# メモリ安全なC言語処理系 Fail-Safe C の実用プログラムへの適用のための改良

大岩 寛
産業技術総合研究所

**概 要**

Fail-Safe C は、完全なメモリ安全性を実現する ANSI C 準拠の処理系である。ANSI C 言語の仕様で定められた全てのメモリ操作（キャストや共用体を含む）に対しその安全性を保証し、全ての危険なメモリアクセスを事前に検知し防止する。このコンパイラを用いることでプログラマは、既存のプログラムを大幅に書き換えたり別の言語に移植したりすることなく、そのままプログラムを安全に実行することができる。　本発表では主に、この処理系の概要のほかに、実際の大規模プログラムに適用する過程で生じた問題点やそれに対する改良、移植性や実用性を向上させるための取り組みについても報告する。

*Fail-Safe C* is a completely memory-safe, completely ANSI-compatible compiler system for the C language. It detects and disallows all unsafe operations, yet conforms to the full ANSI C90 standard (including casts and unions). This system introduces several techniques—both compile-time and runtime—to reduce the overhead of runtime checks, while still maintaining 100% memory safety. In this report the author describes his effort on making the system more reliable, more portable and applicable to the real-world applications, as well as the system's basic design and implementation methods.

## 1　はじめに

本稿では、完全なメモリ安全性を実現し、ANSI C 言語 (C90) の全仕様をサポートした処理系である Fail-Safe C の概要と、その実用化のための取り組みについて報告する。

Fail-Safe C は、ANSI C 言語の仕様で定められた全てのメモリ操作（キャストや共用体を含む）に対しその安全性を保証し、全ての危険なメモリアクセスを事前に検知し防止する。また、コンパイル時や実行時の様々な最適化手法を組み合わせることで、実行時検査のオーバーヘッドの削減を行っている。このコンパイラを用いることでプログラマは、既存のプログラムを大幅に書き換えたり別の言語に移植したりすることなく、そのままプログラムを安全に実行することができまる。このシステムは 2001 年より著者が東京大学において研究を始めたもので、2005 年より産業技術総合研究所で実用化の為の研究開発を進め、2008 年よりホームページ上で一般に実装を公開している。

Fail-Safe C を設計・実装する上で、特に実用プログラムをきちんと安全にし、実サーバの安全性向上に貢献するという目標を達成する為に、著者は次の 4 つの目標とその優先順位を設定した。

- 不正な実行につながる全てのメモリ操作違反を検出し、どんなプログラムに対してもメモリ上のデータ構造の破壊を起こさないようにする
- キャストを含む ANSI C (C90) の全機能を完全にサポートする
- (1)(2) の前提のもとで可能な範囲において、既存のプログラムで良く用いられる ANSI C を厳密に満たさない実装上の慣用もサポートする

- 実装の工夫等により、(1)〜(3) を満たす前提で可能な限り速度の向上をはかる

また、将来の実用化を考える上で、次のような点も意識した。

- 通常の C 言語で可能なモジュール単位の分割コンパイルを可能な限りサポートする。特にその際、分割コンパイルによって生じる型不整合などの問題もきちんと解決し、上記の安全性を複数モジュールのプログラムににおいても維持する。
- できるだけ既存の C 言語処理系と同じような手順でプログラムが処理できるようにする。そのために具体的には、プログラム全体の解析や、依存モジュール間でのコンパイラ処理の順番の制限など、研究上は問題にならなくても実用上問題になる要素についてはできる限り回避する。

本稿では、まず次節で Fail-Safe C の実現方法の概略について簡単に解説する。これらの技術の詳細については、[13, 11] や [12] なども参照されたい。続いて、これらの技術を実用プログラムに適用できるようにする為に行なった開発についても概略を述べた後、実装の現状や性能、将来の展望について述べる。

## 2 実現方法の概要

基本的な Fail-Safe C の設計は、可能な範囲で他の既存の安全な言語の実装を参考に、それらの手法で実現できない C 言語独特の機能をサポートするための拡張を加えるような形になっている。既存の安全な言語の例として、Java や C#、Lisp や ML などは既にメモリ安全性の保証された実装が存在する。これらの言語は、言語設計の最初の段階から安全な実装を作ることを念頭においていると思われる。

例えば、これらの言語ではいずれもポインタ（あるいは参照）と整数は基本的に独立な型になっていて、特に整数からポインタへの変換はこれらの言語のどの処理系でも（少なくとも言語間の連携 API などのための特殊機能を除いては）存在していない。また、これらのいずれの言語にも、ポインタ演算が存在しないことも共通しているほか、静的型のある言語についてはポインタ間のキャストについても厳しい制限が課せられている。これは偶然ではなく、メモリ安全性を保証する実装の仕組みと密接に関係している。通常安全な言語の実装では、1 つのメモリ上のオブジェクトに対する参照は常にそのデータ全体を指し示すように強制される。また、配列のような連続データの集合へのアクセスでは、そのオブジェクトにデータサイズなどのメタ情報を付随して記録しておき、オブジェクトのアクセスの際にアクセス範囲チェックや型の不整合の検知などを行なうことで、範囲外のメモリが破壊されることを防いでいる。このような実装において任意の整数から参照への変換を許してしまえばメモリ安全性を簡単に壊してしまうことができる他、ポインタ演算によりオブジェクトの中途への参照を許してしまうと、オブジェクトの先頭などに格納されているメタ情報へのアクセスが実装できず、アクセス範囲チェックなどができなくなってしまう。

一方で、C 言語においては規格上ポインタ演算やポインタ型間のかなり自由なキャストが許されている他、整数からポインタへのキャストも特定の状況では許されることが明示されているため、既存の言語の実装の方式そのままではこれらの機能を実現できないか、あるいはメモリ安全性を破壊してしまう結果になる。また、C 言語にはその他にも（キャストを含む）関数ポインタ、可変引数、型指定のない `malloc()` なども存在するため、これらの機構をサポートできるような実装方式の拡張が必要となる。

Fail-Safe C の具体的な実装ではこれらの問題を解決するために以下のような仕組みを新たに導入した。

**Fat Pointer:** ポインタ演算を許し、かつその場合でもポインタがどのオブジェクトを指示していたかを追跡し続けることのできるポインタ表現

**Fat Integer:** Fat Pointer と同じ情報を保持することのできる整数の表現

**オブジェクトを用いた型つきメモリブロック:** オブジェクト指向プログラミングの技術を用いてデータのサイズと型の双方を記録するメモリ上のデータ表現

**Virtual Offset:** Fat Pointer などの内部表現を隠蔽し、あらゆる型のデータに対して統一的なメモリの見え方を提供するような特殊なアドレス表現

**Access Method:** キャストされたポインタの利用をサポートするためにメモリブロックに付与された「メソッド」

**Cast Flag:** メモリアクセスの高速化のために Fat Pointer に導入する特殊なヒント情報

本節の残りの部分ではこれらの技術の概要を、解決すべき問題と解決手法を整理する形で説明する。

## 2.1 Fat Pointer

まず、C 言語プログラムを安全に走らせる上で真っ先に問題になるのが、ポインタ演算とアクセス境界検査を両立させることである。そのために Fail-Safe C ではいわゆる「fat pointer」ないし「smart pointer」と呼ばれる手法を導入している。全てのポインタの表現を 2 ワードに拡張し、そのうちの 1 ワードには常にメモリ上の領域の先頭のアドレス（ベースアドレス）を記録し、もう 1 ワードにそこからの「オフセット」を記録する。ポインタ演算があった際にはオフセットだけを変化させることで、C 言語のポインタ演算の semantics を保ったままポインタがメモリ上のどの領域を指し示しているかを追跡することができる。

C 言語では規格上任意のポインタを整数に変換しその値を調べることが許されている。Fail-Safe C ではその際の「実効値」としてはベースアドレスとオフセットの和が整数値として得られるように実装されている。

## 2.2 Fat Integer

C 言語においてはポインタから変換した整数値をそのまま変更せずに使う場合、その整数型がポインタよりサイズが小さくないに限り、もう一度その値を元と同じ型へのポインタに変換し直すことが許されている。これを実現するためには当然、整数もポインタと同じだけの情報を保持できる必要があることになる。そのため、Fail-Safe C では 1 ワード分以上の幅を持つ整数に対しては内部で 2 ワードの表現を用いこの情報を維持することにし、これを fat integer と呼んでいる。Fat integer の表現としては fat pointer と同様の表現を取ることも可能だが、その場合整数の演算の際に常に大きなオーバーヘッドが生じるため、実際にはベースと「実効値」を保持することにし、実効値とオフセットの変換はプログラム中で型変換が出現した箇所で毎回行なうことにした。整数とポインタの間の変換は整数演算より圧倒的に出現回数が少ないため、この方が全体の性能を向上させることができる。

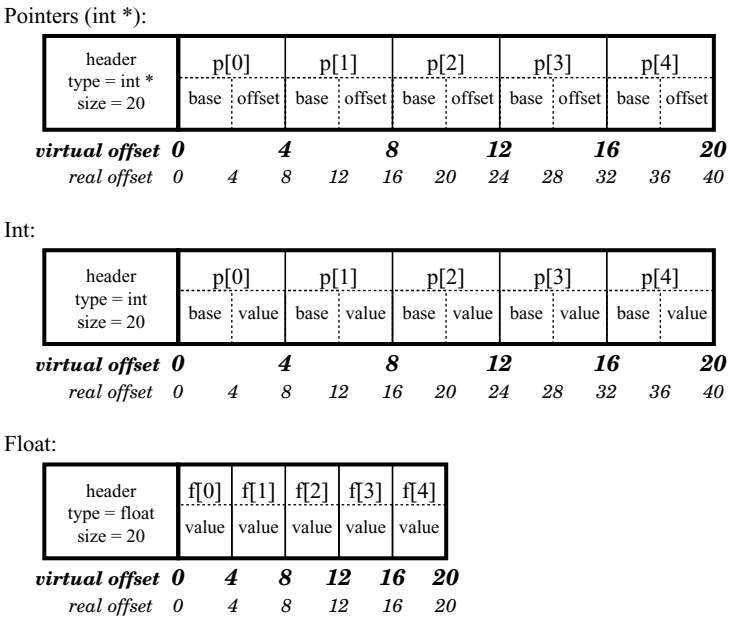



図 1: 基本的な型のメモリブロック上の表現

## 2.3 型つきメモリブロックと Virtual Offset

メモリ上に複数の種類のデータの配列が格納されている場合、データをアクセスする際には一般にオフセットの検査の他に取り出す値の種類がプログラムが期待するものになっていることも保証する必要がある。通常のキャストのない静的型つき言語では参照の型と指し示すデータの型は常に対応が取れているため後者は常に満たされていることを期待することができる。一方で、Fail-Safe C ではキャスト操作が許されているため、一般に後者の性質が成り立たず、実行時に確認を行なう必要がある。

また、C 言語ではしばしば型の整合しないメモリのデータを意図的に読みとる操作がしばしば行なわれる。例えば Unix (POSIX) のソケット API では、ネットワークのアドレス空間毎にそれぞれの空間内のアドレスを表す構造体型が定義されていて、これらの型と汎用の `sockaddr` 型の間でポインタのキャストを行なうことで、複数のアドレス空間のアドレスを統一的に指定できるようになっている。このようなプログラムを動作させるためには、キャストされたポインタの利用に対して何らかの合理的な動作をさせる必要がある。

この問題の解決のために Fail-Safe C ではまずメモリ上のデータ表現を拡張し、データの他にデータのサイズとデータの「型」を記録しておくことにした。これらの情報はグローバル変数などに対してはコンパイル時に、動的な変数やヒープ領域などに対しては領域の確保時に設定される。

更に、キャストされたポインタの利用をサポートするために、メモリ上のさまざまな型のデータ表現に対して統一的な「見え方」を定義するため、「virtual offset」の概念を導入した。これは端的には、メモリ上の（fat pointer など手が加えられた）実データではなく、プログラムからの見た目に対してアドレスを付与することを意味している。Virtual offset の付与されたメモリブロックの例を図 1 に示す。この表現を導入することのメリットは多く、実アドレスや実オフセットを用いた場合との関係では

- 通常のコンパイラと Fail-Safe C でコンパイルした場合でデータサイズ（`sizeof` で得られる値 = 配列での隣接要素のアドレス差）が完全に一致する
- fat integer や fat pointer などでデータ表現のサイズを変更した場合でも、プログラムから見える値の性質（例えば `unsigned int` が 32 ビット計算機で 0 から $2^{32}-1$ の値を取り得る）と、オフセット値から想定される性質（例えば `unsigned int` のサイズが 4 であること）が一致し、新たな混乱を招かない
- ポインタ演算や整数演算で操作できない「ベース」の部分の値が完全にアドレス表現から隠される (オフセットや実行値とサイズが一致するため)

などのメリットが、また要素番号を用いる場合と比較してもデータサイズが異なるポインタを変換した場合でも、変換の際にオフセットを修正する必要がないためキャストを簡単かつ一貫してに表現できる利点がある。但し、キャストとポインタ演算を組み合わせた場合に、型は正しいのに要素を正しく指さないポインタ（例えばオフセットが 1 の整数へのポインタ）が生じる可能性があることだけは注意しなければならない。

## 2.4 Access Method

このメモリ表現と virtual offset を用いれば、例えばポインタの配列に `char *` でアクセスした場合の動作などを、元のプログラムでどのように動作するかとの対応を考えることで「自然に」定義することができる。この例では、通常のコンパイラであればポインタの配列は当然に実効アドレスが並んでいるはずであり、文字の配列として読み出した場合であれば実効アドレス値の 2 進表現に対応したコードの文字が読み出されることになるから、メモリ呼び出しを「ポインタを実効値に変換してその値の一部を返す」ようなプログラムに変換してやれば、元のプログラムと同じ動作をすることになる。このような変換処理をこまめに実装してやれば、原理的には `sockaddr` のような形でキャストを安全に使っているプログラムは元と同じように動作することになる。

とはいえ、実際にはポインタはあらゆる型のメモリブロックを指し示すことができるため、実際にプログラムとしてそのまま実装することには無理がある。そのため、Fail-Safe C ではメモリブロックを更に拡張し、オブジェクト指向のメソッドテーブルの手法を借用することでこの問題を解決した。メモリ上の各ブロックに付与された型情報にメソッドテーブルを付加し、基本的なメモリ読み書き機能をメソッドとして提供する。具体的には読み・書きそれぞれに 1・2・4・8 バイトと、全部で 8 通りのメソッドをプログラムの文面中の型（基本型・ポインタ型・構造体）のそれぞれに実装する。基本的なメモリの読み書きは、アクセスしたいデータサイズに応じてこのメソッドを呼びだし、得られた値を自らが使いたい型にキャストすることで実現することができる。このようにメモリアクセスを 1 段間接化することで、安全性検査のための内部データ表現の変更とキャスト操作を両立させることができた。

## 2.5 Cast Flag

とはいえ、実際に実用プログラムを走らせる時に、メモリアクセス毎にメソッド呼出を伴うことは当然現実的でない。実際に簡単なテストプログラムで試した限りでも、access method を経由したメモリアクセスは通常の C 言語プログラムのメモリアクセスの 10 倍〜16 倍程度の時間がかかってしまう。通常のプログラムではキャストされたポインタを頻繁に用いることはあまりなく、圧倒

的に多くのメモリアクセスはキャストがなされていないポインタを用いる場合であることが想定されるため、このオーバーヘッドはキャストを許す代償としては大き過ぎると考えられる。

この問題の解決のために、Fail-Safe C では各ポインタにキャストの有無を記録し、キャストがされていない場合には access method をバイパスして範囲検査とデータの読み書きを直接行なう手法を導入した。幸い、C 言語はある意味では「きれいな」静的型つき言語であり、プログラム中にキャスト操作が現れない限りはポインタの型と参照先の領域の型は対応が保持される。また、Fail-Safe C の fat pointer の 2 ワードの表現のうち、ベースアドレスの下位数ビットは（文字型の領域を指している場合でも）使われていない。Fail-Safe C ではこのうちの 1 ビットを「cast flag」とし、キャスト操作ではこのビットを変更することで、あるポインタがキャストされているか否かを高速に判定できるようにした。これにより、メモリアクセスを行なう際に cast flag が立っていない場合は、virtual offset の変換とオフセットの範囲チェックだけで直接メモリアクセスを行なうことができ、大幅に速度を向上させることができた。更に、実際の実装では cast flag の表現に細工をすることで、cast flag の検査自体も省略している他、メモリブロック境界情報の関数内でのキャッシュなども用いてキャストがないポインタに対するメモリアクセス検査の高速化を計っている。

なお、virtual offset のところで述べたとおり、プログラム中にキャストがある際には要素を正しく指さないポインタが生じる可能性がある。そのため、キャスト操作においては型の一致の他にオフセットの整合性も検査し、オフセットが整合していない際には型が正しくても cast flag を立てることにしている。また、ポインタ演算でオフセットが桁溢れを起こした場合に、オブジェクトのサイズが 2 の累乗でない場合には同様の場合が生じるため、検査コードを自動的に挿入している。

# 3　実用プログラムの処理へ向けて

前節のような仕組みを実装することで、とりあえず ANSI C の単純なプログラムは通常通りにかつメモリ破壊を防ぎながら動作させることができた。しかし、これらの仕組みを実際のプログラムに適用し、現実のシステムに対応させるためには、その他にも色々な改良を行ない、またいくつかの追加ツールを用意することも必要になった。

## 3.1　分割コンパイルとリンカ

当然大きな C 言語プログラムを処理するためには、分割コンパイルの機能はほぼ必須ともいえる。Fail-Safe C はもともとプログラム全体を通じた解析はなくても安全性を実現できるような設計にしてあったこともあって、コンパイラ本体には大きな変更は必要とせずに分割コンパイルは実現できた。

ただ、プログラムに頻繁に現れる不完全な型の取り扱いには独特の困難があった。Java や Objective Caml などでは 1 つのあるモジュールが型定義等で依存する別のモジュールは先に（あるいは同時に）コンパイルされていることを仮定しているため、同じ構造体などの 2 つの定義がモジュール間で食い違うことは考慮する必要がなかったが、C 言語ではプロトタイプ等の宣言さえあれば依存関係に関係なく自由にコンパイル順を決められるため、モジュールのハッシュ値などを用いて不整合を検知することはできない。また、不完全型（内容の定義されない構造体型）へのポインタを含む構造体など「部分的に不完全な型」が存在するため、型の signature なども個々のモジュールの分割コンパイルの段階で決定することは不可能だった。

Fail-Safe C では、この問題を解く為にリンカで型の整合性の検査を明示的に実行することにして、そのための型検査器とリンカを実装した。この型検査器は不完全型の間の依存関係をモジュール単位できちんと追跡し、ある名前の型が複数のモジュールで非互換な定義がされていて、それらが混在する危険がある場合はリンクを拒否する一方、完全に独立して使われている場合にはエラーにならないようにすることで、C 言語に名前空間がないことによる名前の衝突が問題にならないようにした。また、この時に必要になる型情報は全て通常の `*.o` オブジェクトファイルに埋め込み、ライブラリを含めてリンカが必要な情報を取り出すことで、既存プログラムのビルト手順等に変更が起こらないように配慮している。このリンカのアルゴリズムや、とくに問題となるケース、autoconf など現実のプログラムで起こった問題とその対策などさらなる詳細については別の論文 [12] に記述されている。

## 3.2 標準ライブラリと移植性の確保

通常の C 言語プログラムはまず間違いなく標準ライブラリの関数を使用して記述されているといってよい。Fail-Safe C ではデータ構造の表現に大きく手を加えていることに加え、実プログラムにおいて標準ライブラリの内部で（不適切な引数などにより）起こるメモリ破壊が多いことから、標準ライブラリにおいても安全性をきちんと検査し不適切な引数を事前に排除する必要がある。

Fail-Safe C では実プログラムのサポートの為に、700 以上の標準ライブラリ関数に対して安全な実装を作成し提供している。これらはおおむね 3 年程度で、外部のプログラム作成企業と共同で整備を行なった。提供されている関数にはファイル I/O や文字列処理、メモリ処理やネットワーク・システム関係のもののほか、シグナルや setjmp/longjmp なども含まれており、おおむね旧来の BSD や System V で提供されていた関数はほぼ含まれている。これらの関数のうち一部は C 言語で記述し Fail-Safe C 自身で処理したものが含まれているが、実際にはほとんどのものは性能最適化や外部入出力の必要性などから通常の C 言語を基本に記述されている。特にシステムコールはファイル I/O 周りは、既存の libc と互換を取るように、それらの関数への wrapper のような形を取っている。

ただ、実装を進めてゆく過程において、処理系を移植する上では、アーキテクチャや OS 毎に構造体のサイズや定義が異なっているほか、型のサイズが異なれば Fail-Safe C 側での処理が大きく変わる為、移植性と記述性の確保をすることが大きな要求となった。このために我々は wrapper 記述の為の独自の簡易言語を設計し、アーキテクチャ間の相違をマクロと記述スタイルで吸収できるようにしたほか、コンパイラの作成時にシステムの各種パラメータを自動的に取り込む仕組みを整備した。実際にこの記述言語を用いて、x86, ARM, MIPS の 3 環境でソースの共有がなされている他、x86 の 32bit と 64bit 環境においてもほぼ全ての関数の記述を共用することに成功している。[1]

## 3.3 文法や処理の拡張

その他細かいところでも現実のプログラムに合わせるための改良や変更を随時行なった。

例えば、ANSI C90 の ISO/JIS 規格書や解説書などでは、完全な文法定義が LALR(1) を満たす BNF で記述されている。当初のコンパイラの実装では、これらの規格に合致した実装を作るために当然に規格と対応した文法定義を実装した。しかし、C 言語の文法仕様では

[1] 32bit と 64bit 環境では、int と void * のサイズの関係から int が fat integer になるかならないかの相違が生じ、int 型を多用している標準ライブラリ実装に極めて大きな相違が生じる。

1. typedef された型名と変数名は識別子を見ただけで区別がつくことを前提に定義されている
2. 一方で、文法定義の枠外で「型名で有り得ない場所に型名が現れたら識別子である」と規定されている
3. また、その性質を用いて明示的に型名を識別子で上書き宣言すると、そのスコープが有効である限りにおいてそのシンボルは型名ではなく識別子として扱われる

という性質が有り、これらは文法定義には反映されていなかった。

例えば、

```
typedef struct S S;
S *x;
struct S { ... };
```

という入力では、最初の行の最後のセミコロンにおいて S は型名となるが、3 行目ではルール (2) により型名としては解釈できないためこの S は識別子として解釈され正当な入力になる。また、ルール (3) は特にトリッキーで、

```
typedef int X, Y;
void f1(X X, int y), f2(X y, int z);
void f0(X X, int y) {
    static Z;
    unsigned Y;
    X = Y = y;
}
```

という入力では、

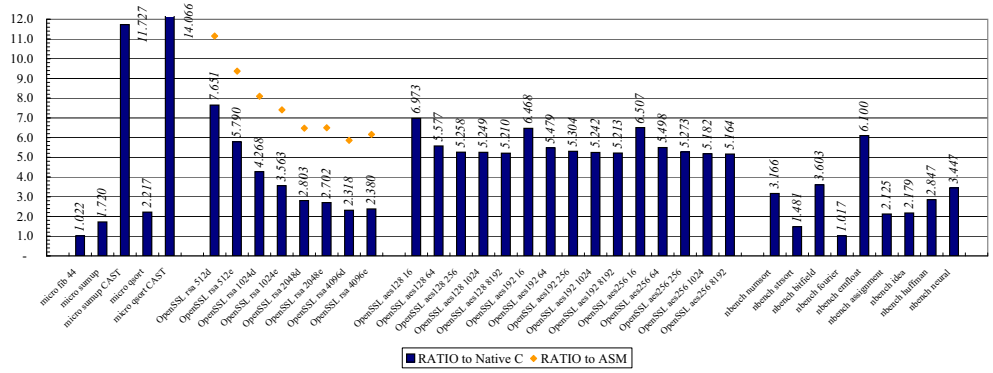
- 2 行目では、f1 の仮引数宣言の最初のコンマから括弧閉じまでの間で X は識別子になり、括弧を閉じた瞬間に X は型名として復活するので、f2 は正当な宣言（y は int 型）になる。
- 一方で、3 行目は関数宣言なので、一旦閉じ括弧で X は型名になるものの、開波括弧の直後で X は識別子として**再上書き**され、ブロック中では変数 X となる。更に、4 行目は型 Z の空の宣言、5 行目は unsigned int 型の変数 Y の宣言で Y が識別子に上書きされる。当然ブロック終了後は X も Y も型名となる。

結局、関数定義と変数宣言に関する文法定義は LALR(1) の元の記述ではマトモに記述できず、左から順に 1 シンボルずつ状態遷移する状態機械を手で 1 から作り、各箇所で型名として宣言されているものがどちらとして扱われるかに応じて文法を全部複製して対処する形で、元の規格上の定義とは対応づけのできない形で再実装することになった。

その他にも、通常のテストケースの実装では気づかないような些細な仕様に基づくバグや、C90 を逸脱しているプログラム (例えば // コメントを一部だけ使っているなど) などが多数見つかり、100 件以上の修正が行なわれ安定性は著しく向上した。

## 4 実プログラムへの適用

この Fail-Safe C を用いて、実際に以下のようなプログラムをコンパイルし実行が正しく行なわれることを確認した。




- テスト対象は ByteMARK benchmark と OpenSSL の speed コマンド。
- OpenSSL ではインラインアセンブリを無効でコンパイルしている。
- ◇ はアセンブリを有効にした OpenSSL 実装での参考値。

図 2: Performance of the Fail-Safe C compiler.

- OpenSSL version 0.9.8l
- OpenSSH version 5.3p1
- GnuPG version 1.4.10
- ISC BIND (Berkley Internet Name Domain) server version 9.4.2
- thttpd version 2.25b
- qmail version 1.03
- postfix version 2.5.5
- libtiff version 3.9.2 and libpng version 1.4.0
- zlib version 1.2.3, bzip2 version 1.0.5
- sed version 4.2.1
- pcre version 8.01
- libogg version 1.1.4, libvorbis version 1.2.3
- bash version 4.1

実際にこれらのプログラムを動かすのに必要なプログラムへの変更はごく些少で、かつそれらは主にビルドプロセス（コンパイラ名や autoconf など）の変更や、プログラム自体の型宣言などの誤りが多く、Fail-Safe C に特有の制限に起因するものはほとんどなかった。

実行性能の評価結果を図 2 に示す。プログラムにより性能オーバヘッドの差が大きいが、ネイティブの C プログラムと比較すると平均で 3〜4 倍程度（最悪で 7 倍程度、最善では 6 %程度）の実行時間となっている。

なお、左側の 5 つの棒グラフは cast flag と fat integer の性能を調査するためのマイクロベンチマークプログラムで、fat integer については実行速度にはほぼオーバーヘッドが生じていないことと、cast flag の有無で 5 倍以上の性能差を生じていることが分かる。

### 4.1 発見したバグなど

我々が実験の一部として前記のようなプログラムに Fail-Safe C に適用する過程で、プログラムの文面上の宣言等の誤りとは別に、いくつかの実際のメモリ操作の誤りを発見した。幸い (?) 発見したものの多くはテストプログラムなどのバグで致命的なものではなく、原作者に既に報告を行なった。

- thttpd がパス名の正規化検査ルーチンでメモリ上の配列の −1 番目にアクセスしてしまうケースがあり、その 1 バイトが '/' の場合、書き換えてしまう可能性もあった。
- BIND9 付属の DNS テスト用クライアントに、ネットワーク通信が成功しなかった時に限ってメモリ領域を事前に解放してからアクセスしてしまう誤りがあった。
- Pcre のビルド時のテストプログラムに、致命的でない範囲外読み出しアクセスがあった。
- Libvorbis のデコード処理中に、2 要素しか参照されないはずの係数テーブルの第 3 要素が呼び出し参照されているケースがあった。

セキュリティの研究という立場では、プログラムを入力として自動的にバグが発見できればプログラムの信頼性の向上に繋がるのだが、Fail-Safe C の場合はあくまで動かしてみた結果で不正メモリ操作があったかどうかが分かるだけなので、バグと同時に誤動作を引き起こす入力がないとバグが発見できないことが、上記でテストプログラムなどでの誤りを良く見つけてしまう原因の 1 つとなっている。今後、例えばプログラムの解析や動的エラー挿入、不正な入力の自動生成ツールなどのデバッグ系の技術と組み合わせることで、効率的なバグの発見に貢献できる可能性がある。

## 5 関連研究

Fail-Safe C と最も類似の研究としては、恐らく George Necula らによる CCured [9, 2] が挙げられる。CCured は「C 言語に対する健全な型システム」という位置付けになっていて、C 言語のプログラムを解析して通常のポインタ型にメモリの利用に関する複数種類のポインタ型のどれかを割り当てる型解析の形に整理されている。この解析の結果として、元のプログラムは大まかに「キャストを含む汚い部分」と「キャストを含まない奇麗な部分」の 2 つに分割され、前者の中のポインタは 2 ワードの fat pointer 表現と 1 ワード辺り 1 ビットのベースかオフセットかを表すフラグで、後者の中のポインタは通常のポインタか、3 ワードの fat pointer 表現のいずれかで表現される。Fail-Safe C と比較した時の彼らの研究の利点は、全てが奇麗な部分として型解析がうまくいった際にはより少ない検査で効率的にプログラムが実行できる点にあり、実際ベンチマークプログラムでは比較的良い性能を達成している。

一方で彼らの研究の欠点は静的解析に強く依存している為にプログラムの全体の解析が常に必要になり、分割コンパイル、特にライブラリなどの扱いに大きな問題がある点と、「汚い部分」に解析上の強い伝染性があり、プログラムで 1 箇所キャストを使ってしまうと極めて広範囲に影響が及んでしまう点がある。これは「汚い部分」のポインタから「奇麗な部分」のデータを（型情報が得られない為）指示できず、かつ静的な型付けの「奇麗な部分」から「汚い部分」を指すポインタは専用の解析型にならなければならない点にあり、その結果キャストされたポインタから辿れる全てのデータと、そのデータと同じポインタ変数で指され得る全てのデータ、またそのデータから辿れる全てのデータと、遷移的に解析結果が「汚い部分」になってしまうことがあるためである。特

に、システムライブラリなどで「奇麗な部分」としてあらかじめ準備されてしまっているルーチンには汚い部分のデータを受渡しできない為、プログラムを書き換えたり検査を省略して健全性を放棄するか、コンパイルを諦めるかの選択を迫られてしまう場合もあり、プログラムサイズのスケーラビリティという点で問題が大きかったようである。

Fail-Safe C ではポインタが（CCured の意味で）「汚い」かどうかは cast flag で動的に管理されており、同じ変数に cast flag のあるポインタとないポインタが共存できる上、キャストで汚れたポインタの指し先であってもメモリの型情報から「奇麗な」ポインタの型情報が得られるため、ポインタの汚染が伝播せずスケーラビリティの問題が生じない。また、キャストの有無が ABI 上の差異を生まず、全てのプログラムにおいてキャストの存在を仮定してコンパイルされる（そうしてもプログラムサイズは若干大きくなるが速度上のペナルティーがほとんどない）為、ライブラリなどの分割コンパイルに関しても問題を生じない。ただ、静的解析自体のメリットは大きいものもあるので、今後彼らの解析手法などの部分的な導入を検討する価値はあると思われる。

その他の関連研究としては、部分的なメモリ安全性に関しては完全ではないものの実用的なツールとして StackGuard [3] や ProPolice [4] などが既に実コンパイラに導入されている他、最近ではより完全に近いものとして SoftBound [8] などもある。但し、完全な C 言語を変更なく取り扱えるものに限ると、関数ポインタやメモリ領域の不正解放などまで含め完全な安全性を理論的に保証している研究は多くない。また、C 言語プログラムにあらかじめ変更や付加情報を与えておいて、安全性検査を効率的に実現するものとしては例えば Cyclone [5, 6] や Deputy [1] などがある。

Fail-Safe C の拡張の可能性としては、C++ などのオブジェクト指向言語に適用する為に必要なシステムの設計への変更について以前に検討した [10] ことがあり、現時点では言語仕様の大きさなど理論上以外の問題がネックになりサポートは行なわれていないものの将来的に C++ サポートに繋がる可能性がある。また、Fail-Safe C のポインタ表現を Java 上で表現し、Java 仮想マシン上で C 言語のプログラムをメモリ安全に動作させる研究 [7] がある。Java 仮想マシン自体がメモリ安全性を保証しているため、この変換の存在がある意味で Fail-Safe C の変換の安全性を示している他、Java プログラムとの相互呼出などに将来応用できる可能性がある。また標準ライブラリの wrapper の作成については、関数の静的な性質を記述し自動的に wrapper を生成することを実現した研究 [14] がある。

## 6　まとめと今後の研究の方向性

本稿では Fail-Safe C の概略と実システムへの適用の実験の過程で現れた問題などについて簡単に報告した。現時点で既に Fail-Safe C は、ある程度プログラムが「正しい」必要はあるものの古典的な Unix のサーバプログラムであればある程度確実に動作させられるようなレベルの実装が完成しており既に実装も公開しているが、実際にこの技術を社会に広め実用システムの安全性に貢献する為には、今後更に研究と普及の努力を重ねる必要があると考えている。

現在我々は、現状の wrapper 記述言語や関連研究の成果を元に、より簡易に既存ライブラリの wrapper を記述したり、逆に通常の C プログラムからライブラリの一部を Fail-Safe C でコンパイルしたものを呼び出せるようにするなど、より既存プログラムと柔軟に Fail-Safe C を組み合わせることのできる仕組みを研究開発している。これらの開発を通じてシステムの実用性を高め、実システムへの展開の努力を続けてゆくつもりである。

# 参考文献


[1] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *ESOP 2007*, 2007.

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

[3] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.

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

[5] 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.

[6] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

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

[8] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[9] 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.

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

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

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

[13] Yutaka OIWA. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 259–269, New York, NY, USA, 2009. ACM.

[14] 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.