returned from a native function must also be converted to the Fail-Safe C representation by wrappers.

Thus, the general structure of a wrapper routines follows a sequence something like the following.

1. Check safety preconditions, especially regarding buffer lengths.
2. Convert input data to the format accepted by the native routine.
3. Call the native function.
4. Convert output data of the native function back to the Fail-Safe C representation.

Unfortunately, there is no single universal method for such a conversion. For some functions, there is no appropriate map at all. Back-conversion to the Fail-Safe C representation tends to be especially difficult because so much information is lost during the first conversion, and it is difficult to guess what data structure native routines expect when pointer aliasing (equivalence) is important to the library.

At the same time, however, there are a few common patterns of conversion which can be applied to the arguments of many functions. Here, I categorize the arguments of external routines into three kinds.

Raw values: the first category holds values having only self-contained structures, mainly from the perspective of the pointer's use. All integers and floating arguments are generally of this kind. The values used as descriptors are also placed in this category, although they are actually an index to other array-like data.

Many pointer values also fall into this category, especially those for system calls. This is not just a coincidence: the pointers passed to system calls are only used while the system call is active, and are not used afterwards, because trusting some well-formedness of the user-space data structure while running a user program in parallel is generally an unacceptable option for ensuring safety of the kernel state. In addition, there is no pointer returned from system calls pointing to the kernel space, for semi obvious reason.

Raw values are generally handled by through data-copying inside wrapper routines, as described in the next section.

Abstract values: the second category contains pointers which are only valid as abstract values. A file pointer (`FILE *`) is a good example from this category. Many high-level libraries, including GUI libraries, numerical libraries, or cryptographic libraries, use this kind of value to simplify the interface between user program and libraries, and to enable internal change of the data structure for any improvements while preserving user-level compatibility and portability.

Abstract values are encoded using the *abstract data implementation* described in Section 4.4.3.

Complex values: the third category is for values which cannot be categorized into the first two categories. This category of values allows access inside its internal data structure or those of pointer targets, and also cannot be easily moved around memory by a user program because of pointer aliasing (data which are pointed to by another pointer kept inside the library). At least one instance exists: some data structures in Xlib library allow reading of some fields of the data structure. Wrappers for functions with this kind of arguments are generally hard to implement.

One way to work around complex values is to compile a library with Fail-Safe C compiler. There are features which provides support for safe separate compilation of libraries. The compiler accepts a language extension to give a fixed name to the encoded name of a structure (see Section A.2.2). Also, a few extended attributes are defined by Fail-Safe C to control the generation of various internally generated subroutines to prevent these routines being generated twice through separate compilation. They also let library programmers to implement customized versions of access methods, instead of automatically generated routines (see Section 4.4.4).

4.4.2 Handling raw data in wrappers

The handling of raw arguments (and return values) is relatively simple, because these types of arguments allow the copying of data.

For simple data types, there are common patterns regarding the use of buffers. For example, some of common usage patterns for `char *` type include (but are not limited to) the following:

- Read access:
  - NUL-terminated strings of unlimited length (many functions)
  - NUL-terminated strings with a length limit provided by another integer argument (printf `"%.80s"`)
  - byte arrays whose sizes are provided by another integer arguments (write, fwrite, etc.)
- Write access
  - byte arrays with a access length limited by another integer argument (read, fread, etc.)
  - byte arrays with an unlimited access length (gets, scanf)

Note that some patterns (e.g., the last pattern in the above list) must be handled a the way other than the copy-invoke-writeback approach, because there are no preconditions which satisfy the safety requirement for all possible inputs. These functions are "insecure" by nature, because however large the temporary buffer allocated for accepting input data, these functions can cause buffer overflow if a huge amount of input data is provided. The wrapper routines for these functions must be implemented on a one-by-one basis with carefully-inserted boundary checks for output. For some other patterns, Fail-Safe C runtime provides several support routines for writing wrappers using such common patterns.

The copying of the arguments is only required when the representations of the arguments differ from native representations. As many input/output primitive functions (and system calls) take pointer arguments to byte arrays, avoiding to copy arguments of `char *` types is important to improve performance. The implementation of wrapper support subroutines checks the *continuous* flag in the type information on the memory block of arguments and omits the copying if possible.[7]

For example, the interface for the helper function for a NUL-terminated string is defined as follows:

```
char *wrapper_get_string_z(base_t b, ofs_t o,
                           void **_to_discard,
                           const char *libloc);
```

[7] A possible way to improve this optimization is to include these subroutines in the access methods, and to allow direct use of native representation data inside structures.

```
value FS_FPc_i_puts(base_t base0, ofs_t ofs)
{
    void *tb0 = NULL;

    char *p0 = wrapper_get_string_z(base0, ofs, &tb0, "puts");
    int r = puts(p0);
    if (tb0) {
        wrapper_release_tmpbuf(tb0);
    }
    return value_of_base_vaddr(0, r);
}
```

Figure 4.8: Wrapper for `puts` library function.

The first two arguments are the fat pointer from the user program. The third argument is a pointer to the pointer variable that receives an the address of a block which should be deallocated before returning from a wrapper function. If the block referred to by *b* is *continuous*, the address of the element at offset *o* in block *b* is directly returned, and `NULL` is written to `*_to_discard`. Otherwise, the data starting from virtual offset *o* in block *b* is converted to a native representation and copied to a newly allocated temporary buffer. The address of the copied data is returned, and the address of the temporary buffer is written to `*_to_discard`. In both cases, the program is halted if the string is not terminated by NUL before reaching the boundary of the memory block. Before exiting from the wrapper, the temporarily allocated buffer must be deallocated. As a special case, there is a set of functions which performs only write operations to memory blocks (e.g., `read` and `fread`). For these functions, the contents of the original memory block do not need to be copied to the temporary buffer. Using this helper, the wrapper for `puts`, for example, can be implemented simply as is shown in Figure 4.8.

If an original function only reads the contents of buffers, the function in the runtime library `wrapper_release_tmpbuf` should be called with the value of `*_to_discard` if it is not `NULL`. The allocated temporary buffer is deallocated through this helper function. If an original function writes to or updates the contents of the buffer, the update must be propagated to the original memory block. Another helper function `wrapper_writeback_release_tmpbuf` receives the original fat pointer (*b*, *o*) and the address of the temporary buffer (`*_to_discard`), along with an argument specifying the length of the overwritten area (e.g. for the `read` system call, it would be the value returned from the original function), and writes the contents in the temporary buffer into the original memory block with converting the representations.

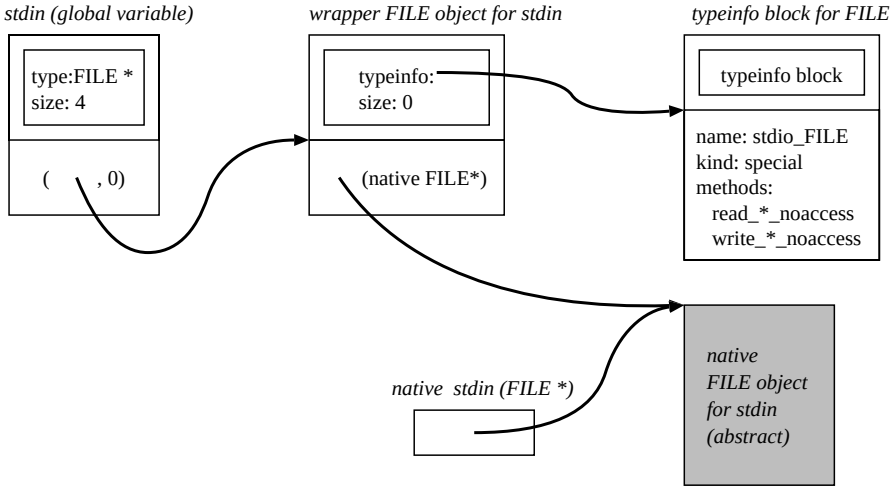


Figure 4.9: Implementation of FILE object in Fail-Safe C

4.4.3 Implementing abstract types

There are some types in the standard C library (e.g., `FILE` type) whose internal structures are not exposed to user programs. Instead of implementing complex conversion routines and safety checking for every implementation of systems, simply providing an abstract interface for such data types is both sufficient and secure, because it further prevents any accidental modification inside such data which should not be touched by user program in any way. Fail-Safe C supports this kind of library interfaces through *abstract type* mechanism.

Figure 4.9 illustrates an implementation structure for such a type (`FILE` is used as an example). To define a new abstract type, firstly we should create a type information block corresponding to the type. All memory accesses to the contents of abstract data should be forbidden by the access methods for the abstract type. Next, we declare that type as an opaque structure inside header files, with an extension keyword `named` to fix its encoded name. In the case of type `FILE` in the current implementation, it is the type `struct FILE` with keyword `stdio_FILE` used for fixed type encoding. Finally, we allocate corresponding memory blocks either statically or dynamically through some externally defined library routines. Because the types of those blocks are opaque to user programs, and their access methods prevent access via cast pointer, the whole data area inside the blocks can be used in an arbitrary way by wrapper routines. For example, a wrapper object for `FILE` type contains a native `FILE` pointer, or `NULL` if the corresponding native `FILE` is already closed. An example code for abstract type implementation is included in Section A.3.

Every library routine has to decode the structure described above before using

its value. To avoid confusing other kinds of value as an abstract data object, the routines should first compare the type of the block against the type information block of the expected type. In addition, whether the offset value of the pointer is zero should be checked.[8] If these checks are successful, the library routines can take values from inside the data area of the block in a way that each library defines for its own purposes.

4.4.4 Implementing magical memory blocks

The method described above can be further extended. For example, the `errno` variable in the standard C library can change after the invocation of many library routines. One way to pass the value of such a special variable to user programs from native libraries is to separately defines a variable which is referred to from user programs, and updates it through wrapper routines whenever native library routines update it. Such an implementation and language support for the insertion of program code for this sort of updating was recently proposed [67]. However, this may be too cumbersome, especially when a library wrapper must be written by hand, or when the timing of the update is complex or difficult to guess. Also, when Fail-Safe C supports multi-threading in the future, it will become especially difficult because `errno` is defined as a thread-local assignable identifier (it can be either a variable or a macro).

These problems can be solved through an extension of the implementation of abstract types described in the previous section. Instead of putting access methods which forbids all accesses, specially implemented access methods can be attached for such abstract types. Each of these will then work as a “magical” hook for memory access to those memory regions. For example, read access methods for the memory block for `errno` variable can read the native `errno` variable instead of the data inside the memory block. Updating `errno` (resetting it to 0 is a common practice) can also be forwarded to the native `errno` by the corresponding write access methods. An example implementation is shown in Section A.3.

This method is also useful if a data type which is almost abstract (i.e., only allocated by a small set of dedicated functions) must allow some trivial access to fields. For such a data type, the library programmer can define a “virtual” struct for the data structure in which the fields accessed a from user program are defined. The allocation routines for those data returns a cast fat pointer to an instance of the magical data type. All accesses to the defined fields are then forwarded to the access methods of the magical type, where any kind of emulations of the behavior can be done.

[8] Although ignoring the offset is completely safe, it is unnatural compared to native semantics.

Chapter 5

Experiments

5.1 Examples of memory overrun detection

This section describes some examples of access overrun that occur in several programs and shows that Fail-Safe C can detect such problems before they can cause memory corruption or allow program invasion.

5.1.1 Integer overflow in the command-line argument parsing routine of Sendmail

Sendmail [64] is the one of the most widely used Internet mail server programs. The versions between 8.11.0 and 8.11.5 of Sendmail had a critical security hole in the parsing routine of the debug option, which is called at a very early stage of program execution [63, 21]. The cause of this security hole is was that it did not correctly treat overflow condition for integer variables, which is often referred to as an "integer overflow" security hole. This kind of security hole differs from a simple buffer overflow (where the memory area immediately after a buffer is sequentially overwritten) in that it directly overwrites the very specific bytes or words of the memory area using variables located far from the victim memory area. This implies the following points of differences with respect to countermeasures:

1. It cannot be prevented through canary techniques, which detect memory corruption by checking the memory area immediately after the buffer boundary.

2. The array used for an attack does not need to be in the stack area. In fact, an attack on the Sendmail program uses globally allocated array to attack the instruction pointer stored in the stack memory.

The cause of this problem lies in the `tTflag` function (Figure 5.1) in `trace.c`: this function receives a string formatted like "`12-17.5X18-19.7`" and writes a value after a period to the bytes in the range specified before the period in the global array `tTvect`. In the above example, it write six 5's to the area from `tTvect[12]` to `tTvect[17]` and two 2's to `tTvect[18]` and `tTvect[19]`. Unfortunately, the

integer parsing routine at lines 14–26 does not care about integer overflow beyond 2^{31}, thus the values in variables `first` and `last` can be negative. At lines 38–41, an overflow condition is checked and rounded to the possible maximal value, but an *underflow* condition is not checked. As a consequence, the assignment in line 45 overwrites an unexpected byte with a huge negative offset, and this is used for an attack.

An exploit code for this security hole to gain root privileges is well-known and available on the Internet. As an experiment, I took the unmodified source file of trace.c (112 lines in total), and combined this with a small main routine which invokes the problematic functions in the way the original Sendmail program did. Thus, the same way of exploiting the hole can be used for attacking this test program with only a small amount of modification to the offset value, which is the offset between the overflowing array and the instruction pointer in the stack area of a running program.[1] The experiment was done on a machine running Linux 2.4.22 on a Pentium-III processor. [2]

Figure 5.2 shows the output generated by a target program compiled by the Fail-Safe C compiler that was executed with an argument to exploit the bug. The first few lines were generated by the attacker program calculating proper values for activating the security hole. The messages between the two rulers were are generated by Fail-Safe C runtime. It shows that the program accessed the byte at offset 3086701108, which is a the negative value -1208266188 in signed integer type, of an array of 100 characters. The same value is also appeared in the output from the attacker program and in the command line passed to the target program. The block status field had `no_dealloc` flag, which means the overflowed array was statically allocated as a global variable. The backtrace is a little hard to decode, but says that the error is occurred inside function `tTflag(char *)` (the fourth line has an encoded name of the function).

5.1.2 Buffer overflow in a GIF decode routine in XV

XV (version 3.10a) is a famous shareware program that displays files of various graphics formats, including GIF and JPEG, for display on X window system environments. It was written before 1994 and is no longer maintained. It has an its own implementation of a GIF decode routine, which was also used for many other

[1] This modification to the exploit code was provided by Dr. Yoshihiro Oyama.

[2] This setting is different from all other experiments. The main reason for this is that Linux kernel version 2.4.22 configured for a symmetric multi-processor architecture with an Intel CPU changes the starting value of the stack pointer for each program execution to avoid overwrapping of the stack addresses which causes contention on cache lines in a Hyper-Threading (a simultaneous multi-threading) architecture. Simple stack buffer overflows are basically unaffected by this behavior, but, interestingly, it make the exploitation of the Sendmail security hole slightly difficult because the address difference between `tTvect` and the stack area changes for each execution. The behavior of the stack movement is almost completely predictable, though, so writing an exploit program assuming this behavior is not very difficult. For this experiment, however, to avoid complexity I used a single CPU environment.

```
1   void
2   tTflag(s)
3           register char *s;
4   {
5           int first, last;
6           register unsigned int i;
7
8           if (*s == '\0')
9                   s = DefFlags;
10
11          for (;;)
12          {
13                  /* find first flag to set */
14                  i = 0;
15                  while (isascii(*s) && isdigit(*s))
16                          i = i * 10 + (*s++ - '0');
17                  first = i;
18
19                  /* find last flag to set */
20                  if (*s == '-')
21                  {
22                          i = 0;
23                          while (isascii(*++s) && isdigit(*s))
24                                  i = i * 10 + (*s - '0');
25                  }
26                  last = i;
27
28                  /* find the level to set it to */
29                  i = 1;
30                  if (*s == '.')
31                  {
32                          i = 0;
33                          while (isascii(*++s) && isdigit(*s))
34                                  i = i * 10 + (*s - '0');
35                  }
36
37                  /* clean up args */
38                  if (first >= tTsize)
39                          first = tTsize - 1;
40                  if (last >= tTsize)
41                          last = tTsize - 1;
42
43                  /* set the flags */
44                  while (first <= last)
45                          tTvect[first++] = i;
46
47                  /* more arguments? */
48                  if (*s++ == '\0')
49                          return;
50          }
51  }
```

Figure 5.1: A routine containing a security hole in the Sendmail program

```
distance from b7fb511c to b7fb5234
jump_target:[0xbfffde80]
I will overwrite 128 (80) to tTvect[3086701108 (b7fb5234)]
I will overwrite 222 (de) to tTvect[3086701109 (b7fb5235)]
I will overwrite 255 (ff) to tTvect[3086701110 (b7fb5236)]
I will overwrite 191 (bf) to tTvect[3086701111 (b7fb5237)]
calling execv("./kiridasi_sendmail.safe", ["./kiridasi_sendmail.safe",
"-d3086701108-3086701108.128X3086701109-3086701109.222X3086701110-308
6701110.255X3086701111-3086701111.191", "", NULL])

--------------------------------
Fail-Safe C trap: access out of bounds
  Address: 0x804d520 + 3086701108
  Cast Flag: not set
  Region's type: char
           size: 100 (FA 100, ST 100)
           block status: normal, no_user_dealloc, no_dealloc

backtrace of instrumented code:
./kiridasi_sendmail.safe(fsc_raise_error_library+0x14e)[0x804b7e6]
./kiridasi_sendmail.safe[0x804b83e]
./kiridasi_sendmail.safe(write_byte_continuous+0x22)[0x804b356]
./kiridasi_sendmail.safe(FS_FPc_v_tTflag+0x3ce)[0x804a006]
./kiridasi_sendmail.safe(FG_main+0xd3)[0x804a253]
./kiridasi_sendmail.safe(main+0xaa)[0x804a5aa]
/lib/libc.so.6(__libc_start_main+0xbb)[0x4006614f]
./kiridasi_sendmail.safe(free+0x61)[0x8049a11]
(8 entries)
--------------------------------

Abort
```

Figure 5.2: An error detection report for an attempt to exploit the Sendmail security hole

programs, but it has a buffer overrun bug which becomes apparent through corrupted input files. The decode routine exists in `xvgif.c` (768 lines) in a mostly self-contained fashion; this is derived from an implementation written in 1989 by Patrick J. Naughton according to comments in the source file. I wrote a stub routine to call the `LoadGIF` function and combined this with `xvgif.c` to make a stand-alone command-line application. A large GIF file (443792 bytes) is was then intentionally truncated to various random sizes and fed into the program to obtain an instance of an input file causing a buffer-overrun condition. Eventually, an instance of 109538 bytes was found to cause a buffer overflow. (Of course this was strongly dependent on the original file.) This instance caused a segmentation fault in the natively compiled program, and a runtime error (Figure 5.3) was issued in the program compiled with the Fail-Safe C compiler. The message suggested that the type of the access violation seems to be was simple sequential buffer overrun (the failed offset (109704) matched the size of the memory area).

Further experiments on the program revealed that for this input data an overflowed read access occurs at up to the address 4039 bytes beyond the end of the input file at maximal. The implementer prepared a 256-byte redundant memory to avoid buffer overruns (Figure 5.4), but this seems to have been insufficient and was not a good way to avoid buffer overruns. This overflow is occurred during the reading of memory, so it is unknown whether it is directly exploitable, except for denial of service attacks.

5.2 BYTEmark benchmark test

BYTEmark [12] is a set of ten synthesized benchmark programs which is firstly proposed by BYTE magazine. I used seven of provided tests provided in the version 2 of BYTEmark (originally released in 1995) to evaluate the overall performance of the Fail-Safe C system. To perform these tests, the following changes were made on a Linux port of BYTEmark by Uwe F. Mayer [43].

- Three tests were only slightly modified to avoid features not implemented in the current Fail-Safe C compiler. These tests are shown under the horizontal rule in Table 5.1.

  The sources for seven other tests as well as core parts of the program sources were *not* modified at all, except for the one additional evaluation discussed later.

- The Makefile was replaced with my own version, as the current compiler driver interface differs from that of conventional compilers.

- A declaration mismatch bug between two source files was corrected.

- The address alignment option in the benchmark is disabled for the Fail-Safe C test, as the method of forcing address alignments in original BYTEmark is

```
--------------------------------
Fail-Safe C trap: access out of bounds
  Address: 0x80d6020 + 109794
  Cast Flag: not set
  Region's type: char
           size: 109794 (FA 109794, ST 109794)
           block status: normal

backtrace of instrumented code:
./xvgif.safe(fsc_raise_error_library+0x15f)[0x8062f27]
./xvgif.safe[0x8062f7e]
./xvgif.safe(read_byte_continuous+0x1f)[0x80625fb]
./xvgif.safe[0x805f4f0]
./xvgif.safe[0x805ea3d]
./xvgif.safe(FS_FPcPS2_i_LoadGIF+0x2430)[0x805d344]
./xvgif.safe(FS_FiPPc_i_main+0x1e1)[0x805acd1]
./xvgif.safe(FG_main+0x74)[0x805aeb8]
./xvgif.safe(main+0xac)[0x806023c]
/lib/libc.so.6(__libc_start_main+0xbb)[0x4006614f]
(10 entries)
--------------------------------
```

Figure 5.3: An error detection report for the XV GIF decoder

```
133    /* the +256's are so we can read truncated GIF files without fear of
134       segmentation violation */
135    if (!(dataptr = RawGIF = (byte *) calloc((size_t) filesize+256, (size_t) 1)))
136      return( gifError(pinfo, "not enough memory to read gif file") );
137
138    if (!(Raster = (byte *) calloc((size_t) filesize+256,(size_t) 1)))
139      return( gifError(pinfo, "not enough memory to read gif file") );
```

Figure 5.4: A failed attempt to avoid buffer overflow in the original `xvgif.c`

Table 5.1: Results of BYTEmark benchmark tests

Test	Native	Fail-Safe C	Ratio	(Typed)	Ratio
Numeric Sort	930.96	361.36	2.593		
String Sort	87.045	68.158	1.277		
Bitfield	362.32 M	114.43 M	3.166		
Fourier	13214	11649	1.134		
IDEA	1679.3	1576.8	1.065		
Huffman	1204	119.52	10.074	217.39	5.538
Neural Net	22.435	6.1386	3.660		
FP Emulation	83.134	14.031	5.925	14.45	5.753
Assignment	18.436	5.4496	2.182		
LU Decomp.	1088.9	—	—	271.28	4.014

(Unit: iterations per second, M denotes 10^6)

incompatible with the strict ANSI-C semantics enforced by Fail-Safe C. The default parameter of 8-byte address alignment is still used for the native code evaluation, because it is the same alignment as that provided by the memory allocator of the current Fail-Safe C implementation.

The experiment was done on a workstation running Linux 2.4.27 on a 2.8 GHz Pentium-4 processor with 1 GB main memory.

The test results are shown in Table 5.1. There was no more than 30% of overhead observed for the String sort, Fourier, or IDEA tests. The execution speed on Numeric Sort, Bitfield, Neural Net, and Assignment tests are about 2 to 3.66 times slower than the native program.

The Huffman test was exceptionally slow compared to the other tests. Furthermore, the execution time for the LU Decomp test does not converge. (BYTEmark tries to acquire a statically reliable result by repeating the test until score converges.) I have inspected the behavior of the translated program for the Huffman test and found the main reason for this was a conflict between the handling of type-undecided blocks returned by the untyped `malloc()` function (Section 4.3) and the integer overflow handling required at pointer arithmetic (Section A.2.3.2 for details). In the Huffman test, an array of a 24-byte struct was allocated and heavily used inside the test, while in the other six tests only primitive types are used heavily. In the current implementation which uses lazy type decisions for all `malloc`'ed blocks, the cast flags of pointers returned from `malloc()` are always set at the first time, and will be removed when the pointer is dereferenced twice[3]. If an array of primitive types (or a struct with the a size of some power of 2) is used frequently inside one function, one base value is shared among all fat point-

[3] At the first dereference the block type is decided, and then at the second dereference the pointer, type matches the type of its referring block

Table 5.2: Results of tests with fast check disabled

	w/fast check	w/o fast check	gain
fib	2.339 s	2.339 s	+0.0%
qsort	2.255 s	2.737 s	+20.8%
qsort (cast)	8.144 s	8.158 s	+0.2%
knapsack	1.076 s	1.118 s	+3.9%

(the average of 5 trials is taken)

ers referencing that block. Thus once the cast flag is removed from one of these pointers, memory accesses via all of those pointers will become faster. However, if an "odd"-sized structure is involved, it is impossible to share the base part of fat pointers between many pointers (because of the integer overflow described in Section A.2.3.2), so the effect of the ad hoc cast flag removal in Section 4.3 decreases. Indeed, a huge number of invocations of the access methods for the involving struct type was observed in the Huffman test.

To avoid this problem, an additional experiment was done where a type annotation was added to the memory allocation code inside the Huffman test. The result with the modified source code is shown in the column "Typed" in Table 5.1. Although the overhead remained slightly larger than in the other eight tests, overhead than other six tests, the performance was greatly improved. Fortunately, the type of the allocations can be easily guessed through the algorithm proposed in Section 4.3, and so the future versions of Fail-Safe C should be able to achieve the improved performance without modification of the source code. Overall performance (with typed memory allocation) is very promising, even under the fact that no optimization have been performed yet.

5.3 Effectiveness of fast cast-flag checking

To check the performance gain obtained through fast cast-flag checking (Section 4.2), the internal logic of the compiler that checks the possibility of omitting cast-flag checking was intentionally blocked, and applied to a set of small test programs. The results are shown in Table 5.2.

These results show noticeable differences between several tests, which roughly correspond to the number of direct memory accesses in the program. For a Fibonacci test there was absolutely no difference in the output code, and the test on quick-sorting test with a cast pointer showed only a small gain (possibly due to the omission of null-pointer checks), that was mostly covered up by the overhead of the access method invocation. The result from the knapsack test showed a moderate performance improvement, while the normal quick-sorting showed a significant improvement.

5.4 Other preliminary tests

In addition to the tests above, some other preliminary tests are performed as well. The descriptions for these tests are in the appendix.

- A micro-benchmark test for deciding the representation of fat pointer and fat integer encoded to C program (described in Section A.4).
- A preliminary test which evaluates the possible gain for future local optimizations (described in Section A.5.1).

Chapter 6

Conclusion and Future Work

6.1 Summary of the dissertation

This dissertation has proposed Fail-Safe C, a method for implementing the full-set of the ANSI C language in a memory-safe way. The system accepts all programs that conform to the specification of the ANSI C language as well as many existing programs that deviated slightly from the specification, while ensuring that no memory corruption which could lead to execution hijacking or other security holes will occur. Fail-Safe C uses two-word representation for every pointer in programs and object-oriented representation for every memory block to ensure safety and correct behavior on cast operations. The representation of memory blocks in Fail-Safe C is so powerful that it can support various tricky operations on memory data in C programs such as variable-length structures and cast-based implementation for variant types. The system also introduces a flag in every pointers which indicates whether the pointer is cast, and when a pointer is not cast, the system avoids additional overhead incurred from cast support by using sophisticated representations for both memory blocks and cast flags.

In addition, an implementation supporting most of the features of the Fail-Safe C system has been described. The implementation incorporated several optimization techniques proposed in this dissertation to enable efficient implementation of Fail-Safe C.

It has been demonstrated that the Fail-Safe C system can prevent some real-world security holes from being exploited by correctly halting the execution of flawed programs. Benchmarking tests have shown that the execution speed of the programs compiled by the current Fail-Safe C compiler was about 30% to five times slower than the original C programs in most cases. These figures are roughly comparable to many other safe languages that have been developed.

The whole system was carefully designed to provide provable safety, and a brief outline of the safety proofs was given in this dissertation. However, the complete proofs have been left as future work.

6.2 Relation to other work

A number of systems have been designed to make C programs safe in various ways. These systems were briefly compared with Fail-Safe C in Section 2.2. Although these systems are useful in practice to prevent some of existing security attacks, most of them are incomplete with regared to either safety [5, 53, 36, 41, 20] or compatibility [27, 35].

CCured [49, 18] is the only system, except for Fail-Safe C, that I know of which can provide a sound semantics for a large part of C language, including cast operations. There are several differences between CCured and Fail-Safe C, but the most noticeable technical difference is in the handling of pointer cast operations: CCured is almost completely based on static analysis, while Fail-Safe C is mainly based on dynamic handling. In other words, CCured statically determines which variables might have a cast pointer, and "quarantines" those wild parts from the other pure parts in programs. The pure part of a program will then behave almost like the programs of pure statically-typed languages; e.g., there will be no type information inside. The weakness of this method is that the system cannot allow any pointers in wilds part to point to values in the pure parts of programs. In addition, as value types are completely determined statically, a pointer which may point to wild values must always point to wild values; conversely, a data which may be pointed to by such pointers must be moved to a wild part, even if that value is used completely in a type-safe way. The relative size of the wild part in a program is therefore likely to increase as program gets larger. Because Fail-Safe C is based on dynamic determination of cast pointers, and because it allows every pointer to refer to both cast values and well-typed values, no such chain of wildness pollution will occur.

The other differences between the two systems can be summarized as follows.

- CCured has basically two kinds of type-safe pointer beside a single kind of wild pointer: one-word "safe" pointer for values which is not the targets of pointer arithmetic, and "seq(-uential)" pointers which are similar to the fat pointers in Fail-Safe C. The "seq" pointers in CCured uses three-word representation which remembers both the head and the tail of the region accessible from the pointer. The advantage of this representation is that it can safely point to an array inside structures, which the uncast fat pointers in Fail-Safe C cannot. The downside is that it consumes a lot of register resources and memory resources.

- Fail-Safe C introduces a notion of virtual offsets, which enables effective and complete concealment of any internal data for safety management completely from user programs. In Fail-Safe C, the areas for base values and other values are completely separated statically, although this does not confuse user programs in any way. As CCured uses only native offsets, the handling of wild values in a heap area is very complicated, as it requires an

additional bit array to remember whether each (native) word in a memory block holds a value valid as the base address of a block.

- Moreover, the virtual offsets hide even the fact that a Fail-Safe C compiler is used. The offsets visible to user programs are always identical to those used in the usual native compilers.

  On the contrary, CCured reveals representation change of pointers to user programs: the sizes and the offsets of pointer data are those of the internal representations, and thus these values are different from the native values. Furthermore, the sizes of pointers will differ depending on the classification of pointer usages by CCured, even if these have the same type in original C programs. Under 32-bit architectures, safe pointers are 4 bytes, seq pointers are 12 bytes, and wild pointers are 8 bytes. This fact confuses some programs and programmers, and requires program rewriting (to associate every memory allocation to the size of the target variable, not to the type of targets).

- The design of the Fail-Safe C system makes support for separate compilation easier than that on CCured. If the value range of a function argument is unknown because of separete compilation, the compiler must assume every possible value. It forces the type of the argument to be "wild" or "may be cast" in both CCured and Fail-Safe C, respectively. However, the former has severe penalty for both compatibility and performance, although the latter has (practically) little overhead.

The safe pointers in CCured, which can point to an internal element as well as to the top of a block, greatly reduced the execution overhead of CCured. By collapsing two memory accesses using the same seq pointer (which require boundary checks twice) into a cast to a safe pointer and two accesses using it (which require boundary check only once), CCured has optimizing out overhead on many redundant boundary checks. Benchmark programs compiled in CCured run less than twice slower than the natively compiled programs. (although the result cannot be compared directly because of the difference on expressiveness).

The static method used in CCured may also be used in Fail-Safe C for optimization. In particular, the CCured's type system built on C language can be modified and applied to the Fail-Safe C system for the global analysis required for optimizations.

6.3 Future Work

Current Fail-Safe C uses little information about static properties of programs. There are many forms of static analysis for program behavior (more details are discussed in Section A.5). These can be used in combination with Fail-Safe C system for optimization. In general, the safety property of C programs cannot be

wholly guaranteed in a static way; thus, both CCured and Fail-Safe C (and many other systems) use dynamic checking. Static analysis is very valuable for those dynamic systems to reduce the runtime overhead introduced by runtime checks. However, the incorporation must be done carefully to avoid opening new security holes due to such optimization, and to work well in the access methods introduced in Fail-Safe C (see Section A.5.1).

The combination of Fail-Safe C with an operating system that relies wholly on type-based safety management (e.g., [7, 42]) is an interesting possibility, as it will allow many existing programs which are written in C to run on such operating system architectures. To ensure the safety of these systems under existence of externally provided programs, so called certifying compiler technique can be useful in combination with the Fail-Safe C system (Section 3.6.4).

Current Fail-Safe C implementation accepts the programs which strictly conform to genuine specification of ANSI-C. Extending the input language (e.g., to accept the C++ language), or extending the dynamic behavior of the programs compiled by Fail-Safe C (e.g., to remedy programs with buffer-overrun problems by internally extending buffers on the fly) might bring us some useful systems. Some perspectives of future research are discussed in Appendix B in more detail.

Appendix A

Implementation Details

This appendix describes the details of the implementation of Fail-Safe C. First, the organization and behavior of runtime system routines are described. Descriptions of the code generated by the Fail-Safe C compiler follow with example fragments of the generated code shown for explanation. These two areas are closely related and in cooperation ensure the safe execution of programs in cooperation.

A.1 Runtime system

A.1.1 Structures inside memory blocks

A.1.1.1 Common structure and block header

A memory block is an atomic unit for memory management in Fail-Safe C. All blocks (both in statically-allocated area and in dynamically-allocated heap area) which may be referred to by any pointers have appropriate block headers to maintain proper boundary checking. Figure A.1 shows the structure of a memory block and its associated block header. All blocks are double-word aligned (by using GCC's `__aligned__` extension) (Section 4.2).

A block header contains the following fields.

typeinfo_ptr A pointer to the type information block described later. This field determines the storage format of the structured data area of the associated block.

runtime_flags A set of flags about runtime information of the block. The following information is stored:

- Kind: a kind of block, one of the following: a normal block, an active block for passing varargs, a finished block for passing varargs (`va_end` called), and a released block.

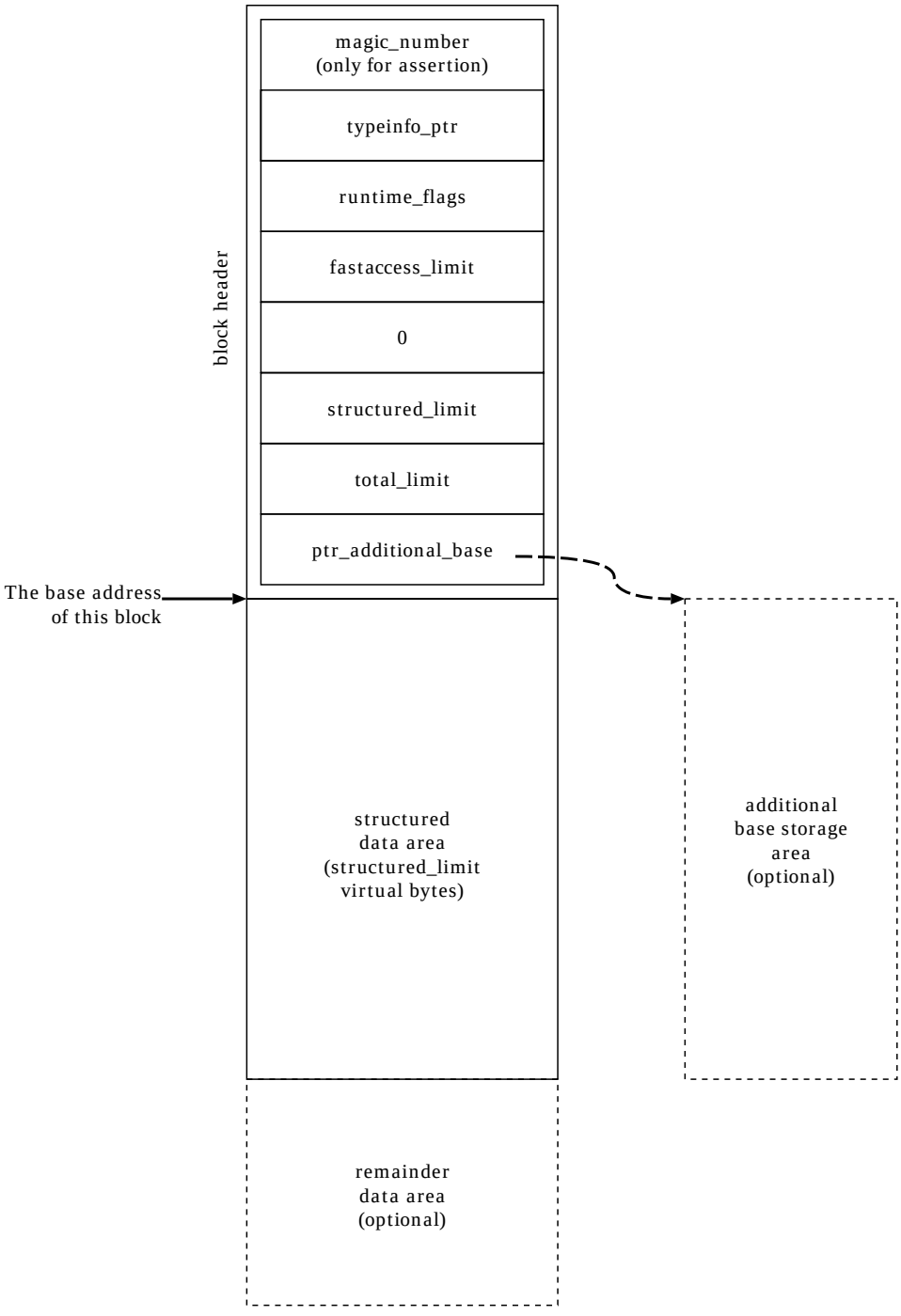


Figure A.1: The structure of memory blocks and block headers.

- No-deallocation: a block with this flag should never be deallocated in any way. Statically allocated blocks such as global variables, function stubs, type informations have this flag set.
- No-deallocation-by-user: a block with this flag should not be an operand of `free()` function. In addition to the block with no-deallocation flag, temporary blocks for varargs and local variables, and blocks allocated by system library (e.g. `FILE` object) have this flag set.
- Out-of-use: a block with this flag cannot be accessed any more, because it is deallocated. blocks which are already released by `free()`, or temporary blocks (e.g., blocks for local variables) whose lifetime is finished have this flag set.[1] Actual deallocation of these blocks are performed by the garbage collector.

fastaccess_limit A number of virtual bytes which can be directly accessed with well-typed pointers.

0 Single zero is stored in this field for fast check of cast flags described in Section 4.2.

structured_limit A number of virtual bytes in the structured data area.

total_limit A number of whole virtual bytes in this block, including both structured and remainder area.

ptr_additional_base An optional pointer to the additional base storage area (see Section 4.1.1). If an area is not allocated for this block, 0 (NULL) is stored.

The three limit values are used for different purpose to optimize runtime operations. *Total-limit* and *structured-limit* determine the size and the structure of the data areas. *Fastaccess-limit* has one of two possible values: that equal to the *structured-limit* in normal blocks, or zero for blocks which need special attentions (e.g., already unallocated blocks). Throughout the program execution, those values must meet the following constraints:

1. The *structured-limit* must be a multiple of the element size of the data type, and must not be greater than *total-limit*.
2. The *fastaccess-limit* must be either zero or equal to *structured-limit*.
3. If any pointer points to the memory block, both *structured-limit* and *total-limit*, along with the type information, should not change.

An exception to these rules is the type-undecided pseudo-type, described in Section 4.3.

[1] A difference between out-of-use block and finished varargs block is that the latter should be "deallocated" once more by a caller function.

The data areas inside memory blocks are split into three parts. The first area, called *structured data area*, contains most part of the block data. Its virtual size is always multiple of the element size of the block's data type. All statically-allocated data only have structured data area at the beginning of program execution. The remainder data area (Section 4.1.2) only appears in dynamically-allocated blocks and holds the extra data which does not fit in the format of associated block type. The final part is an *additional base storage area* (Section 4.1.1).

There are a few kinds of blocks which use special format for data area. They are described in separate sections (functions in Section 3.5.2, type information block in Section A.1.2, type-undecided blocks in Section 4.3, externally-defined abstract types in Section 4.4.3, magical blocks in section 4.4.4). Even for these blocks, the format of the block headers is common.

A.1.1.2 Value representation in structured data area

The format of values stored inside structured data area of memory blocks vary depending on its block type.

Fat pointers and fat integers The block format for fat pointers and word-sized fat integers are simple. Fat values described in Section 3.1 are arranged sequentially. Cast flags in fat pointers are maintained in coherent to the block type stored in the associated block header. Additional base storage area is not required nor used for blocks of these types.

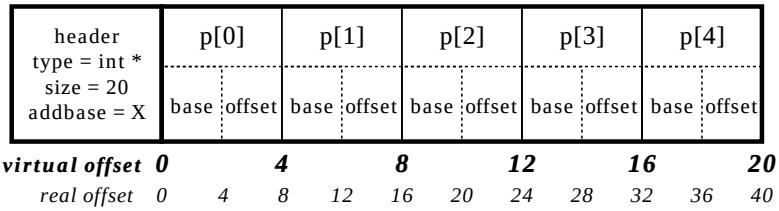
Narrow Integers and Floats Narrow integers (i.e., usually, `char` and `short`) and floating numbers will not hold a valid pointer value. Thus data representations for these types are the same as that of native implementation. If a program stores pointer data in data blocks of these types, additional base storage areas will be allocated and used. Figure A.2 shows the structure of the blocks of pointers, integers and floating values.

Structures For struct types, the packed-style relesentation of each element described in Section 3.4 is arranged in the main data area of data blocks. Additional base storage areas is used if the struct type contains any non-fat members. Figure A.3 shows an example of a block representation for struct values. If no members of the struct are fat data, the representation will be equivalent to the native representation.

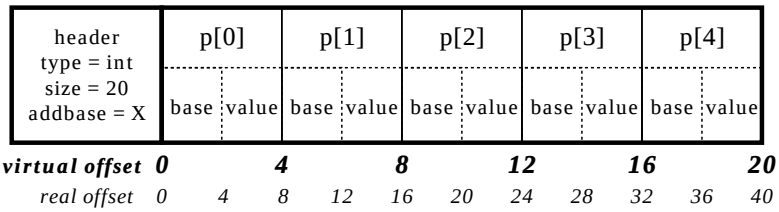
A.1.2 Type information and access methods

Type information blocks keep the information of various runtime types, and also serve as dynamic dispatch tables for access methods.

Pointers (int *):



Int:



Float:

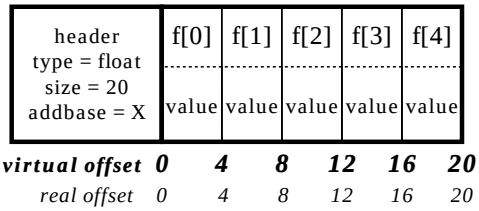
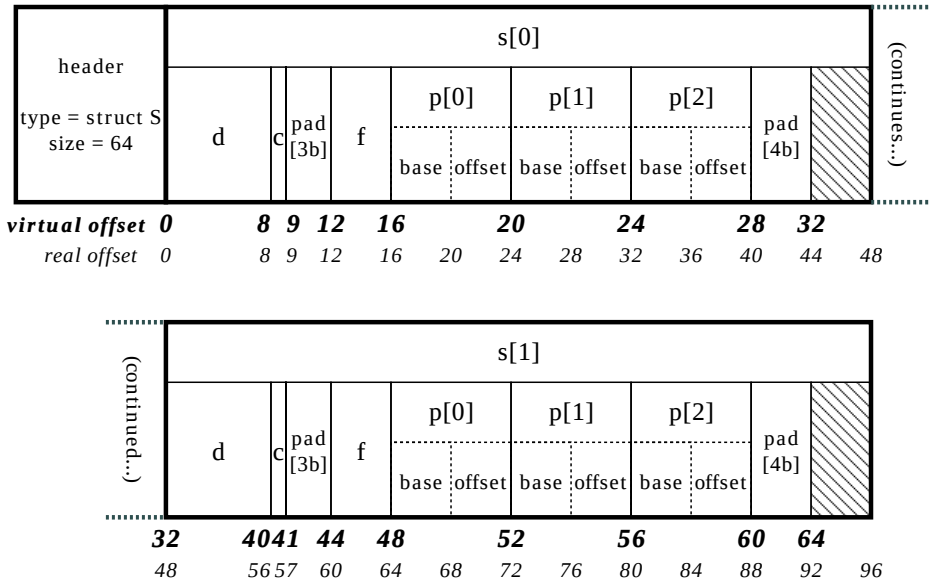


Figure A.2: Block structure for pointers and primitive types.

```
struct {
 double d;
 char c;
 float f;
 char *p[3];
} s[2];
```



- See Figure 3.4 for the use of paddings.

Figure A.3: Representation of struct data blocks

Figure A.4 shows the structure of type information blocks. Each type information block consists of three parts: a block header, an information section, and a method table.

Following information is stored in an information section:

Name of the type User-readable string representation of the type's name. Used in error handlers.

Element sizes Sizes of a single element of the corresponding type, counted both in virtual bytes and in real (representation) bytes. These values (specifically, the ratio of these values) are used for memory allocations and for accessing remainder areas (described in Section 4.1.2).

Flags This field holds the following information.

- **Kind of the type** One of primitives, pointers, functions, structures, or special/abstract types.
- **User allocation information** The flag indicates that instances of this type cannot be dynamically allocated by user programs. Typically functions, abstract types and other special types have this flag set.
- **Continuous flag** It indicates whether the representation of the type is continuous, i.e., it matches to the native representation. For example, narrow integers, floats, arrays of continuous data types, or structs composed of only continuous types are continuous. Data blocks of continuous types can be passed directly to external routine (e.g. system calls) once boundary check succeeds. Wrapper routines for native functions check this flag to avoid redundant data copying. The flag also changes the semantics of the remainder data area slightly (Section 4.1.2).

Referee's type information This field in type information blocks for pointers points to the type information block of the target type of the pointer type. For example, this field in the type information block for `int *` points to the type information block for `int`.

The method table in type information blocks contains pointers to the access methods. Currently the following fields are defined.

```
dvalue (*ti_read_dword)(base_t, ofs_t);
value (*ti_read_word)(base_t, ofs_t);
hword (*ti_read_hword)(base_t, ofs_t);
byte (*ti_read_byte)(base_t, ofs_t);

void (*ti_write_dword)(base_t, ofs_t, dvalue, typeinfo_t);
void (*ti_write_word)(base_t, ofs_t, value, typeinfo_t);
void (*ti_write_hword)(base_t, ofs_t, hword, typeinfo_t);
void (*ti_write_byte)(base_t, ofs_t, byte, typeinfo_t);
```

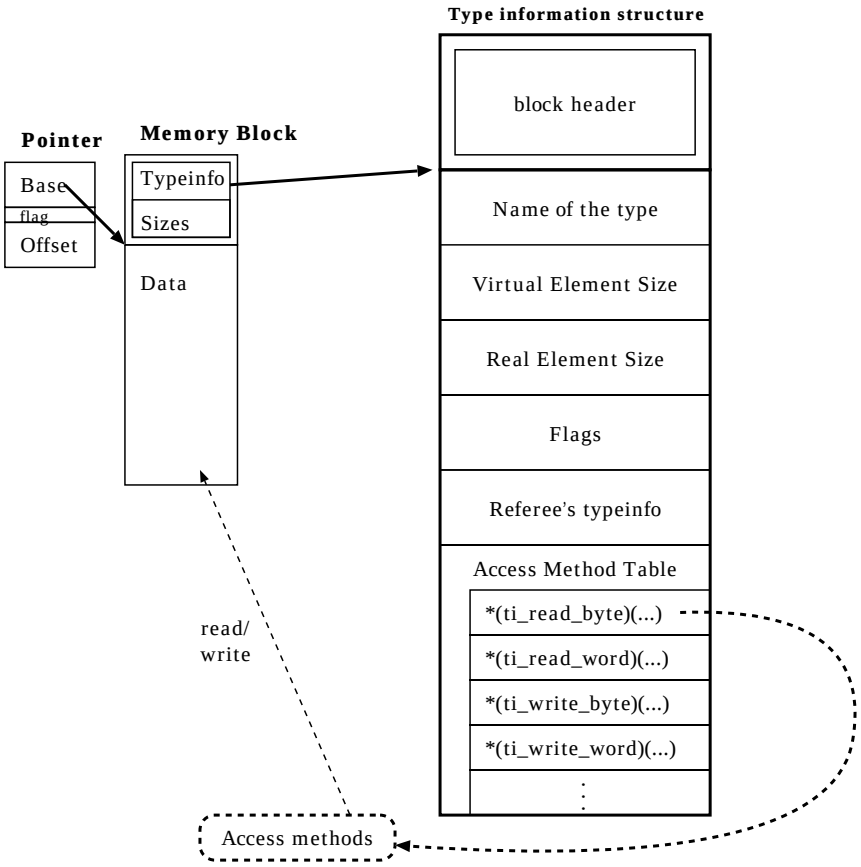


Figure A.4: Structure of type information blocks.

First four methods are access methods for read access to memory blocks in four different sizes. Each of them takes one unpacked fat pointer and returns the value stored in the corresponding virtual memory location in generic integer types. The other four methods are for write accesses. First two arguments of each method indicate a virtual memory location to access, the next one does a value to be stored. The final argument passes of the type of the memory access. For access with normal primitive types and pointer types (e.g., `*(char *)p = 'x'`), it will be the pointer to the type information block of the corresponding type. However, if the access is performed on a part of a struct (e.g., `p->f.g = 'x'`), the type of the mostly outer struct in the assignment expression (e.g., the type of `*p`, not of the field `g`) will be passed. Almost all access methods simply ignore this information, but the access methods associated with type-undecided blocks use this information to initialize a memory block to the correct type.

Finally, every type information block also has a valid block header, to make it possible to be referred to by pointers in user programs. It has a special runtime type like abstract data types 4.4.3 to prevent any modification to the information. The addresses of the type information blocks can be retrieved from user program by using a primitive operator `__typeof(`*x*`)`, where *x* can be either a type or an expression (like `sizeof` operator in C), which returns a pointer to the information block of the corresponding type as a `void *` pointer. The intended use of this operator is to implement a special runtime routines (e.g., a type-specified memory allocator, a runtime type checker for debugging purpose). Figure A.5 shows an example of relation between type information blocks.

The type information blocks and access methods for the primitive types (`char`, `short`, `int`, `long long`, `float`, and `double`), as well as those for some special types (`void`, type information), are defined in the runtime library. Type information for all compound types (i.e., pointers, functions, structs and unions) are generated by the compiler, because these have infinite possibility of variations. The generation of access methods is discussed in Section A.2.4.

A.1.3 Memory management

As already mentioned, invocation of `free()` library function by user program does not immediately release the memory block. Instead, it just marks the block as inactive and prevents further access to this block. To mark that the block is inactive, `free()` sets the *runtime-flag* field of the target block to the "released, out-of-use" state. In addition, it sets *fastaccess-limit* to 0 (Figure 4.7), redirecting all memory accesses to associated access methods. The access methods check the *runtime-flag*, find out that the accessed block is already "deallocated", and raise access error.

Current implementation of Fail-Safe C uses the conservative mark-sweep garbage collector implemented by Hans-J. Bohem et al. [10, 9] as the back-end memory manager. As memory blocks returned from Bohem's collector are only word aligned, the runtime system aligns block addresses to double-word boundary by itself.

```
int y[4] = {0,1,2,3}; int *x[4]={&y[0],0,0,0};
```

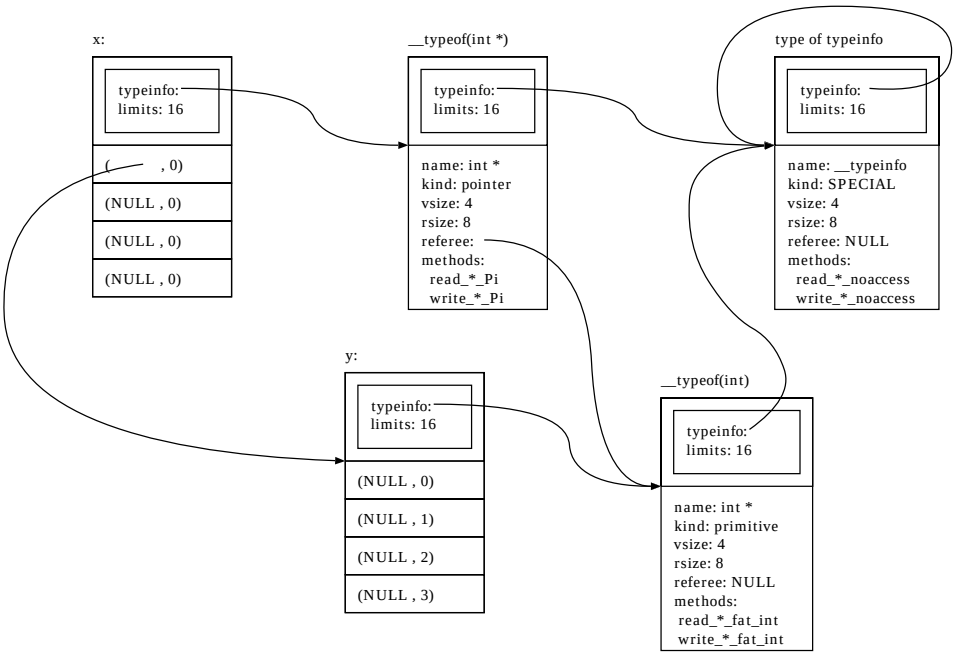


Figure A.5: An example configuration of relationship between typeinfo blocks

Although Bohem's garbage collector is well implemented and is reasonably fast, it is desirable to adopt exact (non-conservative) garbage collection when possible. Theoretically it is possible to adopt exact garbage collector to Fail-Safe C system, because all base addresses stored in the memory block can be reliably identified by its block type, and all of those in local variables can be identified by its static type. However, utilizing exact garbage collector is impossible while using usual C compiler as a back-end code generator, because no type information on native stacks can be obtained. There are several method for workaround:

1. Use partially-conservative garbage collectors. Bohem's gc allows programs to tell that some words in memory blocks do not contain any pointer values. Unfortunately, its interface is not well documented, and it cannot be used for Fail-Safe C because the block format expected by their gc is not compatible with the block format of Fail-Safe C. Further more, it still sweeps all other memory words conservatively.

   Some garbage collectors [6, 39][2] allow exact handling of pointers in heap by passing type information (more exactly, the locations of pointers inside blocks) to memory allocator, while using conservative approach for native stacks and other untyped areas. For example, Kaffe, a virtual machine for Java byte-codes, uses a kind of this approach (described in [54]).

2. Generate native assembly code directly, and make own records for tracing pointer values in native stack. Many advanced implementation of safe languages, such as Objective Caml [56] system, take this approach. It requires huge amount of implementation work and damages portability of systems.

   One possible, realistic variation of this approach is to use low-level intermediate language which has a support for stack inspection. C−− [37, 57] is one of such intermediate languages, which provides a similar level of abstraction as C language, performs various tiresome job for code generation such as register allocation and spilling, and provides a set of routines for inspection of stack structures and values which can be used for exact garbage collectors.

A.2 Generated code

This section describes internal of the code generated by current Fail-Safe C compiler.

[2] The referred articles are discussing about adopting conservative collection technique for copying garbage collection. As memory blocks which are indefinitely pointed by values which are conservatively guessed as pointer are impossible to move around memory locations, these systems use conservative, mark-sweep strategy for type-unknown area (such as stacks) and use exact, copying strategy for other values. Note that C copying collection is not useful even for type-known values on Fail-Safe C, because Fail-Safe C reveals the real address of objects to user programs as integers. Copying collection thus changes behavior of existing user programs which do not expect such movements.

Table A.1: Translated types for various builtin types.

original type	translated type: packed type	translated type: unpacked types: base address	translated type: unpacked types: value/offset
char	byte (*u_char*)	—	byte
short	hword (*u_short*)	—	hword
int, long	value (*u_long long*)	base_t (*u_int*)	word (*u_int*)
long long	dvalue (*a struct*)	base_t	dword (*u_long long*)
float	float	—	float
double	double	—	double
pointers	ptrvalue (*u_long long*)	base_t	ofs_t (*u_int*)

Each entry shows the name of translated types, with real typedef'ed type shown in parentheses. The type specifier `unsigned` is abbreviated to "*u_*". For local variables of integer types, the original type is used instead for value part of unpacked translated types.

A.2.1 Encoding for primitive types

Table A.1 shows the name of translated types corresponds to various builtin types in usual 32bit architecture.

Current implementation uses gcc's double-word integer type (`long long`) to hold fat integers and fat pointers in packed representations. Under this encoding, hereafter called "standard encoding", primitive operations on the standard encoding are implemented as follows.

- Composing a fat value: `((word)(`*v*`) | (dword)(word)`*b*` << 32)`
- Converting an integer to a fat integer: `(value)(word)`*x*
- Taking the base part: `(base_t)(`*v*` >> 32)`
- Taking the value/offset part: `(word)(`*v*` & 0xffffffffU)`

On Intel i386, inline assembler facility of gcc is also used. The composition operation is replaced with the following "empty" assembly directive:

```
static inline value value_of_base_vaddr(base_t b, word va)
{
    value p;
    __asm("": "=A" (p): "a" (va), "d" (b));
    return p;
}
```

This directive directs the compiler that variables *va* and *b* should be arranged to `eax` and `edx` registers respectively, and then assume that the double-word result is on register pair `edx:eax`.

Alternatively, another encoding which uses the `__complex` extension of gcc can also be possible. The type of fat values is declared as `unsigned int __complex`, and operations are implemented as follows.

- Composing a fat value: `(value)((word)(`*v*`) + (word)`*b*` * 1i)`
- converting an integer to a fat integer: `(value)(word)`*x*
- Taking the base part: `__imag` *v*
- Taking the value/offset part: `__real` *v*

The relative performance of these encodings varies among several programs, but in some preliminary experiments the standard encoding (with an inline assembly code) performs slightly better than others. The result of those tests are shown in Section A.4. Unfortunately, gcc (at least version 2.95.4 for Intel architecture and version 2.95.3 for SPARC architecture) has severe bugs in handling of complex values, which makes a program code included to every compilation units under Fail-Safe C cause an internal compiler error inside a register allocation routine. For this reason, current Fail-Safe C implementation avoids using alternative encoding.[3]

A.2.2 Encoding of typenames and other identifiers

Type inconsistency between library routines and user programs is severe problem to whole system under Fail-Safe C. Thus, it uses an ASCII-encoding of various data type, which are similar to those used in C++ language to support function overloading, in various places: the name of (specific main entry of) functions, type information blocks, access methods, various support inline functions, and others. The type-name encoding rules used in Fail-Safe C is shown in Table A.2.

There are two different encoding for structs: The structs defined in user programs are currently referred by its internal identification number (encoded as `S`*n*), which differentiate the encoding of the same struct in different programs. As a compile-time option, the current compiler also provides limited support for separate compilation by encoding the location of struct definitions into the type name. Unfortunately, this encoding may produce unsound compilation in very tricky programs, although it is much safer than simple name-based encoding when there are two different declarations of structs with the same name. True support for separate compilation is left as future work.

[3]On Intel architecture, the experiments on alternative encoding is performed by disabling inline expansion for some library functions which causes internal errors. On Sparc architecture, even a non-inline version of these functions failed, and thus experiments for the alternative encoding are completely abandoned.

T	$\langle T \rangle$ (encoded name of T)
Primitive types:	
`void`[†]	`v`
`char`	`c`
`short`	`s`
`int`	`i`
`long`[‡]	`l`
`long long`[‡]	`q`
`float`	`f`
`double`	`d`
Pointers:	
T' `*`	`P`$\langle T' \rangle$
Functions:	
T_r`(void)`	`F_`$\langle T_r \rangle$
T_r`(...)`	`FV_`$\langle T_r \rangle$
T_r`(`T_1`,---,`T_n`)`	`F`$\langle T_1 \rangle$`---`$\langle T_n \rangle$`_`$\langle T_r \rangle$
T_r`(`T_1`,---,`T_n`,...)`	`F`$\langle T_1 \rangle$`---`$\langle T_n \rangle$`V_`$\langle T_r \rangle$
Structures:	
`struct` S (user-defined)	`S`i
`struct` S (external)	`Sn`ℓK`_`

[†] `v` is used for the base type of pointers and the return type of functions. The `void` specification in function parameters is represented by null string.

[‡] `l` and `q` are only used when size of its type are different from other integer types.

- Attributes such as `signed`, `unsigned`, `volatile`, `const`, and `inline` are ignored for type encoding.
- i: decimal internal ID of the structure
- K: keyword associated with the external structure
- ℓ: the length of the name K

Table A.2: ASCII encoding of type names

On the contrary, the structs defined in system library headers will have specific, fixed names to allow separate compilation of libraries. For example, a `FILE` structure in the standard library are defined in `stdio.h` with special attribute as

```
struct __fsc_attribute__((named "stdio_FILE", external)) FILE;
```

and its type encoding becomes "`Sn10stdio_file_`". This ensures type-consistency between user program and the Fail-Safe C standard library.

Various other names in the program are also renamed systematically to avoid unintended crash of two names. Table A.3 summarizes such renaming.

A.2.3 Translating body of functions

The type-specific entry point of each functions has program code translated from the original definition. The entry point accepts unpacked values as arguments and returns packed translated value. For example, an function which has an original type `int(int, char *, double)` is translated to a function of translated type `value(base_t, int, base_t, ofs_t, double)`.

A.2.3.1 Variables and control flow

Fail-Safe C compiler firstly perform various preprocessing before translating memory operations in user program. Body of functions is expanded into a sequence of simple intermediate instructions. Especially, all local variables whose addresses are taken are expanded to pointer variables with a code performing explicit allocations and initializations (see Section 3.3.1).

Next, all fat variables (both pointers and integers) are separated into two variables. The purpose of this translation is to find out redundant and duplicate variables as much as possible. For example, almost all numeric operations does not refer to the base parts of operands, and generates null (0) base values. In addition, functions with heavy use of pointer arithmetics is likely to hold several pointer variables which points to the same array.

A.2.3.2 Arithmetics

Integer and floating arithmetic operations are translated into the operation on the value parts if operands are fat integers. The base part of the result is set to constant zero, which are often removed by redundant variable elimination in post-processing.

Pointer arithmetic operations are slightly more complicated. If an integer (i) is added to a pointer $[(b,o)^f]$, the virtual size of the target type of the pointer (vs) is multiplied to the integer operand, then it is added to the offset part of the pointer. If the virtual size of target type is a power of two, base part of the pointer does not need to be updated, because under modulo the size of the range of offsets (vms) which is a larger power of two (namely 2^{32} or 2^{64}),

$$((o + vs \cdot i) \bmod vms) \bmod vs = o \bmod vs$$

Renamed global identifiers:	
global variables	`GV_`*x*
function stub blocks	`GV_`*x*
static variables and functions	`GV_`*i*`_`*x*
string constants in expressions	`GSTR_`*i*
type-specific entry of functions	`FS_`⟨*T*⟩`_`*x*
type-generic entry of functions	`FG_`*x*
Renamed local identifiers:	
base part of function arguments	`FAB_`*i*`_`*x*
value/offset part of function arguments	`FAV_`*i*`_`*x*
(arguments for handling varargs	`FAva_B`, `FAva_V`)
local variables	`T_`*i*
Names for type-dependent values:	
type information block	`fsc_typeinfo_`⟨*T*⟩
type of translated structures	`struct struct_`⟨*T*⟩
type of memory block for single value	`struct fsc_storage_`⟨*T*⟩`_s`
memory block type for array of values	`struct fsc_storage_`⟨*T*⟩`_`*n*
Names for synthesized type-dependent internal functions:	
calculate real offset from virtual offset	`get_real_offset_`⟨*T*⟩
update cast flag	`set_base_cast_flag_`⟨*T*⟩
coerce integer to pointer	`ptrvalue_of_value_`⟨*T*⟩
access methods for user-defined structures	`read_`*size*`_`⟨*T*⟩
	`write_`*size*`_`⟨*T*⟩

- Legends for symbols: ⟨*T*⟩ is the encoded string for type *T*, *x* is the user-supplied identifier, *n* is the number of elements, *i* is an internally-generated unique identification number, and *size* is a keyword describing size of access.
- See respective subsections under this section for the meaning of entries.

Table A.3: Name encodings in Fail-Safe C

Table A.4: Symbols used in translation rules

$x, y, p, q, \ldots$	packed local variables
$x_b, p_b, \ldots$	base field of variables
$p_o, q_o, \ldots$	offset field of fat pointer variables
$x_v, y_v, \ldots$	value field of fat integer variables
$T_x, T_p, \ldots$	static type of variables
slanted-name	field name, internal operator, etc.
slanted-name$_T$	type-dependent operation
sans_serif_name	functions in runtime library or generated functions
$\langle T \rangle$	encoded string of type name T
L1:, **L2:**, ...	targets of branch instructions
$[\![E]\!]$	E translated by another translation rule

- $[\![\cdot]\!]$ may appear in variable positions of other statements. Internally, temporary variables are allocated for these values. For example, $\mathsf{f}([\![(T)x]\!])$ means $[\![t = (T)x]\!]; \mathsf{f}(t)$ where t is a fresh temporary variable.

is always satisfied (because $vms \bmod vs = 0$), that means the result pointer is aligned if a pointer operand is aligned. However, if the virtual size is not a power of two, the cast flag must be updated when integer overflow is occurred during offset calculation. Figure A.6 summarizes the translation rule for arithmetic operations.

A.2.3.3 Cast operations

Cast operation between integer types do not trash the base part of the operand value if the result is also a fat type. If the operand does not have base part, the base part of the result, if any, will be set to 0. Cast operation between pointer types recalculates the cast flag of the target pointer, not changing other parts.

Because pointers and fat integers uses different representations, cast between these types converts virtual offsets to virtual addresses by adding the base part of the operand (removing cast flag), or vice versa. The cast flags are removed on integers and recalculated for pointers, as usual.

Figure A.7 summarizes the translation rules for cast operations.

A.2.3.4 Taking address of variables

Taking the address of a simple global variable is almost straightforward. The address of the main part of the block (`val` field, see Section A.2.6) is copied into the base part of the result. However, taking the address of a field of a global variable must be done slightly carefully. Because the type of the field is different from the type of the enclosing variable, cast flag of the result pointer must be set to 1 (Figure A.8).

Table A.5: Internal operators used in translation rules.

sizeof(a)	The virtual size of the expression, type, or field *a* in bytes. [*constant integer*]
real-sizeof(a)	The real size of the expression, type, or field *a* in bytes. [*constant integer*]
remove-cast-flag(b)	Returns copy of *b*, which is base part of unpacked pointer, with cast flag changed to 0. [*inline function in runtime library*]
set-cast-flag(b)	Returns copy of *b* with cast flag changed to 1. [*inline function in runtime library*]
cast-flag(b)	Returns cast flag of *b* in boolean. [*inline function in runtime library*]
update-cast-flag$_T(b,o)$	Returns the copy of *b* with cast flag changed so that (b,o) will be a valid pointer as type T. Assuming type T' to be the referee type of pointer type T, the cast flag of the result will be set when (1) *b* is null, (2) *b* points to memory blocks with type different from T', or (3a) the offset *o* is not multiple of the virtual size of element in concrete type T' or (3b) the offset *o* is not 0 and T' is abstract, and in other cases it will be cleared. [*inline function, either in standard library or generated by the compiler*]
isnull(b)	Returns 1 if the base *b* is null (cast flag may be either 0 or 1). [*inline function in runtime library*]
offsetof(f)	Returns the virtual offset of field f counting from the top of enclosing struct. [*constant integer*]

Numeric arithmetics:

$$z = x \odot y \quad \text{(binary)} \implies \left[\begin{array}{l} z_v = x_v \odot y_v \\ z_b = 0 \end{array} \right]$$

$$z = \$x \quad \text{(unary)} \implies \left[\begin{array}{l} z_v = \$x_v \\ z_b = 0 \end{array} \right]$$

- The code $z_b = 0$ is omitted for narrow integers and floats.

Pointer addition:

- if $sizeof(T_p)$ is a power of 2:

$$q = p \pm x \implies \left[\begin{array}{l} q_o = p_o \pm x * sizeof(T_p) \\ q_b = p_b \end{array} \right]$$

- if $sizeof(T_p)$ is not a power of 2:

$$q = p \pm x \implies \left[\begin{array}{l} q_o = p_o \pm x * sizeof(T_p) \\ \text{if overflow/underflow:} \\ \qquad q_b = update\text{-}cast\text{-}flag_{T_q}(p_b, q_o) \\ \text{else:} \\ \qquad q_b = p_b \end{array} \right]$$

Pointer-pointer subtraction:

$$x = p - q \implies \left[\begin{array}{l} \text{if } q_b = p_b \text{ (modulo cast-flag):} \\ \qquad x_v = (p_o - q_o) / sizeof(T_p) \\ \qquad x_b = 0 \\ \text{else: error} \end{array} \right]$$

Figure A.6: Translation rules for arithmetic operations

Cast between fat integers:

$$y = (T_y)x \implies \left[\begin{array}{l} y_v = (T_y)x_v \\ y_b = x_b \end{array} \right]$$

Cast from narrow integers to fat integers:

$$y = (T_y)x \implies \left[\begin{array}{l} y_v = (T_y)x_v \\ y_b = 0 \end{array} \right]$$

Cast from fat integers to narrow integers:

$$y = (T_y)x \implies y_v = (T_y)x_v$$

Cast between pointers:

$$q = (T_q)p \implies \left[\begin{array}{l} q_o = p_o \\ q_b = \textit{update-cast-flag}_{T_q}(p_b, q_o) \end{array} \right]$$

Cast from pointers to integers:

$$x = (\texttt{int})p \implies \left[\begin{array}{l} x_b = \textit{remove-cast-flag}(p_b) \\ x_v = x_b + p_o \end{array} \right]$$

Cast from integers to pointers:

$$p = (T_q)x \implies \left[\begin{array}{l} p_o = x_v - x_b \\ p_b = \textit{update-cast-flag}_{T_p}(x_b, p_o) \end{array} \right]$$

Figure A.7: Translation rules for casts

Taking address of global variables:

$$p = \&v \implies \left[\begin{array}{l} p_o = 0 \\ p_b = \texttt{(base\_t)\&GV\_}v\texttt{.val} \end{array} \right]$$

Taking address of a field of global variables:

$$p = \&v.f \implies \left[\begin{array}{l} p_o = \textit{offset-of}(f) \\ p_b = \textit{set-cast-flag}(\texttt{(base\_t)\&GV\_}v\texttt{.val}) \end{array} \right]$$

Taking address of a field of a target of pointers:

$$q = \&(p\texttt{->}f) \implies \left[\begin{array}{l} q_o = p_o + \textit{offset-of}(f) \\ q_b = \textit{update-cast-flag}_{T_q}(p_b, q_o) \end{array} \right]$$

Figure A.8: Translation rule for pointer address operation

Taking the address of a field of a object via pointer is essentially a variation of pointer arithmetic. Cast flag is recalculated to maintain runtime type safety.[4]

A.2.3.5 Memory accesses

Memory access operations are most important operations to perform safety check in Fail-Safe C system. Figure A.9 shows the translation rules for pointer dereferences (read accesses). First the code checks the boundary, cast, and null condition of the dereferenced pointer. As already discussed in Section 4.2, Fail-Safe C uses an implementation trick to perform those three checks in single comparison. If boundary test succeeds, the real address of the referenced element in target memory block is calculated, and data are read. The ratio of the real offset to the virtual offset is hard-coded in output code. For simple types and pointer types it will be an integer. If the check is failed, there are many possible cases: boundary overrun, type mismatch, null pointer dereferencing, or dereferencing a pointer to the remainder area or type-undecided region. Except for the null pointers, the system picks up a read access methods from the header of the referred block and call it to delegate detailed safety check and real memory access. The returned value is either a fat integer or narrow integer depending the type, thus it should be converted to the expected type by the caller.

Field access via pointer (-> operator in C language) is a variation of the simple pointer dereference. If the pointer is not cast, the pointer is correctly aligned and pointing to the top of an element of the enclosing struct, thus the access can simply be translated to a field dereferencing in output code. Otherwise, the access

[4]There is a chance the resulting pointer may be well-typed, when the operand was ill-typed (the cast flag is 1).

Reading memory via pointers:

$$x = *p \implies \left[\begin{array}{l} \binom{\text{if } \textit{is-null}(p_b)\text{: error}}{\text{if } \textit{cast-flag}(p_b) = 1\text{: goto L1}}^{\dagger} \\ \text{if } p_b\text{->}\textit{header.fastcheck-limit} < p_o\text{:} \\ \qquad x = *\left(p_b + p_o * \left(\frac{\textit{real-sizeof}(T_p)}{\textit{sizeof}(T_p)}\right)\right) \\ \text{else:} \\ \qquad \textbf{L1:} \\ \qquad t = p_b\text{->}\textit{header.typeinfo->read-access-method}(p_b, p_o) \\ \qquad [\![x = (T_x)t]\!] \end{array}\right]$$

Reading field of struct via pointers:

$$x = p\text{->}f \implies \left[\begin{array}{l} \binom{\text{if } \textit{is-null}(p_b)\text{: error}}{\text{if } \textit{cast-flag}(p_b) = 1\text{: goto L1}}^{\dagger} \\ \text{if } p_b\text{->}\textit{header.fastcheck-limit} < p_o\text{:} \\ \qquad x = \left(p_b + p_o * \left(\frac{\textit{real-sizeof}(T_p)}{\textit{sizeof}(T_p)}\right)\right)\text{->}f[\text{.cv}] \\ \text{else:} \\ \qquad \textbf{L1:} \\ \qquad t = p_b\text{->}\textit{header.typeinfo->read-access-method} \\ \qquad\qquad\qquad\qquad (p_b, p_o + \textit{offset-of}(f)) \\ \qquad [\![x = (T_x)t]\!] \end{array}\right]$$

- †: these checks are merged into next *if* instruction in the actual implementation (see Section 4.2).
- Appropriate *read-access-method* will be chosen based on the size of x.
- The field ".cv" is only used when field f contains a fat integer or a fat pointer (see Section A.2.6).

Figure A.9: Translation rule for pointer dereference

is translated as if it were a combination of pointer cast, an addition of the element offset, and a dereference operation.

Write access is almost a dual operation to read access, except that access methods require one additional argument, which is the type information about the context of the access. For simple write access, the information is just the static type of the element to be written. For field access, however, the type of the enclosing structure, not the type of accessed element, is passed to the access method (Section A.1.2).

A.2.3.6 Invoking functions directly

Invoking function with fixed number of arguments via direct identifier is translated straightforwardly as shown in Figure A.11. Type-specific entry points of translated functions require unpacked representation for arguments. Contrarily, return values are packed values so that it will be unpacked when needed (not shown explicitly in the figure).

If a function receives varargs, an array of word-size fat integers is allocated by invoking a library function, and all arguments for the varargs slot are put sequentially into the array. Then, a fat pointer to the array, is passed to the function as additional arguments with special names. If there are no real arguments for varargs, a null pointer is passed instead. The offset part of the additional pointer is always zero when the function called under these rules, but it may be different when the function is invoked via generic stub entry point (described in Section A.2.5).

A.2.3.7 Invoking functions via pointers

When the program invokes a function using a function pointer, the pointer in the translated program will point to the stub block of the function (Section A.2.5). At the invocation, the translated code (Figure A.12) first checks for the cast flag of the pointer. If the pointer is not cast, the pointer to the type-specific entry point is taken from the stub block and invoked in the same way as in usual function invocation (see the previous section). The offset part of the function pointer is always zero when function pointer is not cast, thus no checks are needed.

If the pointer is cast, however, it may point to any kind of blocks, which may be not even a function stub, and offset may also be arbitrary. First, the kind of the referred block and the offset part of the pointer is checked. If it is a correct pointer to a function (of a different type), all arguments, including fixed arguments, are passed to the generic entry point of the function in the same way as varargs arguments. The value returned from the generic entry is a fat integer type and will be converted to the expected type by the caller.

A.2.3.8 Receiving varargs arguments

The additional fat pointer for variable-number arguments are received by the callee by specially named formal parameters `FAva_b` and `FAva_v`. Because these names

Writing into memory via pointers:

$$*p = x \implies \left[\begin{array}{l} \binom{\text{if } \textit{is-null}(p_b)\text{: error}}{\text{if } \textit{cast-flag}(p_b) = 1\text{: goto L1}}^{\dagger} \\ \text{if } p_b\text{->}\textit{header.fastcheck-limit} < p_o\text{:} \\ \qquad *\Big(p_b + p_o * \Big(\frac{\textit{real-sizeof}(T_p)}{\textit{sizeof}(T_p)}\Big)\Big) = x \\ \text{else:} \\ \qquad \textbf{L1:} \\ \qquad p_b\text{->}\textit{header.typeinfo}\text{->}\textit{write-access-method} \\ \qquad\qquad (p_b, p_o, [\![(\texttt{int})x]\!], \mathsf{fsc\_typeinfo\_}\langle T_x\rangle.\mathsf{val}) \end{array}\right]$$

Writing into field via pointers:

$$p\text{->}f = x$$

$$\implies \left[\begin{array}{l} \binom{\text{if } \textit{is-null}(p_b)\text{: error}}{\text{if } \textit{cast-flag}(p_b) = 1\text{: goto L1}}^{\dagger} \\ \text{if } p_b\text{->}\textit{header.fastcheck-limit} < p_o\text{:} \\ \qquad \Big(p_b + p_o * \Big(\frac{\textit{real-sizeof}(T_p)}{\textit{sizeof}(T_p)}\Big)\Big)\text{->}f[.\mathsf{cv}] = x \\ \text{else:} \\ \qquad \textbf{L1:} \\ \qquad p_b\text{->}\textit{header.typeinfo}\text{->}\textit{write-access-method} \\ \qquad\qquad (p_b, p_o + \textit{offset-of}(f), [\![(\texttt{int})x]\!], \mathsf{fsc\_typeinfo\_}\langle T_{(*p)}\rangle.\mathsf{val}) \end{array}\right]$$

- †: these checks are merged into next *if* instruction in actual implementation (see Section 4.2).
- Appropriate *write-access-method* will be chosen based on the size of x, and the type `int` will actually be an integer of that size.
- The field ".cv" is only used when field f contains a fat integer or a pointer (see Section A.2.6).

Figure A.10: Translation rules for pointer write

Invoking simple function:

$$x = f(a_0, a_1, \ldots, a_n) \implies x = \mathsf{FS\_}\langle T_f \rangle\mathsf{\_}f(a_{0.b}, a_{0.v}, a_{1.b}, a_{1.v}, \ldots, a_{n.b}, a_{n.v})$$

- Base addresses for narrow integers, floating numbers and struct arguments are skipped. Offsets are used instead of values for pointer arguments.

Invoking function with variable number of parameters:

$$x = f(\overbrace{a_0, a_1, \ldots, a_n}^{\text{fixed}}, \overbrace{b_0, b_1, \ldots, b_n}^{\text{varargs}}) \implies \left[\begin{array}{l} \text{(prepare fixed arguments)} \\ t = \mathsf{fsc\_alloc\_varargs}(n) \\ \mathsf{fsc\_put\_varargs}(t, 0, [\![(\texttt{int})b_0]\!]) \\ \qquad \vdots \\ \mathsf{fsc\_put\_varargs}(t, n, [\![(\texttt{int})b_n]\!]) \\ x = \mathsf{FS\_}\langle T_f \rangle\mathsf{\_}f(\ldots, t, 0) \\ \mathsf{fsc\_dealloc\_varargs}(t) \end{array}\right]$$

- If some arguments are double-word size, fsc_put_varargs_2 will be called with double-word fat integer argument, and all offset parameter passed for fsc_put_varargs and fsc_alloc_varargs will be adjusted to skip positions occupied by double-word arguments.

Figure A.11: Translation rules for direct function invocation

$$x = (*p)(a_0, a_1, \ldots, a_n)$$

$$\Longrightarrow \left[\begin{array}{l}
\text{if } \textit{is-cast}(p_b): \\
\quad \text{if } p_b\texttt{->}\textit{header.kind} \neq \texttt{FUNCTION}\text{: error} \\
\quad \text{if } p_o \neq 0\text{: error} \\
\quad t = \mathsf{fsc_alloc_varargs}(n) \\
\quad \mathsf{fsc_put_varargs}(t, 0, [\![(\texttt{int})a_0]\!]) \\
\qquad \vdots \\
\quad \mathsf{fsc_put_varargs}(t, n, [\![(\texttt{int})a_n]\!]) \\
\quad y = p_b\texttt{->}\textit{gen-entry}(t) \\
\quad \mathsf{fsc_dealloc_varargs}(t) \\
\quad [\![x = (T_x)y]\!] \\
\text{else:} \\
\quad p_b\texttt{->}\textit{spec-entry}(a_{0.b}, a_{0.v}, a_{1.b}, a_{1.v}, \ldots, a_{n.b}, a_{n.v})
\end{array}\right]$$

- See notices in Figure A.11. If the type of function pointer have varargs, it will be passed to specific entry in the way shown in Figure A.11, and passed to generic entry by putting them into t after usual arguments.

Figure A.12: Translation rule for function invocation via pointers

do not overlap with translated names of other parameters, there is no direct way to access those parameters from user programs. Instead, a special library function `__builtin_va_start` is declared in the runtime library. This function is actually a macro composing a fat pointer from these special formal parameters.[5] The standard library macro `va_start()` uses this special function to get a fat pointer to variable arguments, and all other operations on varargs are implemented solely in user-level macros.

Because the values of type `va_list` type can be passed to other functions like `vsprintf`, the block containing values for variable arguments must be a valid fat pointer (i.e., it must have a valid block header), and care must be taken for misbehaving user programs which store values of type `va_list` in a long-live heap area. Thus these blocks are heap-allocated and not released after returning from functions. The function fsc_dealloc_varargs only checks runtime flags and then disables the block by setting *fastaccess-limit* to 0. The actual deallocation is delegated to the garbage collector.

[5] Using `va_start()` in functions without varargs causes a compilation error.

A.2.4 Generating type-related data and methods

A.2.4.1 Pointer types

Access methods for pointer types are not generated by compiler: a single set of access methods for pointer types in the runtime library is shared among all pointer types, because the data representation of these types are almost identical. The methods use the *referee* field in the type information block to check the type safety of the written pointers and put a cast flag appropriately.

For each pointer type appeared in the user program, two inline helper routines for cast operations are generated. First one, named `set_base_castflag_`$\langle T \rangle$, converts an unpacked pointer of any type to the target type by setting the cast flag of the argument. It sets the cast flag when (1) the type of the block referred to by the pointer does not match with target type, or (2) the offset of the pointer is (a) not a multiple of the element size (for concrete types) or (b) not zero (for abstract types). It also resets the cast flag if all of above conditions are not met. The second helper routine, named `ptrvalue_of_value_`$\langle T \rangle$, converts a packed fat integer to the target type.

A type information block is also generated for each pointer type. The values of fields are almost common to all pointer types: Access methods for word-sized access are already described, and other methods delegates the operation to the word-sized access methods. Figure A.13 shows an example of generated code for `char **` type.

A.2.4.2 Struct types

As the data layout inside structures might not be uniform, access methods for structures are more complicated than those for primitive types and pointer types. Thus Fail-Safe C compiler generates the code of the access methods for each structure.

To generate access methods for each structure type whose size is multiple of word size, Fail-Safe C compiler internally generates a table called *element access table*. For each virtual offset inside one element of the structure, the compiler calculates the element which contains the target byte as a part of it, and the real offset of the byte which corresponds to the virtual offset (if any). The real offsets inside elements which do not use native-compatible representation (i.e. fat pointers and fat integers) are undefined. The left three column in Figure A.14 show the table obtained from the following structure.

```
struct S {
  double d;
  char c;
  float f;
  char *p[3];
};
```

```
inline static base_t set_base_castflag_PPc(base_t b, ofs_t o)
{
  base_t b0 = base_remove_castflag(b);
  if (b0 &&                                /* null check */
      &fsc_typeinfo_Pc.val == get_header_fast(b0)->tinfo &&
                                           /* type check */
      o % 4 == 0)                          /* alignment check */
    return b0;
  else return base_put_castflag(b0);
}

inline static ptrvalue ptrvalue_of_value_PPc(value v)
{
  base_t b = base_of_value(v);
  ofs_t o = ofs_of_value(v);
  return ptrvalue_of_base_ofs(set_base_castflag_PPc(b, o), o);
}

struct typeinfo_init __attribute__ ((weak)) fsc_typeinfo_PPc =
  {EMIT_HEADER_FOR_TYPEINFO, /* macro emitting block header */
    {"**char",               /* human-readable type name */
     TI_POINTER,             /* kind, flags */
     &fsc_typeinfo_Pc.val,   /* referee */
     4, 8,                   /* virtual, real size of element */
     read_dword_by_word,     /* read access methods */
     read_word_fat_pointer,
     read_hword_by_word,
     read_byte_by_word,
     write_dword_to_word,    /* write access methods */
     write_word_fat_pointer,
     write_hword_to_word,
     write_byte_to_word
    }
  };
```

- For all code examples in this dissertation, comments are inserted and indentations are revised by hand.

Figure A.13: A set of auto-generated code for `char **` type.

virtual offset	real offset	element	access type			
			byte	half word	word	dbl. word
0	0	d	d + 0	d + 0	d + 0	d
1	1		d + 1			
2	2		d + 2	d + 2		
3	3		d + 3			
4	4		d + 4	d + 4	d + 4	
5	5		d + 5			
6	6		d + 6	d + 6		
7	7		d + 7			
8	8	c	c	c + 0	c + 0	c + 0
9	9	_pad0[0]	_pad0[0]			
10	10	_pad0[1]	_pad0[1]	_pad0[1] + 0		
11	11	_pad0[2]	_pad0[2]			
12	12	f	f + 0	f + 0	f	
13	13		f + 1			
14	14		f + 2	f + 2		
15	15		f + 3			
16	(*16*)	*p[0]*	*	*	p[0]	*
17			*			
18			*	*		
19			*			
20	(*24*)	*p[1]*	*	*	p[1]	
21			*			
22			*	*		
23			*			
24	(*32*)	*p[2]*	*	*	p[2]	*
25			*			
26			*	*		
27			*			
28	40	_pad1[0]	_pad1[0]	_pad1[0] + 0	_pad1[0] + 0	
29	41	_pad1[1]	_pad1[1]			
30	42	_pad1[2]	_pad1[2]	_pad1[2] + 0		
31	43	_pad1[3]	_pad1[3]			

Legends for Elements:

- Roman: A field which uses native representation
- *Italic*: A field which uses non-native representation

Legends for Access Type Rows:

- Field name: read the value of field with appropriate type conversion
- name + offset: read the memory directly inside a field of native representation by pointer manipulation
- *: decompose/delegate access to word-sized access

Figure A.14: Element access table for structure shown in Figure 3.4

After that, the compiler traverses the table to find out a correct way to access the data inside the structure for each access width (byte, half word, word, double word). The methods for data accesses are chosen from one of the following:

1. If whole part of the accessed area matches to one element inside the structure, the corresponding element will be accessed.

2. Otherwise, if every bytes of accessed area corresponds to a part of element which uses native representation, and the real offsets for these bytes are continuous, the access is directly performed on the corresponding memory region by using pointer casts and offset manipulations.

3. Otherwise, if it is a word-sized access and the target word is part of a non-natively represented double-word datum, the access is delegated to double-word access method.

4. Otherwise, the access is delegated to word-sized access.

Due to the fact that data types with non-native representation are always at least word aligned, word-sized accesses are guaranteed to be handled in first three methods, thus no infinite delegation will occur. Selected access patterns are compiled into one big `select` statement, and program codes for handling array of structs and remainder area are added. Figure A.15 shows a read access method of half-word access generated for the above structure type. In the example code, the internally-defined routine `read_hword_remainder` handles buffer-overflow error handling as well as remainder areas.

Access methods for word or double-word access support handling for additional base storage area described in Section A.1.1.2 when the target offset points to a natively-represented field. These support are implemented by combination of generated code and internally-provided support routines, as shown in Figure A.16.

All structures which is not multiple of word-size always use native representation, because all types using non-native representation require word alignments in virtual addressing. These structures are handled by the common access methods prepared for continuous data types.

A.2.5 Generic entry points and stub blocks for functions

As mentioned in Section A.2.3.7, generic stub entry points of functions receive a base address of an array which contains all arguments passed as fat integers. The stub function retrieves required arguments from the array and then passes it to the main entry of the functions. Values returned from the main entry are converted to the largest fat integer type and returned to a caller of the stub entry. Shortage of arguments raises runtime error, while redundant arguments are silently ignored. If the function receives varargs, the offset of the next slot of the last argument is passed to the additional argument for varargs mentioned in Section A.2.3.8. If the

```
/* struct struct_S1
   {
    double d;
    unsigned char c;
    unsigned char __pad1[3];
    float f;
    union fsc_initUptr p[3];
    unsigned char __pad2[4];};
   }; */

hword read_hwordS1(base_t b0, ofs_t ofs)
{
  base_t base = base_remove_castflag(b0);
  fsc_header * hdr = get_header_fast(base);
  if (ofs + 2 > hdr->structured_ofslimit)
     return read_hword_remainder (base, ofs);
  else {
    size_t ofs_outer = ofs / 32;
    size_t ofs_inner = ofs % 32;
    struct struct_S1 *bp = (struct struct_S1 *)base + ofs_outer;
    if (ofs_inner % 2) return read_hword_offseted_hword(base, ofs);
    else switch (ofs_inner) {
      case 0: return *((hword *)&(*bp).d);
      case 2: return *((hword *)((char *)&(*bp).d + 2));
      case 4: return *((hword *)((char *)&(*bp).d + 4));
      case 6: return *((hword *)((char *)&(*bp).d + 6));
      case 8: return *((hword *)&(*bp).c);
      case 10: return *((hword *)&(*bp).__pad1[1]);
      case 12: return *((hword *)&(*bp).f);
      case 14: return *((hword *)((char *)&(*bp).f + 2));
      case 16: return read_hword_by_word(base, ofs);
      case 18: return read_hword_by_word(base, ofs);
      case 20: return read_hword_by_word(base, ofs);
      case 22: return read_hword_by_word(base, ofs);
      case 24: return read_hword_by_word(base, ofs);
      case 26: return read_hword_by_word(base, ofs);
      case 28: return *((hword *)&(*bp).__pad2[0]);
      case 30: return *((hword *)&(*bp).__pad2[2]);
    }
  }
}
```

Figure A.15: A generated access method for half-word read access to struct type

```
value read_wordS1(base_t b0, ofs_t ofs)
{
  base_t base = base_remove_castflag(b0);
  fsc_header * hdr = get_header_fast(base);
  if (ofs + 4 > hdr->structured_ofslimit)
    return read_word_remainder(base, ofs);
  else {
    size_t ofs_outer = ofs / 32;
    size_t ofs_inner = ofs % 32;
    struct  struct_S1 *bp = (struct struct_S1 *)base + ofs_outer;
    if (ofs_inner % 4) return read_word_offseted_word(base, ofs);
    else {
      word result_v = 0;
      switch (ofs_inner) {
        case 0: result_v = *((word *)&(*bp).d); break;
        case 4: result_v = *((word *)((char *)&(*bp).d + 4)); break;
        case 8: result_v = *((word *)&(*bp).c); break;
        case 12: result_v = *((word *)&(*bp).f); break;
        case 16: return value_of_ptrvalue((*bp).p[0].cv);
        case 20: return value_of_ptrvalue((*bp).p[1].cv);
        case 24: return value_of_ptrvalue((*bp).p[2].cv);
        case 28: result_v = *((word *)&(*bp).__pad2[0]);
        break;
      }
      return read_merge_additional_base_word(result_v, b0, ofs);
    }
  }
}
```

Words at virtual offsets 0, 4, 8, 12, 28 have native representations. The case blocks for those offsets use `break` statement to pass the value read to internal subroutine `read_merge_additional_base_word` which cares about additional base area of the block. Other case blocks directly returns value to the caller by `return` statement.

The meaning of "`.cv`" field is described in Section A.2.6.

Figure A.16: A generated access method for word read access to a struct type

(Assuming the function f is type $T = T_r(T_{a_0}, T_{a_1}, \ldots, T_{a_n})$)

$$
\begin{array}{l}
\texttt{dvalue}\ \mathsf{FG\_}f(\texttt{base\_t}\ b)\{ \\
\quad i_0 = \mathsf{read\_word}(b,0) \\
\quad\quad a_0 = [\![(T_{a_0})i_0]\!] \\
\quad i_1 = \mathsf{read\_word}(b,4) \\
\quad\quad a_i = [\![(T_{a_1})i_1]\!] \\
\quad\quad\quad \vdots \\
\quad i_n = \mathsf{read\_word}(b,4n) \\
\quad\quad a_n = [\![(T_{a_n})i_n]\!] \\
\quad r = \mathsf{FS\_}\langle T\rangle\mathsf{\_}f(a_{0.b}, a_{0.v}, a_{1.b}, a_{1.v}, \ldots, a_{n.b}, a_{n.v}) \\
\quad \big(\mathsf{fsc\_finish\_varargs}(t,0)\big) \\
\quad \text{return } [\![(\texttt{long long})r]\!] \\
\}
\end{array}
$$

- See notices in Figure A.11 for handling of narrow arguments and double-word arguments.
- If the specific entry does not return any value, 0 is returned to caller.
- See the main text for the handling of varargs.
- fsc_finish_varargs is only called when f does not have varargs (otherwise it is already called inside f)

Figure A.17: Generation rule for stub entry point of functions

```
dvalue FG_main(base_t FAva_b)
{
  auto value T4;
  auto ofs_t T7;
  auto value T9;
  auto value T11;
  T4 = read_word(FAva_b, 0);
  T9 = read_word(FAva_b, 4);
  T7 = ofs_of_value(T9);
  T11 = FS_FiPPc_i_main
     (base_of_value(T4), (unsigned int)vaddr_of_value(T4),
      set_base_castflag_PPc(base_of_value(T9), T7), T7);
  return dvalue_of_value(T11);
}

struct fsc_function_stub_init GV_main = {
  EMIT_FSC_HEADER(fsc_typeinfo_FiPPc_i.val, 1),
  { (void *)FS_FiPPc_i_main, FG_main }
};
```

Figure A.18: Stub entry point for the `main` function

function returns nothing (void), the stub function generated by current implementation returns 0 for the caller. Figure A.17 shows a generation rule for stub entry.

A function stub block is also generated for each function definitions. It consists of block header, a pointer to the specific entry point of the function (coerced to the `void *` type) and a pointer to the generic stub entry. Figure A.18 shows an example of the generic entry and the function stub block for function `int main(int, char *)`.

The performance overhead introduced by this stub block seems not to be so large, but further optimization can be considered to remove indirection overhead for type-specific entry points, by placing stub blocks just before the type-specific function entry point. This is easy in assembly language, but is impossible while C compiler is used as back-end code generator. The Glasgow Haskell Compiler [25] performs some dirty trick which post-processes the compiler-output assembly code to achieve this, but this might have severe compatibility problem with various version of underlying C compilers. Future version of Fail-Safe C may implement its own code generator for native assembly languages or utilize some low-level intermediate language like C−− [37, 57] to implement this optimization.

A.2.6 Layout static data onto memory

As well as dynamically-allocated data, all statically-allocated data (global variables and string constants) must have appropriate headers attached. the back-end native C compilers, however, only guarantee a specific data layout inside single variable: relative layout between two or more variables may vary for each compilation. This

```
/* BIG-ENDIAN DEFINITIONS */
#define EMIT_INIT_TWO_WORDS(h,l) { (h), (l) }
#define EMIT_DECL_TWO_WORDS(h,l) h; l


#define EMIT_INIT_i(b,o) {EMIT_INIT_TWO_WORDS((b),(b)+(o))}
#define EMIT_INITPTR(b,o,f) {EMIT_INIT_TWO_WORDS((b)+fsc_canonify_tag(f),(o))}

union fsc_initU_i {
    struct fsc_initS_i {
      EMIT_DECL_TWO_WORDS (word base, word ofs);
    } init;
    value cv;
};
union fsc_initUptr {
    struct fsc_initSptr {
      EMIT_DECL_TWO_WORDS (word base, word ofs);
    } init;
    value cv;
};
```

Figure A.19: Macros and unions used to emit global initializers

means that Fail-Safe C compiler must encode the required memory layout in single variable declaration in usual C syntax.

In addition, C compilers and linkers introduce certain limitation on statically-initialized values. Specifically, addresses of global variables can be cast to word-size integer in static initializers, or added to constant integers, but cannot be multiplied to or divided by constant integers. Further more, static initializers containing any kind of addresses are not permitted for double-word variables. This means that a packed fat pointer pointing to a global variable `v`, that might be expressed like "`(dword)v << 32`", cannot be written directly as a constant.

These problems are solved in the Fail-Safe C compiler by using `unions` and `structs`. To solve first problem, for each unique type T or type $T[n]$ appeared in global declaration, Fail-Safe C compiler generates a temporary structure declaration. The structure have two fields, the first of which corresponds to the block header, and the second contains real data. All references to the global variables are translated to the code referring the second field. The same approach are carried out for type information blocks and static string constants (which are translated to `char[]` global variables).

The solution to the second problem is as follows. for pointers and integers, union types shown in Figure A.19 are defined in standard library. The first field `.init` is used for static initialization. while all runtime reference to this field refer the `.cv` field. Macros are used to absorb the byte-order differences: these macros swap the two arguments on little-endian architectures. Figure A.20 shows the example output code for global initializations.

```
/* input source:
      int a[5] = { 17 };
      int j = 3 * 5 + (int)a;
      int i = (int)&j;
      int *p = &a[3];
*/

struct fsc_storage_Pi_s {
  struct fsc_header fsc_header;
  union fsc_initUptr val;
};
struct fsc_storage_i_5 {
  struct fsc_header fsc_header;
  union fsc_initU_i val[5];
};
struct fsc_storage_i_s {
  struct fsc_header fsc_header;
  union fsc_initU_i val;
};

struct fsc_storage_i_5 GV_a = {
  EMIT_FSC_HEADER(fsc_typeinfo_i.val, 20),
  {EMIT_INIT_i(0, 17)}
};
struct fsc_storage_i_s GV_j = {
  EMIT_FSC_HEADER(fsc_typeinfo_i.val, 4),
  EMIT_INIT_i((base_t)&GV_a.val, 15)
};
struct fsc_storage_i_s GV_i = {
  EMIT_FSC_HEADER(fsc_typeinfo_i.val, 4),
  EMIT_INIT_i((base_t)&GV_j.val.cv, 0)
};
struct fsc_storage_Pi_s GV_p = {
  EMIT_FSC_HEADER(fsc_typeinfo_Pi.val, 4),
  EMIT_INITPTR((base_t)&GV_a.val, 12, 1)
};
```

Figure A.20: An example output of global initialization

A.2.7 Dynamic initializations

Unlike static initializations, dynamic initializations inside function bodies resemble to assignment statements, i.e., expressions for dynamic initializers can be almost any kind of expressions, not limited to constant expressions. Fail-Safe C compiler thus treats dynamic initializers for scalar variables in the same way as usual variable assignments.

Local variables of array type are currently allocated in heap. Each element of initializers for local arrays are analyzed and determined whether it can be treated as static initializers. If it can be calculated as constant value, it is assigned directly into the members of the heap-allocated array. For members which cannot be calculated statically, the corresponding elements are initialized by zero at first and assignment statements for corresponding elements are inserted. Local scalar variables whose address is taken by `&` operator is preprocessed to an array of one element, and thus translated in the same way as other arrays.

An example is shown in Figure A.21. First three elements are initialized statically, and the last element is translated in the same way as an assignment to array, `a[3] = (int)v`. Current implementation of Fail-Safe C does generate a redundant boundary checking code for element assignment. This check can be erased by simple pointer analysis.

A.3 Summary of the current standard library

Various methods are used to implement library functions in the current standard library. The following is a summary for some of standard library functions with explanations on implementation method used.

1. Simple wrapper functions:

   - Ctype functions (`isascii`, `toupper` etc.)
     For these functions, wrappers are suitable to reflect locale support of underlying operating system. Type-specific entry-points of these functions are declared as inline functions for faster execution. The argument, which has type `int`, must be cast to `unsigned char` type before passed to corresponding native functions because the behaviour of native functions for value outside `unsigned char` range is undefined.
   - `fopen`, `fclose`, `ftell`, `fseek`, `fread`, `fgetc`, etc.
     Using abstract type block for `FILE` pointers. Figures A.22 and A.23 show an example implementations for wrapper functions on `FILE` type.

2. Custom implementation provided in native C language:

   - `errno`
     This special variable is implemented using magical blocks. Figures A.24 and A.25 show the current implementation of the `errno`

Original Source:

```
int main(int c, char **v) {
    int a[4] = { 1, 2, 3, (int)v };
    return 0;
}
```

Translated Source (comment inserted):

```
value FS_FiPPc_i_main (base_t FAB_1c, unsigned int FAV_1c,
                       base_t FAB_2v, ofs_t FAV_2v)
{
   auto base_t T2;
   auto base_t T5;
   auto ofs_t T11;
   auto unsigned int T12;
   auto value * T18;
   auto int T39;
   B0:
   T2 = fsc_alloc_stack_block(&fsc_typeinfo_i.val, 4);
   T18 = (value *)T2;
   *(T18 + 0) = value_of_base_vaddr(0, 1); /* first three elements */
   *(T18 + 1) = value_of_base_vaddr(0, 2); /* initialized directly */
   *(T18 + 2) = value_of_base_vaddr(0, 3);

   /* calculating (int)v */
   T5 = base_remove_castflag(FAB_2v);
   T11 = 0 + 4 * 3;
   T12 = (unsigned int)(int)vaddr_of_base_ofs(FAB_2v, FAV_2v);

   /* assignment */
   T39 = is_offset_ok(T2, T11);
   if (!T39) goto LL_37_0;
   *get_realoffset_i(T2, T11) = value_of_base_vaddr(T5, T12);
   goto LL_37_1;
   LL_37_0:
   write_word(T2, T11, value_of_base_vaddr(T5, T12), 0);
   LL_37_1:
   return value_of_base_vaddr(0, (unsigned int)0);
}
```

Figure A.21: Handling of dynamic initializer for local arrays

```
struct typeinfo_init fsc_typeinfo_Sn10stdio_FILE_ = {
    EMIT_HEADER_FOR_TYPEINFO,
    {
        "stdio_FILE",
        TI_SPECIAL,
        NULL,
        4,
        sizeof (FILE *),
        EMIT_TYPEINFO_ACCESS_METHODS_TABLE_NOACCESS
    }
};

struct stdio_FILE_init {
    struct fsc_header header;
    FILE *p;
};

...

FILE **get_FILE_pointer_addr(base_t b0, ofs_t o) {
    base_t b;
    fsc_header *h;
    FILE *p;

    initialize_stddesc();
    b = base_remove_castflag(b0);
    if (b == 0)
        fsc_raise_error_library(b0, o, ERR_NULLPTR, "get_FILE_pointer");
    h = get_header_fast(b);
    if (h->tinfo != &fsc_typeinfo_Sn10stdio_FILE_.val)
        fsc_raise_error_library(b0, o, ERR_TYPEMISMATCH, "get_FILE_pointer");
    if (o != 0)
        fsc_raise_error_library(b0, o, ERR_OUTOFBOUNDS, "get_FILE_pointer");
    return (FILE **)b;
}

FILE *get_FILE_pointer(base_t b0, ofs_t o) {
    FILE *p = *get_FILE_pointer_addr(b0, o);
    if (!p)
        fsc_raise_error_library(b0, o, ERR_OUTOFBOUNDS,
                                "get_FILE_pointer: file already closed");
    return p;
}
```

- The function `initialize_stddesc` (not shown in this figure) prepares three standard file objects, `stdin`, `stdout`, and `stderr`.

Figure A.22: Implementation of the `FILE` abstract type.

```
value FS_FPSn10stdio_FILE_ii_i_fseek(base_t b, ofs_t o,
                                     base_t lb, int lo,
                                     base_t wb, int wo) {
    FILE *p;
    int r;

    p = get_FILE_pointer(b, o);
    return value_of_int (fseek(p, lo, wo));
}

value FS_FPviiPSn10stdio_FILE__i_fread(base_t ptr_b, ofs_t ptr_o,
                                       base_t size_b, unsigned int size_o,
                                       base_t nmemb_b, unsigned int nmemb_o,
                                       base_t fp_b, ofs_t fp_o) {
    void *ptr;
    void *p0;
    FILE *fp;
    unsigned int s;
    unsigned int r;

    fp = get_FILE_pointer(fp_b, fp_o);
    if (size_o == 0 || nmemb_o == 0)
        return 0;

    s = size_o * nmemb_o;
    if (s / size_o != nmemb_o) {
        fsc_raise_error_library(0, nmemb_o, ERR_OUTOFBOUNDS,
                                "fread: I/O size exceeds integer");
    }
    ptr = wrapper_get_read_buffer(ptr_b, ptr_o, &p0, s, "fread");
    r = fread(ptr, size_o, nmemb_o, fp);

    assert(r <= nmemb_o);
    wrapper_writeback_release_tmpbuf(ptr_b, ptr_o, p0, r * size_o);
    return value_of_int(r);
}
```

Figure A.23: Wrapper routines for `fseek` and `fread` functions.

```
value read_fsc_errno_word(base_t base_c, ofs_t ofs) {
    base_t base = base_remove_castflag(base_c);

    if (ofs != 0)
        fsc_raise_error(base_c, ofs, ERR_OUTOFBOUNDS);
    return value_of_base_vaddr(*(base_t *)base, errno);
}

void write_fsc_errno_word(base_t base_c, ofs_t ofs, value v, typeinfo_t ti) {
    base_t base = base_remove_castflag(base_c);

    if (ofs != 0)
        fsc_raise_error(base_c, ofs, ERR_OUTOFBOUNDS);
    *(base_t *)base = base_of_value(v);
    errno = vaddr_of_value(v);
}

struct typeinfo_init fsc_typeinfo_Sn12stdlib_errno_ = {
    EMIT_HEADER_FOR_TYPEINFO,
    {
        "stdlib_errno",
        TI_SPECIAL,
        NULL,
        4,
        4,
        read_dword_by_word,
        read_fsc_errno_word,
        read_hword_by_word,
        read_byte_by_word,
        write_dword_to_word,
        write_fsc_errno_word,
        write_hword_to_word,
        write_byte_to_word
    }};

struct  fsc_storage_Sn12stdlib_errno__s
{
    struct  fsc_header   fsc_header;
    struct  struct_Sn12stdlib_errno_ val;
};

struct fsc_storage_Sn12stdlib_errno__s GV___errno = {
    EMIT_FSC_HEADER(fsc_typeinfo_Sn12stdlib_errno_.val, 0), {0}
};
```

Figure A.24: Implementation of the `errno` special variable (library part)

```
struct __fsc_attribute__((named "stdlib_errno", external)) __stdlib_errno;

extern struct __stdlib_errno __errno;

#define errno (*(int *)&__errno)
```

Figure A.25: Implementation of the `errno` special variable. (include file)

variable. The memory block `GV___errno` contains only the base part of the value. If the block is read, the read access method combines the base part with the current value of the native `errno` variable.

- `malloc`, `free`

  These functions are implemented directry for an obvious reason. By default `malloc` generates an type-undecided block (Section 4.3).

- `printf`, `fprintf`, `vprintf`, `vfprintf`

  The formatting routine for these functions is implemented directly, and these functions use native `fwrite` for output.

  If these functions were written as a wrapper function, these should have handle varargs arguments. However, it is impossible in the C standard to construct varargs arguments or a `va_args` value dynamically by the program.

- `sprintf`

  This function is basically the same as the above functions. A two output routines are provided, both for continuous memory blocks and for generic memory blocks.

  The output string may be arbitrary length, thus it is impossible to guess whether buffer overrun occurs or not before execution.

- `gets`

  This function is implemented using `getchar`, not a native `gets`.

  This function may generate output strings which are arbitrarily long, thus it is impossible to prepare long enough buffers beforehand.

3. Custom implementation written in Fail-Safe C:

   - `strcpy`, `strcat`, `strncmp`, etc.

     Wrappers are inconvenient for these functions, mainly because the outputs may be arbitrarily long. If these functions are written as wrappers, the input strings must be scanned twice, first for determining the input length, and then for the actual operation. Of course it can also be written in native C language, but for those functions providing custom native implementation does not reduce the required safety checks.

Table A.6: Result of the Fibonacci test

	Pentium4		Sparc	
	time	ratio	time	ratio
Native	†1.931 s	(1.00)	5.022 s	(1.00)
Fail-Safe C Std.	2.302 s	1.19		
Fail-Safe C Std. (no asm.)	2.339 s	1.21	4.602 s	0.92
Fail-Safe C Alt.	2.092 s	1.08		

(the average of 5 executions)
(†: the average of 10 executions)

A.4 Result of preliminary micro-benchmarks

As described in Section A.2.1, the encoding of fat integers and pointers are decided by comparing execution performance of several small programs. Three tests are shown here: one is a Fibonacci to check integer operations, and another is a quick-sorting to check pointer operations. Another test, knapsack, is a slightly more larger program which is not originally written for Fail-Safe C. All experiments (unless notified as otherwise) are performed on two different architectures:

- a Linux workstation operating Pentium 4 CPU at 2.8GHz with 1GB of main memory. The versions of the Linux kernel, standard library, and the back-end compiler is Linux 2.4.27, glibc-2.2.5 (Debian woody), and gcc 2.95.4 (with -mpentiumpro option).
- Sun Fire V880 operating four UltraSPARC-III CPUs at 1.2GHz with 8GB main memory. Software versions are SunOS 5.9, gcc-2.95.3 configured in 32bit environment (with -msupersparc option).

A.4.1 Fibonacci

A very simple test which calculates the 30th element of Fibonacci sequence is performed to evaluate base-line performance evaluation and the quality of assembly code emitted by the back-end C compiler. The program implements a simple, well-known recursive method of the calculation.

The result is shown in Table A.6. The execution overhead, relative to the native execution time, is between 10% to 20%.

On recent SPARCv9 CPU, the instrumented code runs faster than original code, Although the number of instructions in instrumented code is significantly larger than native code (Figure A.26). I have run the same binary output on various available Sun workstations, but this trend does not change.

On Pentium 4, the assembly code generated for Fail-Safe C seems to be very clean (Figure A.27), although a significant amount of overhead is observed. Only

```
FS_Fi_i_fib:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1

        cmp     %i1, 1
        ble     .LL100

        add     %i1, -1, %o1

        mov     0, %o0
        call    FS_Fi_i_fib, 0
        mov     0, %i0

        mov     %o1, %l1
        mov     0, %o0
        call    FS_Fi_i_fib, 0
        add     %i1, -2, %o1

        add     %l1, %o1, %o1
        b       .LL115
        mov     %o1, %i1
.LL100:
        mov     0, %i0
        mov     1, %i1
.LL115:
        ret
        restore
```

```
fib:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        mov     %i0, %l0
        cmp     %l0, 1
        ble,a   .LL3
        mov     1, %i0

        call    fib, 0
        add     %l0, -1, %o0

        mov     %o0, %i0

        call    fib, 0
        add     %l0, -2, %o0

        add     %i0, %o0, %i0

.LL3:
        ret
        restore
```

Figure A.26: Two codes generated for Fibonacci on SPARC

```
FS_Fi_i_fib:
        pushl %ebp
        movl %esp,%ebp
        subl $12,%esp
        pushl %edi
        pushl %esi
        pushl %ebx
        movl 12(%ebp),%edi
        cmpl $1,%edi
        jle .L139
        addl $-8,%esp
        leal -1(%edi),%eax
        pushl %eax
        pushl $0
        call FS_Fi_i_fib
        movl %eax,%ebx
        addl $-8,%esp
        leal -2(%edi),%eax
        pushl %eax
        pushl $0
        call FS_Fi_i_fib
        addl %ebx,%eax
        xorl %edx,%edx
        jmp .L155
 .L139:
        movl $1,%eax
        xorl %edx,%edx
 .L155:
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        popl %edi
        movl %ebp,%esp
        popl %ebp
        ret
```

```
fib:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp

        pushl %esi
        pushl %ebx
        movl 8(%ebp),%ebx
        cmpl $1,%ebx
        jle .L3
        addl $-12,%esp
        leal -1(%ebx),%eax
        pushl %eax
        call fib

        movl %eax,%esi
        addl $-12,%esp
        leal -2(%ebx),%eax
        pushl %eax
        call fib

        addl %esi,%eax

        jmp .L6
.L3:
        movl $1,%eax

.L6:
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi

        movl %ebp,%esp
        popl %ebp
        ret
```

The left column is a code generated for Fail-Sate C system (standard encoding). The right column is a code generated by native compilation.

Figure A.27: Two codes generated for Fibonacci on Pentium4

```
.data
.LC6:
        .long 1
        .long 0
.text
FS_Fi_i_fib:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        pushl %esi
        pushl %ebx
        movl 12(%ebp),%ebx
        cmpl $1,%ebx
        jle .L101
        addl $-8,%esp
        leal -1(%ebx),%eax
        pushl %eax
        pushl $0
        call FS_Fi_i_fib
        movl %eax,%esi
        addl $-8,%esp
        leal -2(%ebx),%eax
        pushl %eax
        pushl $0
        call FS_Fi_i_fib
        leal (%eax,%esi),%ecx
        movl %ecx,%eax
        xorl %edx,%edx
        jmp .L117
.L101:
        movl .LC6,%ecx
        movl .LC6+4,%ebx
        movl %ecx,%eax
        movl %ebx,%edx
.L117:
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        movl %ebp,%esp
        popl %ebp
        ret
```

```
fib:
        pushl %ebp
        movl %esp,%ebp
        subl $16,%esp
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%ebx
        cmpl $1,%ebx
        jle .L3
        addl $-12,%esp
        leal -1(%ebx),%eax
        pushl %eax

        call fib
        movl %eax,%esi
        addl $-12,%esp
        leal -2(%ebx),%eax
        pushl %eax

        call fib
        addl %esi,%eax


        jmp .L6
.L3:
        movl $1,%eax


.L6:
        leal -24(%ebp),%esp
        popl %ebx
        popl %esi
        movl %ebp,%esp
        popl %ebp
        ret
```

The left column is a code generated for Fail-Sate C system (alternative encoding). The right column is a code generated by native compilation.

Figure A.28: The code generated for Fibonacci on Pentium4 with the alternative encoding

Table A.7: Result of the Quicksort test

	Non-Cast		Cast Ptr.	
	time	ratio	time	ratio
P4 Native	0.958 s	(1.00)	—	—
P4 Std.	2.287 s	2.38	8.067 s	8.42
P4 Std. (no asm.)	2.255 s	2.35	8.144 s	8.50
P4 Alt.	2.527 s	2.64	8.251 s	8.62
SPARC Native	2.241 s	(1.00)	—	—
SPARC Std.	7.710 s	3.44	22.020 s	9.82

(Native version: the average of 10 executions)
(Fail-Safe C versions: the average of 5 executions)

a few additional instruction is inserted to set base part to 0, compared to a natively compiled code. An output code for alternative encoding seems slightly less efficient than standard encoding at least for human's eye (Figure A.28). A constant $1+0i$, which is for the result of the base cases, is stored in read-only memory area (`.LC6`).

The reason that Pentium4 executes this code faster than the code for the main encoding is a gcc's fault to manage one more callee-save register (`%edi`) in the main encoding, which is not used in the body of function at all. Removing superfluous `pushl`/`popl` by hand gives similar result as alternative encoding (2.069 s).

A.4.2 Quick sorting

This test is performing quick sorting on an array of pesudo-random numbers. The arrays of narrow integers and fat integers, which are initialized to an identical random sequence of integers up to 10000, are passed to the routines compiled by both native compiler and Fail-Safe C. In addition to this, an additional test which intentionally put a cast-flag on a pointer to the passed array is performed. The number of elements is ten million.

The result is shown in Table A.7. The execution overhead which is about 135% of the native execution time is observed. It is also shown that if all memory accesses to the array is performed via access methods, the execution overhead will be about 750% on Pentium 4.

Under this test, the alternative encoding performs worse than the standard encoding. Unlike the case of the Fibonacci test, the reason for the performance difference is slightly more visible: it seems to be the code generated for a operation which composes a base part and a value part to one fat integer. When the alternative encoding is used, gcc fails to optimize the multiplication of real value and purely imaginary value ($0+1i$) and thus generates redundant multiply instructions shown in Figure A.30. The same operation for the standard encoding is defined

```
1   void SWAP(int *x, int *y) {
2     int t;
3     t = *x;
4     *x = *y;
5     *y = t;
6   }
7
8   void qsort_int(int *p, unsigned int len) {
9     int pivot;
10    int i, j, mid;
11    int *l, *r;
12    if (len <= 1)
13      return;
14    if (len == 2) {
15      if (p[0] > p[1]) {
16        SWAP(&p[0], &p[1]);
17      }
18      return;
19    }
20    mid = len / 2;
21
22    if (p[0] > p[mid])
23      SWAP(&p[0], &p[mid]);
24    if (p[mid] > p[len - 1]) {
25      SWAP(&p[mid], &p[len - 1]);
26      if (p[0] > p[mid])
27        SWAP(&p[0], &p[mid]);
28    }
29    pivot = p[mid];
30    l = p; r = &p[len - 1];
31    do {
32      while(*l < pivot)
33        l++;
34      while(*r > pivot)
35        r--;
36      if (l < r) {
37        SWAP(l, r);
38        l++;
39        r--;
40      }
41      else if (l == r) {
42        l++;
43        r--;
44        break;
45      }
46    } while (l <= r);
47
48    qsort_int(p, (r - p) + 1);
49    qsort_int(l, len - (l - p));
50  }
```

Figure A.29: A quicksort test program.

(The base part is in %ecx, and the value part is in %ebx at label .L126. Comments added.)

```
.LC0:
        .long 0
        .long 1         ! A constant (0 + 1i)

...

.L126:
        movl 12(%ebp),%eax          ! an offset in a local variable
        cmpl %eax,-20(%edi)         ! check boundary
        jbe .L133                   !    failed: call an access method
        leal (%edi,%eax,2),%edx     ! calculate real address
        movl %ecx,%eax              ! eax := base
        imull .LC0+4,%eax           ! eax := 1 * base
        imull .LC0,%ecx             ! ecx := 0 * base
        addl %ecx,%ebx              ! ebx := 0 * base + value
        movl %ebx,(%edx)            ! write the value part
        movl %eax,4(%edx)           ! write the base part
```

Figure A.30: A generated code composing a fat integer under the alternative encoding.

(The base part is in %edx, and the value part is in %eax at label .L164. Comments added.)

```
.L164:
        cmpl %edi,-20(%esi)         ! check boundary
        jbe .L171                   !    failed: call an access method
        movl %eax,%ecx              ! ecx := value
        xorl %ebx,%ebx              ! ebx := 0
        movl %edx,%eax              ! eax := base
        xorl %edx,%edx              ! edx := 0
        movl %eax,%edx              ! edx := base
        xorl %eax,%eax              ! eax := 0
        orl %eax,%ecx               ! ecx := value | 0 = value
        orl %edx,%ebx               ! ebx := base | 0 = base
        movl %ecx,(%esi,%edi,2)     ! write the value part
        movl %ebx,4(%esi,%edi,2)    ! write the base part
```

Figure A.31: A generated code composing a fat integer under the standard encoding (without inline assembly code).

Table A.8: Result of the Knapsack test

	Pentium4		Sparc	
	time	ratio	time	ratio
Static, native	0.330 s	(1.00)	3.286 s	(1.00)
Static, Std.	0.784 s	2.37		
Static, Std. (no asm.)	1.076 s	3.26	3.430 s	1.04
Static, Alt.	0.910 s	2.76		
Stack, native	0.330 s	1.00		
Stack, Std.	4.044 s	12.25		

(the average of 5 execution is taken)

by shift instruction, and gcc generates slightly better code for this (Figure A.31), although there are still many redundant logical instructions. Thus I implemented the assembly version of the composition function to remove this overhead. The same trend holds also with gcc version 3.0.4.

It is also confirmed that boundary checking is correctly performed. Examples is shown in Figure A.32 for two cases: one for the simple buffer overrun, and another for the buffer overrun regarding integer overflow.

A.4.3 Knapsack problem

This test program solves "knapsack problem" strictly. The problem is to find a subset of given set of goods which gives maximal total value within given limit for total weight. The program uses recursive search of the possible solution space with branch cutting based on upper bounds of possible solution. The program declares a structure of two integers and one double-precision floating-point value, which gives 3/2 ratio of the real size to its virtual size. There is no internal pointers to the array of this structure in the program.

A recursively-called function (`find_ans`) in the program declares one array of 1000 integers as a local variable, and its address is passed to a subroutine (`try_greedy`). The value in the array is not used for recursions, and the address of the array is not leaked outside those two functions, thus it can be either statically allocated or stack-allocated in theory. As described in Section A.2.3.1, the array is heap-allocated in the translated program. Hereafter the original program is called "stack" version, and the program modified to declare the array as `static` is called "static" version. The input data for performance evaluation contains 500 items of similar value/weight ratios and similar weights, which gives bad condition for branch cutting.

The result of the experiments is shown in Table A.8. On Pentium 4, the static version shows gives an overhead slightly more than twice of original execution time, which is in the expected range. However, the stack version gives overhead

```
% ./qsort 5 6
native: 0 msec

--------------------------------
Fail-Safe C trap: access out of bounds
  Address: 0x805dfa0 + 20
  Cast Flag: not set
  Region's type: int
           size: 20 (FA 20, ST 20)
           block status: normal

backtrace of instrumented code:
./qsort(fsc_raise_error_library+0x15f)[0x804b277]
./qsort[0x804b2ce]
./qsort(read_word_fat_int+0x46)[0x804a136]
./qsort(FS_FPii_v_qsort_int+0x14d)[0x8049945]
./qsort(main+0x19c)[0x8049e4c]
/lib/libc.so.6(__libc_start_main+0xbb)[0x4006614f]
./qsort(backtrace_symbols_fd+0x59)[0x8049531]
(7 entries)
--------------------------------

Abort
% ./qsort 5 2147483648
native: 0 msec

--------------------------------
Fail-Safe C trap: access out of bounds
  Address: 0x805dfa0 + 4294967292
  Cast Flag: not set
  Region's type: int
           size: 20 (FA 20, ST 20)
           block status: normal

backtrace of instrumented code:
./qsort(fsc_raise_error_library+0x15f)[0x804b277]
./qsort[0x804b2ce]
./qsort(read_word_fat_int+0x46)[0x804a136]
./qsort(FS_FPii_v_qsort_int+0xdd)[0x80498d5]
./qsort(main+0x19c)[0x8049e4c]
/lib/libc.so.6(__libc_start_main+0xbb)[0x4006614f]
./qsort(backtrace_symbols_fd+0x59)[0x8049531]
(7 entries)
--------------------------------

Abort
```

Figure A.32: An example of boundary overflow detection in quick-sorting

four times as many as the static version, which indicates that the overhead of heap allocation of local variables in frequently-called function cannot be neglected. Under this test, the alternative encoding outperformed the standard encoding.

The author has also investigated the overhead caused by a fractional ratio on offset conversions, by modifying the output code of Fail-Safe C compiler by hand to add a padding element and make the real size of the structure just twice of the virtual size. The result was 1.072 s (the average of 10 executions), it means there is no observable overhead. The assembly code generated for reading an element of the array of the structure is like following:

```
shrl $1,%eax
leal (%eax,%eax,2),%eax
movl GV_data+40(%eax),%eax
```

The fractional multiplication is done by using `shrl` (shift right) and the powerful `leal` (load effective address) instruction in i386 architecture, to avoid use of multiplication instruction.

On SPARC architecture, there is only a little overhead ($\sim$4%) observed. The author has no knowledge about the exact reason because the output code is already huge with this program (2913 lines of C code generates 5070 lines of assembly output). However, when compared with the results on Pentium 4 architecture, it seems to be that there is some reason that the native version of the program behaves badly on this architecture. In fact, the native version of Knapsack on SPARC runs almost 10 times slower than Pentium 4, while Quicksort runs only 2.3 times slower. Comparing the Fail-Safe C output using standard encoding, those figures are about 3.2 and 3.4 which seems natural.

A.5 Further extensions to the implementation

There are several possibilities of studies which can improve implementations of the Fail-Safe C systems. In this section, some of these possibilities are discussed.

A.5.1 Local optimization

There are many studies (for example, [60, 72]) on local analyses for reducing redundant boundary checks proposed for various safe languages. Most of these can be applied to Fail-Safe C to reduce runtime overhead. However, there is one big difference between the semantics of Fail-Safe C and other safe languages. On other safe languages, failure on the boundary check is a fatal error: it immediately means the failure of the memory access, and the program executions are either terminated, or aborted from current scope by raising exceptions. Thus most (possibly all) of the proposed optimizations assume that when program execution reaches some location in the program, all preceding boundary checks in the program are succeed. It means if a boundary check for the same memory address is exist in such preceding checks, the current check will never fail.

Table A.9: Preliminary result of the local optimization in Quicksort test

	standard		optimized		optimize
	time	ratio	time	ratio	ratio
Native	0.958 s	(1.00)	—	—	—
Without Cast flag	2.255 s	2.35	2.109 s	2.20	−6.5%
With Cast flag	8.144 s	8.50	8.130 s	8.49	(−0.2%)

(Native version: the average of 10 executions)
(Fail-Safe C versions: the average of 5 executions)

On the contrary, the failure of the access check may be non-fatal in Fail-Safe C. As shown in Figure 4.4 in page 44, the failure of the inlined access check in Fail-Safe C only means the situations that some other methods for memory access is needed, which may either fail or succeed. Execution paths of the program after check failure merge into its original execution pass, and thus future boundary check for the same memory location may fail again and thus may not be removed.

To apply existing approach for boundary check optimization to Fail-Safe C, there are two possible approaches to be taken. One possibility is to analyze program and find boundary checks on which failed check always means a fatal error. For example, if a pointer to simple type like `char` is known to be never cast, the invocation of access methods for this pointer always leads to fatal errors, because there is no possibility that the access succeeds. Boundary checks of such cases can be used as a source information for optimizations. The another, more general approach is to apply code duplication. As shown in Figure A.33, the compiler can duplicate all code of the function to “fast code” (which will initially be executed) and “slow code”, and make all invocation to access methods transfer execution to the “slow code”. After this code duplication, the property required by existing optimizations are recovered on the “fast code”. As long as such optimizations are done locally inside single function, executing `return` instructions inside the “slow code” can transfer to the “fast code” of the caller functions.

In this way, a fast code may access the contents of memory blocks already marked as deallocated. This is unfavorable, but the safety of the execution is still maintained, because (1) these deallocated blocks are still on memory until the pointers pointing to the memory blocks are disappeared, and (2) the deallocation does not affect the block contents itself.

A preliminary experiment is performed to evaluate an effect of this optimization. I have modifed the output of the Fail-Safe C compiler by hand to implement the code duplication method shown in Figure A.33, and removed two obviously redundant boundary checks in the `SWAP` function in Quicksort program (Figure A.29). Under this experiment, the standard encoding of fat values is used on the Pentium 4 machine.

The result is shown in Table A.9. It shows about 6.5% reduction of execution

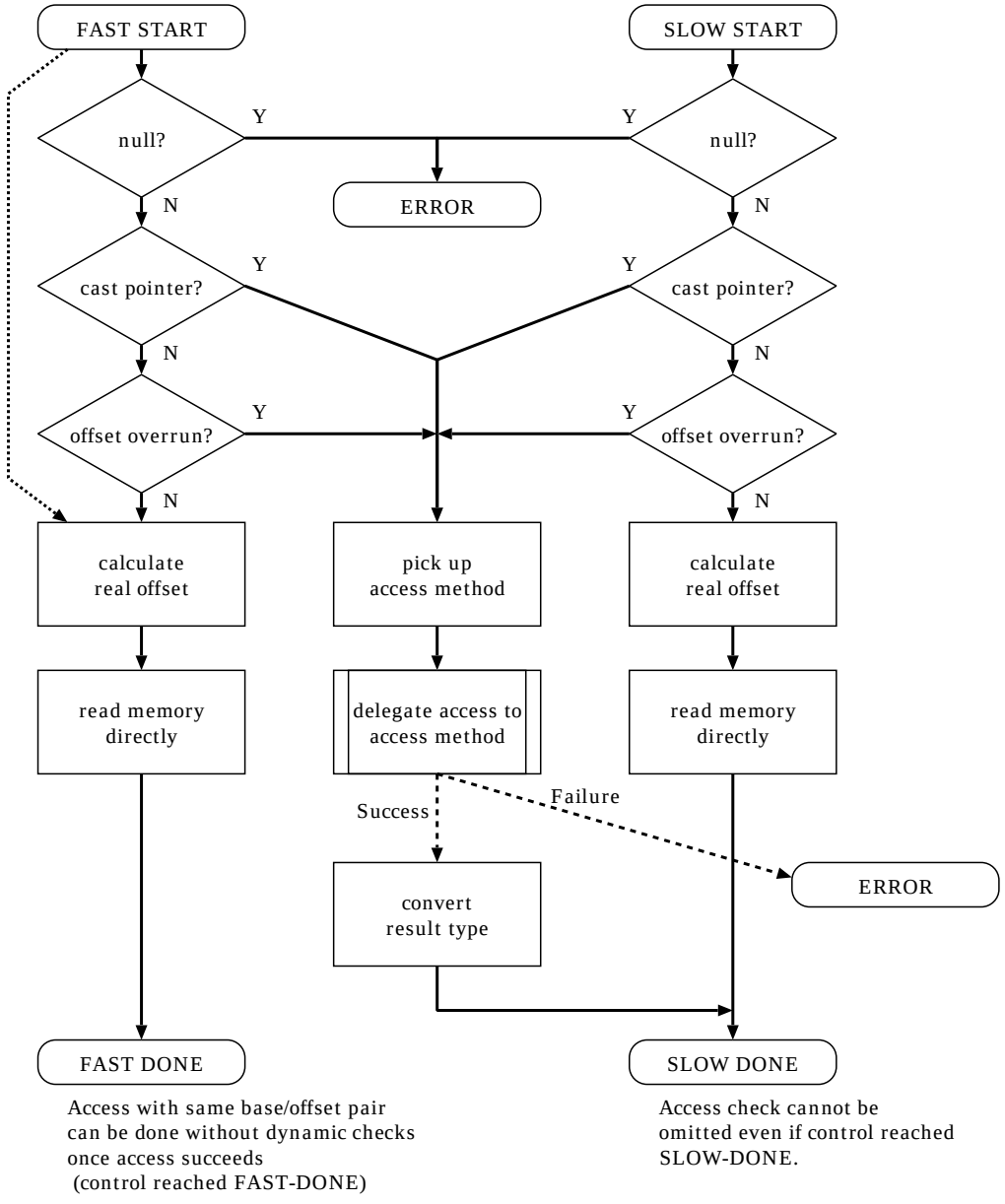


Figure A.33: Code duplication for boundary access reduction

time, with non-cast pointers. The handling of non-cast pointers are not changed, and the results has shown that, at least in this small example, the the growth of code size caused by code duplication did not affect the performance.

Another important issue which should be noticed is the handling of integer overflow conditions which occur during calculation of pointer offsets. For example, in the quicksort test program shown in Figure A.29 (at Page 115 in Section A.4.2), the offset of `r` in line 34 may point outside memory region. This will occur when the offset of `p` is 8, `len` is $2^{31}-1$ (assuming word size to be 32 bits), and `p` contains 4 elements $\{3,2,1,4\}$. In this case, `mid` becomes $2^{30}-1$, while `len - 1` becomes $2^{31}-2$. As the virtual offset of the elements in integer arrays is 4 times the index, the offsets of `&p[mid]` and `&p[len - 1]` become $2^{32}+4$ and 2^{33}, which will be rounded to 4 and 0 respectively. Thus, the accesses during pivot selection in lines 22–28 will *succeed*. The values in the array are not modified during pivot selection, and the `pivot` becomes 2. The loop at line 32 terminates without access violation (`l` pointing the value 4 at offset 12). As the value `p[len - 1]` is 3, the the loop condition at the first iteration in line 34 holds, and `r` will be decremented to have offset $2^{32}-4$ and cause a buffer overrun error. A correct handling for such integer overflow is possible but complex, and increases much number of required boundary checks which are not required under ideal consideration on integers.

An obvious exception to this is the access to the same element in memory blocks which occurs shortly. There is no fear about the complication from integer overflow in such cases, and such cases appear very frequently in programs (e.g., modifying the elements in array.) The optimized part in quicksort test above is an instance of this pattern.

A.5.2 Global optimization

A.5.2.1 Value analysis

The output program code of the Fail-Safe C has many redundant data. For example, the following optimization may be possible.

- Most integer variables in programs will only contain non-pointer values or values which are never used as pointers. There is no need to add base fields for those variables.
- In modern use of the C language, many data tend to be represented as a set of heap-allocated non-array values. Pointers pointing to only these memory area do not need offset fields.

Some of these optimizations inside single function is already done, implicitly by the design of translations in the compiler, but to perform these optimization throughout a program, the compiler requires global knowledge on possible values

of each variables. There are many previous studies on global analysis of C programs, which can be applied to Fail-Safe C. For example, the type system proposed for CCured [49, 18] can be altered for Fail-Safe C.

One noticeable fact about the application of these analysis for Fail-Safe C is that the analyses applied are not needed to be conservative in general. That is, although the results of analyses are used for safety enforcement, the analyses are not needed to care about ill-typed accesses performed by programs. Restricted domains of values derived from those analyses can be enforced by access methods, because all ill-typed accesses are handled only through access methods, which can report error condition and halt the program execution. This can greatly reduce the false possibility of values stored in variables and may improve the quality of the analyses results.

A.5.2.2 Temporal analyses

There are many instances of local variables whose address is taken by `&` operator in existing programs, because it is very common practice in C programs to allocate temporary arrays inside functions, perform local computation on the arrays, or pass an address of such local variables to subroutines or library functions to receive a result by modification of the variables via passed pointer.

Currently, such local variables are always heap allocated to avoid dangling pointers. However, it sometimes imposes a relatively large performance penalty. Even in simple program, the caused overhead exceeds twice the execution time (e.g., Knapsack test in Section A.4.3). However, most of those variables can be actually stack-allocated because all pointers to the variables cease to exist before the variable is deallocated (it should be so because these variables are stack allocated in native compilation!), and in many cases the possibility of safe stack allocation can be proved by kinds of temporal analysis, region inference [70, 8] or escape analysis [52, 59, 31].

The analysis applied to Fail-Safe C should be inter-procedural one because addresses of local variables are casually passed to another functions. It is also guessed that the property should be maintained between separately compiled modules as possible, because most of pointer arguments for functions in the standard library are in fact “non-escaping”—e.g. `printf`, `strcpy` and others. For safety, such safety-related properties should be encoded in the mangled name used in Fail-Safe C by extending the translation rule shown in Section A.2.2.

A.5.3 True support for separate compilation

Separate compilation is a common practice for developing large programs. Almost all C programs are coded using several modules (compilation units). Under usual C compiler, several modules for a program are linked into one execution binary by performing a unification on every symbol occur in the compilation units, making all references to the same symbol point to the same location.

However, this simple schema is not applicable for Fail-Safe C, because separate compilation in the native C implementation is indeed unsafe. This holds even for the C++ language, whose compilers embed type names into the output binaries. There is no guarantee that modules in a program is compiled with a same set of type definitions. Even worse, using the same name for two or more incompatible structures (in several disjoint set of modules) is possible, although it is disallowed in the language specification. Such a name conflict is not rare in existing programs, especially when a name of struct is used both in a user code and in an external library.

There are two possible method to solve this problem. One method is so-called "whole-program compilation", i.e., to compile the whole program at once. Existing work such as CCured [49, 18] also use this approach. The merits of this approach are that it simplifies the guarantee of safety, and that it enables various global optimizations, such as one described in Section A.5.2. However, it also has several demerits: it changes the compiler interface dramatically, it incurs longer compilation time, and it makes impossible to reuse compiled modules for several execution programs. At least, there must be a support for true separate compilation of library modules.

The another method is to handle separate compilation in linking stage. Existing work on functional language by Leifer et al. [40] uses a hash value of a canonically defined representation of the structure to support type-safe linking of separately-compiled modules. However, this simple method seems not to be work with the existence of abstract declaration in the C language. A possible solution is to compile every module assuming that every struct is disjoint to any structs in another module, and then unify any compatible types at the linking time. Although this method reduces opportunities for global optimization, it does not damage the basic design of Fail-Safe C, because distinction between non-cast and cast pointers in Fail-Safe C is solely a runtime property which can be checked in a light-weight operation (described in Section 4.2). When compiling each modules, the compiler can assume that cast fat pointers may be passed to externally exported functions with almost no runtime overhead. This is not true for CCured which heavily relies on the compile-time distinction between cast and non-cast pointers.

Furthermore, it is possible to annotate the possibility to optimize explicitly.At least, the annotation can be employed for standard libraries, system calls, and other library wrappers, because they are already prepared specially for Fail-Safe C and they are separately compiled even in the current system.

A.5.4 Multi threading

Multi-thread programming is getting more and more common these days. Most of the Unix-like operating systems currently on production line already have a POSIX-define interface for multi-threading. However, ensuring safety for multi-thread program is more difficult than for single-threaded programs, especially on the management of the consistency of safety-related values.

Current implementation schema of the Fail-Safe C runtime is carefully designed for future support of multi-threading in several places, although it is not yet supported. Especially, the design does not require operation on exclusive locks for usual memory accesses. The design assumes that underlying hardware ensures atomic word-width accesses without any additional consideration. The race will happen between two or more of the following accesses:

- direct access to memory
- access via access methods
- updating type of type-undecided blocks
- deallocating (forbidding further access for) blocks
- allocating additional base storage

The following considerations on implementation are sufficient for multi-threading support.

- Double-word fat pointers in the data area must be atomically accessed, unless the value accessed are statically known to not to be cast.
- Exclusive locks on a memory block must be taken for updating the type of the block, allocating additional base area, and deallocating the block. In addition, the first two operations must be aware that another thread might already have done the same job while waiting for the lock.
- The update of the *type* field and *ptr-additional-base* field must be done as a final operation.

The conditions on each combinations under above treatment is investigated separately.

Direct-access to direct-access If a read access and a write access to the same word cause race condition, There is a chance that the value neither original value nor currently-written value will be read. In this case, the read value will be a mixed combination of base value and offset value of those two values.

If either an old or new value may be cast, and if a double-word access incurs a race condition, there is a chance that a base value without cast value is paired with an unaligned offset value which requires a cast flag. Thus, an atomic operation for double-word might be required, if either of the values might have cast flag set.

If no cast flags are involved (fat integers, or values assured by static analysis), no additional treatment is required. Although the resulting value is unexpected in usual sense, it does not break safety conditions in this case. The similar thing happens on a write-to-write race condition.

Direct-access to Access-method, Access-method to Access-method The memory accesses in the access method can be considered in the same way as that for direct access.

Type-update to Direct-access Basically, no race of this kind will occur, because no pointers without cast flag may point to a type-undecided blocks. However, the order of updating header values will be important: before updating *type* field, *structured-limit* field and *total-limit* must be properly updated.

Type-update to Access-method The race between type-update and access-method for types other than the undecided type is avoided, in the way shown above. invocation of access method for undecided type will cause type-update to type-update race.

Type-update to Type-update This race is critical and a mutual exclusion is required. Furthermore, if an access method which wins the mutual exclusion updates type, second (and later) access method must see updated type, to prevent performing type-update twice. Thus, a proper implementation of type update must follow the following order:

1. Take an exclusive lock of the block.
2. Double-check the type field in the block.
3. If the type is updated, release the lock, and call the access method associated to the new type.
4. If not updated, initialize a block contents and update all fields other than type information.
5. Update the type field of the block.
6. Release the exclusive lock.

Deallocation to Direct-access No additional consideration are required. The deallocation only modifies the *fastaccess-limit* and the *runtime-flags* fields in the block header, and the contents of the block is not modified during deallocation. Thus, direct access will see the value of *fastaccess-limit* as either 0 or the original value. In the former case, the access invokes access methods and causes runtime error because of accessing deallocated blocks. In the latter case, the access succeeds even after deallocation.

Deallocation to Access-method No additional consideration required, too. The access methods only see the *structured-limit* and *total-limit* for accessing contents of blocks. Thus, modification to *fastaccess-limit* by deallocation does not break the operation of access methods. The race on the *runtime-flags* determines whether the access is granted or not, which is natural. (Atomicity on the update of the flag is assumed.)

Deallocation to Type-update Mutual exclusions are required.

Deallocation to Deallocation No additional consideration required.[6]

Additinal-base-allocation to Direct-access Direct memory accesses do not touch the additional base area.

Additinal-base-allocation to Access-method The race on the *ptr-additional-base* determines whether the access methods see the additional base or not. This means that the contents of the additional base area must be initialized before setting the address to the *ptr-additional-base* field.

Additinal-base-allocation to Type-update This race will not occur.

Additinal-base-allocation to Additinal-base-allocation Mutual exclusions are required. The double check for the field update must be done, in the same way as described in the type-update race.

Additinal-base-allocation to Deallocation Basically, no care will be required.[7]

On the SPARC architecture, it is relatively easy to implement atomic double-word access. In fact, the generated code in current compiler is already using double-word memory access instructions (e.g., `std` and `ldd` instructions) which is guaranteed to be atomic [69, Sections A.70.5 and A.70.12]. On Intel IA32 architecture, however, there are no generic double-word atomic memory-access instructions on integers [32]. The possible alternatives are (a) a complicated `CMPXCHG8B` instruction introduced in Pentium, (b) `FIST/FILD` instructions on floating-point processor, or (c) MMX or SSE multimedia instruction extensions. Among those, the choice (b) seems mostly unsuitable, because it is no way to move the values in floating-point registers to general-purpose registers without using external memory locations. The method (a) is current used in Linux kernel, but it requires complex coding shown in Figure A.34, because it is a "compare-and-exchange" instruction, not a simple store/load instruction. The alternative (c) may be useful if newer SSE extensions can be used, however it seems unrealistic with older MMX extensions because it cannot be coexist with floating-point operations.

A.5.5 Compiling to more low-level language than C

Current choice of C language for output language of the Fail-Safe C compiler seems to be a realistic solution, but at the same time the system suffers several restrictions from this choice: it is practically impossible to implement precise garbage collectors (already discussed in Section A.1.3), function stub blocks must be placed separately from main function bodies (Section A.2.5), and it is hard to control the backend compiler to generate optimal code for various places, such as overflow detection or handling of double-word values including fat pointers.

[6] This race conditions may be mutually-excluded by lock acquisition required for other race conditions.

[7] The same as above.

An excerpt from `include/asm-i386/system.h` in Linux 2.4.27.

```
/*
 * The semantics of XCHGCMP8B are a bit strange, this is why
 * there is a loop and the loading of %%eax and %%edx has to
 * be inside. This inlines well in most cases, the cached
 * cost is around ~38 cycles. (in the future we might want
 * to do an SIMD/3DNOW!/MMX/FPU 64-bit store here, but that
 * might have an implicit FPU-save as a cost, so it's not
 * clear which path to go.)
 *
 * chmxchg8b must be used with the lock prefix here to allow
 * the instruction to be executed atomically, see page 3-102
 * of the instruction set reference 24319102.pdf. We need
 * the reader side to see the coherent 64bit value.
 */
static inline void __set_64bit (unsigned long long * ptr,
                unsigned int low, unsigned int high)
{
        __asm__ __volatile__ (
                "\n1:\t"
                "movl (%0), %%eax\n\t"
                "movl 4(%0), %%edx\n\t"
                "lock cmpxchg8b (%0)\n\t"
                "jnz 1b"
                : /* no outputs */
                :       "D"(ptr),
                        "b"(low),
                        "c"(high)
                :       "ax","dx","memory");
}
```

Figure A.34: An atomic double-word memory store in IA32 architecture

All of these problems can be solved when the output language is changed to the assembly languages of underlying hardware, which requires extremely huge effort to implement. This is not only because assembly languages are complex, but also because the Fail-Safe C compiler relying backend C compilers for various low-level handling of native architectures, for example instruction scheduling, register allocation and spilling, peep-hole optimizations, choice of the strength to perform common value eliminations, and so on. Therefore, porting Fail-Safe C compiler to every architectures the users use seems not a worth effort to do. The author is currently considering use of an intermediate language which is between C and the assembly languages. As already mentioned (in Section A.1.3), C−− [37, 57] seems to be one of possibilities for this purpose.

Appendix B

Perspectives on derived research

There are several potential extensions of the Fail-Safe C system that could effectively utilize various aspects of Fail-Safe C. Some of these possibilities are discussed below.

B.1 Language extensions

Extensions to the input language of Fail-Safe C, which is currently the pure C language, might make the system more useful (or interesting). Obviously these extensions will require some modifications of the source code of programs, and thus be a slight diversion from the original design basis—no source modification, gain complete safety—, but it might still be very useful to allow programmers to use extended features with less modification to original source, compared to the modification required when rewriting whole programs to other languages such as Java.

B.1.1 Recovery from failure

Fail-Safe C currently halts program execution whenever a runtime error is signaled. This behavior is based on the current design principle. Moreover, this is a realistic choice when assuming the input language is the pure C language because it is practically impossible to guess what countermeasures are needed to avoid failure once a fatal access error is detected.

On the other hand, many would prefer that the behavior after failures be controllable at user's discretion. For example, a possible recovery process for thread-based web servers might be to silently terminate only failed threads, and allow the master thread to revive dead sub-threads. Alternatively, the server may return a result to clients saying that a fatal error has occurred during processing. It is possible to implement such a user-directed failure rescue feature by defining an language extension for the usual C language. One possibility is to introduce the exception

handling syntax from the C++ language and map runtime memory access errors to a predefined exception.

B.1.2 Incorporation with high-level security mechanisms

Many studies have been on security enforcement and verification. For example, language-based studies have aimed at ensuring confidential data in programs cannot leak to low-level output by analyzing/checking the data flow or information flow [71, 44, 61].

Most of this work has been done using a safe language (either concrete or abstract) as a base language, so findings are not directly applicable to the C language. More precisely, the proofs regarding the satisfaction of security properties typically assume that the running program does not cause undefined behavior such as buffer overruns or other low-level bugs. There are many instances, though, of security holes which bypass high-level security protection features (e.g., security zones or per-domain separation of scripting languages in web browsers) through low-level invasion (e.g., a buffer overrun). Analyzing C programs based on these high-level theories, assuming no existing buffer overruns, is still valuable as an effort to search for programming bugs, but not as a practical guarantee of security properties. Fail-Safe C can be used to close such loopholes, so it can help make the application of those high-level security theories to the C language a form of real security protection.

B.2 Altering semantics

Currently the main design goal of Fail-Safe C is to maintain the highest possible compatibility with the native semantics of C language as high as possible. Moreover, the error condition signaled by the Fail-Safe C system is designed to be easily understood by users familiar with the usual C language. However, setting a slightly different design goal might also leads to interesting developments. The entire Fail-Safe C compiler system can be thought of as a powerful tools for modifying the runtime semantics of C language in various ways, and the object-oriented design of memory blocks can be considered a strong weapon for modifying the runtime semantics of the heap area, which cannot be easily achieved by using simple preprocessor to the C language. Several possible modifications to the semantics of Fail-Safe C are discussed below.

B.2.1 *Fail-Soft* C—partial remediation of buffer-overrun problems

Through the combination of object-oriented memory block design, implementation of the remainder area in memory blocks, and the ability to implement several different semantics in a single memory block depending on the offsets, we can partially allow writing beyond a memory block's boundaries. Current access methods deny all access to the memory area which is outside the boundary of a memory

block. Instead, an implementation can dynamically allocate additional space to allow buffer overflows while preserving safety. It is not always possible to support all kinds of buffer overflow, though, because memory resources are limited. Still some types of simple buffer overflow can be remedied without loss of operations.

There are many ways to implement this feature. For example, the data format for extra data may be either hash (to allow sparse, distributed invalid memory access, like that found in Sendmail (see Section 5.1.1)), or array (to allow only simple cases of (string) buffer overrun, but faster).

Another consideration is memory addressing. One possibility is to simply use current addressing format. While this would make implementation easier, and would maintain higher compatibility with current native semantics for programs without problems, but a drawback is that valid fat pointers corresponding to one virtual address (the sum of the base and the offset) will no longer be unique. In other words, two different addresses might correspond to one integer. To avoid this difficulty, another form of addressing is also possible. Assuming a 32b-it architecture, we can extend both the base and offset to 64 bits, while keeping a 32-bit size for the default integer type. Then, only the higher 32 bits of base addresses and the lower 32 bits of offsets will be used (limiting the possible range of writable offsets to $[0, 2^{32}-1]$ or $[-2^{31}, 2^{31}-1]$). This mapping ensures the virtual addresses of the tops of different memory blocks will differ by at least 2^{35} bytes, and thus keeps the addresses of memory blocks disjoint. Of course, this mapping changes existing native setting of word size and thus only accepts "portable" programs without modification.

B.2.2 Fail-Safe C on Java (or Scheme)

The output language of the compiler is not limited to C native assembly languages, or other low-level intermediate languages. As the data format of the memory blocks in Fail-Safe C is strictly formatted and typeable, it can be mapped to data structures in other high-level languages. The representation of the fat pointer is also mappable to the references (although embedding a cast flag in the base field is not possible). As a consequence, an entire program can be compiled into languages such as Java and Scheme. One possible positive benefit of this would be achieving safe interoperability between such a safe language and C language. Another consequence is that the mapping to the safe language would provide indirect proof the safety of the Fail-Safe C semantics. This is a promising subject for further research in the near future. term research.

Bibliography

[1] Advanced Micro Devices, Inc. AMD 64 and enhanced virus protection. `http://www.amd.com/evp`.

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

[3] Starr Andersen and Vincent Abella. Changes to functionality in Microsoft Windows XP Service Pack 2, part 3, August 9, 2004. `http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx`.

[4] A. W. Appel. Foundational proof-carrying code. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[5] 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.

[6] Joel Bartlett. Mostly-Copying garbage collection picks up generations and C++. Technical report, DEC WRL, 1989.

[7] Brian N. Bershad, Craig Chambers, Susan J. Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN - an extensible microkernel for application-specific operating system services. In *Proc. of ACM SIGOPS European Workshop*, pages 68–71, September 1994.

[8] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1996.

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

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

[11] Brandon Bray. Compiler security checks in depth, February 2002. http://msdn.microsoft.com/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp.

[12] BYTE Magazine. BYTEmark Benchmarks. http://www.byte.com/bmark/bmark.htm.

[13] CAN-2002-0702 (format string vulnerabilities in the logging routines for dynamic dns code). An entry candidate in *Common Vulnerabilities and Exposures*, July 16, 2002. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0702.

[14] CERT/CC. Double free bug in zlib compression library. CERT Advisory CA-2002-07, July 20, 2002. http://www.cert.org/advisories/CA-2002-07.html.

[15] CERT/CC. Format string vulnerability in ISC dhcpd. CERT Advisory CA-2002-12, October 7, 2002. http://www.cert.org/advisories/CA-2002-12.html.

[16] CERT/CC. Heap overflow in cachefs daemon. CERT Advisory CA-2002-11, May 14, 2002. http://www.cert.org/advisories/CA-2002-11.html.

[17] CERT/CC. Double-free bug in CVS server. CERT Advisory CA-2003-02, March 27, 2003. http://www.cert.org/advisories/CA-2003-02.html.

[18] 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.

[19] Intel Corporation. Execute disable bit functionality blocks malware code execution. http://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf.

[20] 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.

[21] CVE-2001-0653 (sendmail 8.10.0 through 8.11.5, and 8.12.0 beta, allows local users to modify process memory). An entry in *Common Vulnerabilities and Exposures*, March 9, 2002. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0653.

[22] CVE-2002-0033 (heap-based buffer overflow in cfsd_calloc function of Solaris cachefsd). An entry in *Common Vulnerabilities and Exposures*, April 2, 2003. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0033`.

[23] Igor Dobrovitski. Exploit for CVS double free() for Linux pserver. A message posted to Bugtraq mailing list, February 2, 2003. `http://www.securityfocus.com/archive/1/309913`.

[24] Noah Friedman. ssh 1.2.22: premature memory deallocation. A message posted to Secure-Shell Mailing List, February 12, 1998. `http://www.securityfocus.com/archive/121/230289`.

[25] The Glasgow Haskell Compiler. `http://www.haskell.org/ghc/`.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[27] 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.

[28] Helge Hafting. Re: Unexecutable Stack / Buffer. A message posted to Linux Kernel mailing list, January 20, 2000. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0001.2/0916.html`.

[29] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. Technical Report YALEU/DCS/TR-1224, Dept. of Computer Science, Yale University, 2002.

[30] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

[31] P. Hill and F. Spoto. A foundation of escape analysis. In *Proceedings of 9th International Conference on Algebraic Methodology and Software Technology (AMAST2002)*, 2002.

[32] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual, 2004.

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

[34] International Organization for Standards and International Electrotechnical Commission. Programming languages — C. ISO/IEC Standard ISO/IEC 9899:1999.

[35] 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.

[36] 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.

[37] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C−−: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.

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

[39] Yoshinori Kobayashi. An efficient garbage collector in the presence of ambiguous references. Master's thesis, University of Tokyo, February 2002.

[40] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of 8th ACM SIGPLAN International Conference on Functional Programming (ICFP2003)*, August 2003.

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

[42] Toshiyuki Maeda and Akinori Yonezawa. Kernel Mode Linux: Toward an operating system protected by a type theory. In *Proceedings of the 8th Asian Computing Science Conference (ASIAN '03)*, volume 2896 of *Lecture Notes in Computer Science*, pages 3–17, December 2003.

[43] Uwe F. Mayer. Linux/Unix nbench. `http://www.tux.org/~mayer/linux/bmark.html`.

[44] John McLean. Security models and information flow. In *IEEE Symposium on Security and Privacy*, pages 180–189, 1990.

[45] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proc. of ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[46] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. of Types in Compilation*, pages 28–52, 1998.

[47] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.

[48] George Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, January 1997.

[49] 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.

[50] Michael Norrish. *C formalized in HOL*. PhD thesis, University of Cambridge, December 1998. Available as a Technical report UCAM-CL-TR-453 from Computer Laboratory, University of Cambridge.

[51] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

[52] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 116–127, 1992.

[53] 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.

[54] Alexandre Petit-Bianco. No silver bullet – garbage collection for java in embedded systems. `http://gcc.gnu.org/java/papers/nosb.html`.

[55] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[56] Projet Cristal, INRIA Rocquencourt. The Caml language. `http://caml.inria.fr/`.

[57] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00)*, June 2000.

[58] Sergei Romanenko, Claudio Russo, Niels Kokholm, and Peter Sestoft. Moscow ML. `http://www.dina.kvl.dk/~sestoft/mosml.html`.

[59] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'88)*, pages 285–293, January 1988.

[60] 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.

[61] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[62] Martin Schulze. cvs – double freed memory. Debian Security Advisory DSA 233-1, January 21, 2003. `http://www.debian.org/security/2003/dsa-233.en.html`.

[63] SecurityFocus. Sendmail Debugger Arbitrary Code Execution Vulnerability, August 17, 2001. `http://www.securityfocus.com/bid/3163`.

[64] Sendmail, Inc. and the Sendmail Consortium. Sendmail. `http://www.sendmail.org/`.

[65] Fermín J. Serna. ISC dhcpdv3, remote root compromise. Next Generation Security Technologies security advisory, June 6 2002. `http://www.ngsec.com/docs/advisories/NGSEC-2002-2.txt`.

[66] Standard ML of New Jersey. `http://www.smlnj.org/`.

[67] Kohei Suenaga, Yutaka Oiwa, Eijiro Sumii, and Akinori Yonezawa. The interface definition language for Fail-Safe C. In *Proceedings of International Symposium on Software Security (ISSS2003)*, volume 3233 of *Lecture Notes in Computer Science*, pages 192–, November 2003.

[68] Sun Microsystems, Inc. Security in the Solaris 9 operating system data sheet. `http://wwws.sun.com/software/solaris/9/ds/ds-security/`.

[69] Sun Microsystems, Inc. UltraSPARC III Cu Processor User's Manual, January 2004.

[70] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[71] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[72] 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.

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