安全なシステム記述言語および高信頼 OS 記述言語

代表者　東京大学大学院情報理工学系研究科コンピュータ科学専攻 米澤 明憲
http://www.yl.is.s.u-tokyo.ac.jp/e-society/

1. はじめに

本研究開発の目的は，型理論をベースとした「静的解析技術」[1]の先進的な発展に寄与すると同時に，その応用によって，基盤ソフトウェアシステムを実用レベルで高安全・高信頼にしようとすることである.本プロジェクトの成果は，別編の「概要編」にもあるように次のようなものである.

1. メモリ安全な C 言語コンパイラの開発
2. 高安全 C 言語コンパイラの開発
3. OS を記述可能な型付きアセンブリ言語の設計・実装
4. システムソフトウェアの正しさの数学・形式的検証

近年，C 言語で書かれたプログラムの低い安全性と信頼性の薄さが大きな問題になっている.Web サーバの乗っ取りやコンピュータウィルスの拡散の大半は，C 言語プログラムに含まれるプログラミングミスが原因である．バッファオーバフロー攻撃と呼ばれる攻撃によって乗っ取りや機能停止を引き起こされてしまう欠陥が，システム中の C 言語で記述された部分から極めて多く発見されている.

一方，Java 言語などのメモリ操作の安全性を意識して設計された言語で書かれたプログラムに対してバッファオーバフロー攻撃を行うことは困難である．しかし，C 言語には膨大なソフトウェア資産があり，実行が高速であるという利点があるため，C 言語は現在も広く使用され，今後も使用され続けることが予想されている．C 言語プログラムの安全性を高めるための技術は今までにいくつか提案されているが，多くは場当たり的な方策である．このような状況の中，プログラム言語の研究成果を利用してバッファオーバフローなどの攻撃を防止することが強く求められており，緊急の課題となっている.

（1）我々は小規模ながら，従来より型システムを利用してメモリの操作の安全性を保証する C 言語処理系の研究開発を行ってきていた．そこで，本プロジェクトにおいて，平成 15 年度からこのアプローチを大規模に展開させることにした．このアプローチでは，C 言語プログラムを機械語コードにコンパイルする段階で，ソースコードの精緻な解析で得られる情報をもとに，メモリ操作の安全性をプログラムの実行時に検査するコードを自動的に挿入する．その結果，プログラムは安全性を検査されながら実行され，バッファオーバフロー攻撃等を検知・防止することができるようになる.

本プロジェクトの成果を応用すると，ビジネス・社会に大きなインパクトを与えることができる．我々は，安全な C 言語処理系（コンパイラ）を開発したが，これは ANSI 標準の C 言語仕様を満たす全てのプログラムがコンパイルできるので，これを普及させることにより，社会基盤として使われているが安全面での欠陥を持つプログラムを大幅に減少することができる．特に，サーバの乗っ取りやコンピュータウィルスなどの被害を減少させることに大きく貢献できる．さらに，我々の研究開発の成果により，バグを早期に発見することができ，システム開発の生産性を向上できるため，安全性・信頼性が高いソフトウェアシステムを構築することが容易になる.

（2）近年の機密情報漏洩事件の頻発により機密情報に対する社会的な関心が高まっている.我々のプロジェクトにおける高安全な C 言語コンパイラの開発では，機密情報を保護する情報流解析の手法を用いた VITC (Vulnerability and

[1] プログラムのソースコードをコンパイル時あるいはそれ以前に解析する技術.

Intrusion Tolerant Compilation technology) というコンパイル手法を着想し，その定式化および実装を行った．ここでは，前述のメモリ安全 C コンパイラを拡張して情報流解析を行う静的・動的型システムを実装し，基本的な C の機能のほとんどに対して情報流の解析を行える段階にまで達した．

（3）OS 記述用の型付きアセンブリ言語の設計・実装では，メモリとスレッド管理機能を記述可能な強く型付けされたアセンブリ言語 TALK を実装，これを用いて実際に簡単な OS カーネル (TOS) を作成し，期待通りの動作を確認することができた．さらに，TALK を発展させ，割り込み機構と共有メモリを扱うことのできる型システム拡張，iTALK を定式化した．割り込みは，型付きアセンブリ言語では扱うことが出来なかったが，OS が周辺ハードウェア等と連携する際に必須となる重要な機能であり，これを型システムの枠組みで対処できたのは大きな前進である．これを用いて実際に簡単な OS カーネル (TOS) を作成し，期待通りの動作を確認することができた．

（4）システムの安全性を保証するために，形式的論理に基づく言語で仕様を記述し，検証ツールを用いて数学的・形式的に検証することは有効な方法として知られている．しかし，基盤システムの多くは並列分散系で低レベルシステムであるため，仕様を記述することが困難であり，状態数の爆発などの問題で現実的なシステムの検証は大変難しいものであった．

我々のプロジェクトでは，まず高次論理系に基づく定理証明支援系 Coq を用いて，メールシステムの核となるプロトコルの実装の正しさを証明した．さらに，Coq の枠組で「分離論理」体系を定式化し，ヒープメモリやポインタへの操作の安全性を証明する体系を構築した．そして，この体系と Spin モデル検査器を用いて，教育用として実用されている OS である Topsy の，

(a)メモリ管理モジュールの安全性，

(b)メッセージ通信サービスの正しさ，

(c)カーネルメモリ領域の保護に関する性質，

などを検証することに成功した．また，この検証の過程で Topsy 内のこれまで知られていなかったバグも発見した．

2. メモリ安全 C 言語コンパイラ Fail-Safe C の開発

2.1. はじめに

本プロジェクトにおいて我々は，Lisp 系言語と同等レベルのメモリ安全性と ANSI C 規格への準拠を両立させた C 言語処理系である Fail-Safe C [1]を開発した．C 言語で書かれた既存のプログラムをそのまま安全に実行させ，バッファ・オーバーフローなどのメモリ破壊バグによるサーバのユーザ権限の乗っ取りなどを防ぐことを目標としている．この処理系は C 言語のキャスト操作を，プログラムのメモリ安全性を壊さない形で完全にサポートしている．また，キャスト操作の安全性と非キャストポインタのメモリ操作の効率を両立させるために，キャストの有無を示すフラグをポインタに埋め込み，かつその表現を工夫することでフラグのチェックによる効率の低下を防いでいる．

2.2. Fail-Safe C におけるポインタ操作の実装

Fail-Safe C は C 言語のキャスト操作を完全にサポートしている．特に，ポインタの型と参照されるデータの型が一致しない場合にも安全性を保ちつつメモリ操作を許すために，プログラム中で用いられる全てのデータブロックに「アクセスメソッド」と呼ばれる汎用の読み書きインタフェースを付加し，型の異なるブロックの間でのデータ表現の違いを吸収している．一方，プログラム中で圧倒的に多く出現するキャストされないポインタによるメモリ操作では，直接メモリ領域内部をアクセスすることにして，実行効率の著しい低下を防いでいる．この 2 つの異なる特性を持つ動作を実行時に柔軟に切り替えるために，Fail-Safe C はポインタに「キャストフラグ」という 1 ビットのフラグを付加し，キャストの有無をポインタ自身に記憶させてい

る．本節では簡単にその具体的な実装を説明する．

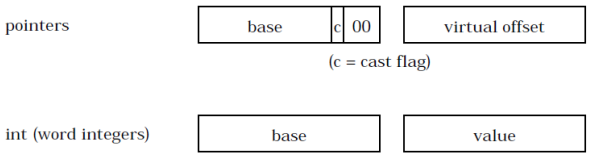

図 1: Fail-Safe C 内部の値の表現

2.2.1. 安全なポインタとキャストフラグ

Fail-Safe C では，キャストを含むプログラムにおいてもポインタを用いたメモリ操作を安全に行なうために，全てのポインタと，ポインタのサイズと等しい整数の値を内部的に 2 ワードで表現している(図 1)．ポインタにおいては，そのうちの 1 ワードは常にメモリブロック（動的または静的に確保されたもの）の先頭の物理アドレス(base address) を保持し，もう 1 ワードはポインタの指し示す値の，ブロック先頭からのオフセットを保持している．このオフセットは virtual offset と呼ばれ，実際のデータ表現における物理アドレスの差ではなく，実行中のプログラムから見た値のサイズに基づく値を用いている．例えば，ポインタの配列中の添字 1 の要素を表す virtual offset は，2 ワード表現における物理のアドレスの差 8 ではなく[2]，32 ビットの大きさに対応する 4 である．このような値を採用することで，ポインタの値について既存の C 言語処理系との互換性を向上させ，またポインタのキャストが行なわれた際も同じ virtual offset の値を読み書きすることで，通常の処理系と同様の動作を模倣することができる．

さらに，base address の取り得る値を常に 8 (2 ワード) の倍数に制限することで，base を保持するワードの下位 3 ビットを余らせ，このうちの最上位ビットにキャストフラグ(cast flag) と呼ばれる 1 ビットの論理値を格納する．

ポインタに格納されるキャストフラグは，参照先の各メモリブロックの表現型との関係において，次の条件を満たすように設定される．

1. 静的型 T * のポインタが，T 型以外の表現の値を格納しているメモリブロックを指す場合，そのポインタのキャストフラグは 1 でなければならない．
2. 静的型 T * のポインタのオフセットが，型 T のサイズ(virtual offset に基づくもの) の倍数でない場合，そのポインタのキャストフラグは 1 でなければならない．
3. ポインタが null ポインタ(無効なポインタを表す base = 0 の特別な値) である場合を含め，上記のどちらでもない場合，そのポインタのキャストフラグは任意である．通常，このような「正しい」ポインタはキャストフラグが 0 であることが望ましい．

言い替えると，ポインタのキャストフラグが 0 であることは，このポインタがブロックの有効なサイズの内側を指しているならば，ポインタの静的型から想定されるデータの 1 つの先頭を指し示していることを示している．Fail-Safe C では，この事実を用いて，キャストフラグが 0 であるようなポインタを用いた場合は，範囲チェックのみを行ない直接ブロック内のデータをアクセスする．一方，キャストフラグが 1 である場合は，キャスト操作によってポインタ型の健全性が壊れており，ポインタの指し示すデータが想定と異なるデータ表現でメモリに格納されている可能性を想定して，次節で説明する，遅いがデータ表現に依存しない方法でメモリアクセスを行なう．

なお，C 言語におけるポインタの加減算においては，暗黙のうちに加算数の整数が要素サイズ倍されてから，被加算数のポインタに加えられることに注意されたい．このため，整数溢れが起こる場合を除いて，ポインタの加減算の前後で性質 2. が(もちろん 1. も) 変化することは

[2] この 8 は 32 ビットアーキテクチャ上での値で，64 ビットアーキテクチャでは 16 になる．説明の都合上，これ以降 32 ビットアーキテクチャにおける具体的な値を用いることにする．

ない．そのため，キャストフラグの再計算はキャスト操作の時と，2 の累乗でないサイズのポインタにおいてオフセットの整数溢れを検出した場合にのみ行なえばいい．

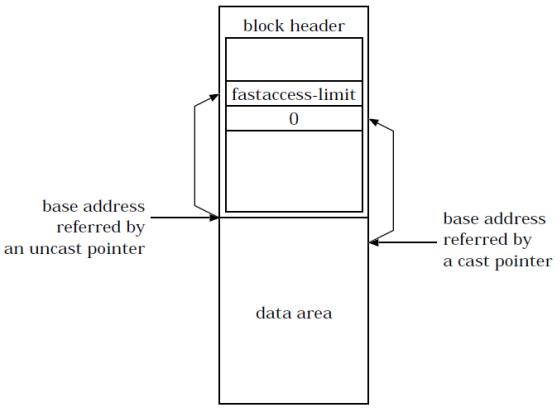

図 2: キャストフラグ検査の効率化

2.2.2. ブロックとアクセスメソッド

Fail-Safe C では，プログラム中の全てのメモリブロックにヘッダ領域を付加し，そのブロックの virtual offset に基づくサイズと，そのブロックが格納する値の表現型を表すデータを格納している．この表現型情報には,「アクセスメソッド表」と呼ばれる表が含まれていて，ちょうど通常の C++ 処理系におけるオブジェクトの実装と同じように，表現型によらず抽象化・共通化されたインタフェースを提供する．アクセスメソッド表に含まれる「アクセスメソッド」は, ブロックの base address と対象となる virtual offset を受け取り，そのブロック内のデータを読みとって（あるいは値を書き込んで）返すという基本的なメモリアクセス処理を実装している．キャストされたポインタがメモリ読み書きに使われた際，Fail-Safe C は対象ブロックのメソッドを呼び出すことで，どんな型のブロックに対しても，常に統一された処理でメモリ読み書きを行なうことができるようにしている．データ型の違いはアクセスメソッドの内部で吸収されるため，この処理によって型の不整合を起こすことがないことが保証される．

2.2.3. キャストフラグ検査の最適化

上記のような実装では，ポインタを用いてメモリをアクセスする際には，(1) ヌルポインタのチェック，(2) キャストフラグのチェック，(3) 配列範囲のチェックの 3 つの検査を行なう必要がある．このうち，(1) と(3) は Java や ML のような他の安全な言語においても必須であるが，(2) は Fail-Safe C 特有なものであり，実験によれば約 10 %程度のオーバーヘッドになっている．このオーバーヘッドを削減するため，Fail-Safe の実装では次のような手法で検査の最適化を行なっている．各メモリブロックに付加されたヘッダの中には，メモリアクセス可能なオフセットの上界を表す値(図 2 中の fastaccess-limit)が格納されているが，ブロックヘッダのサイズを 1 ワード増やし，上界の値の直後のワードに常にダミー値 0 を格納しておく．このようなメモリ構造と前節のキャストフラグの表現の組合せで，メモリアクセスの際にキャストフラグを除去せずに(3)の範囲チェックをいきなり行うと，キャストフラグが 0 のポインタについては正しくアクセス上界の値を読み取ることができるが，キャストフラグが 1 のポインタにおいては，キャストフラグの表現が 1 ワード分のアドレスの差に相当するため，ダミー値 0 を代わりに読みとることになり，範囲チェックが常に失敗する．また，ポインタがヌルである場合, ベースの値は 0 か 4 であるので，範囲チェックはメモリ最下位の物理メモリの存在しない領域を参照し，確実にページ例外シグナルが送出される．そのため，この範囲チェックが成功した場合，(1)(2)(3)の全てのチェックが成功したことを意味し，直接メモリアクセスを行なっても安全であることが判明する. また，キャストフラグが 0 であることも保証されるので, base の値はそのまま物理アドレスとして用いることができる．一方で，範囲チェックが失敗した場合は(2) または(3) のどちらかで失敗したことになるので，改めて範囲チェックを（今度はキャストフラグを除去してから）行い判別する．

この場合でもメモリの近接した領域を操作す

るため，オーバーヘッドはほとんど無いことが確かめられている．これにより，Fail-Safe C はキャストが行われないプログラム部分について，既存の ML や Java 言語とほぼ同等の比較的効率の良いメモリ操作を行うことが可能になっている．

2.3.コンパイラの実装と公開

Fail-Safe C の実装は，[2]で公開中である．

この実装によって, OpenSSL (ネットワーク通信の暗号化や通信相手の認証等を行うプログラム，ソースコード約 300,000 行)， BIND9 (インターネット上での名前解決に広く用いられているサーバ，ソースコード約 350,000 行)，などの基盤的システムをコンパイル・実行した．

参考文献

[1] 大岩 寛, 安全な ANSI C コンパイラの実装手法. 博士論文, 東京大学大学院情報理工学系研究科コンピュータ科学専攻, 2004 年度.

[2] Fail-Safe C project[3]: https://staff.aist.go.jp/y.oiwa/FailSafeC/

3. 高安全 C 言語コンパイラ VITC の開発

3.1. 機密情報漏洩と，言語面からの対策

インターネットを通じた機密情報漏洩事件と，それに伴う情報被害が多発しているが，現状でこの問題を解決することは非常に難しい．なぜなら，一般的に OS やアプリケーション開発において，情報の機密性を指定し，その漏洩を防止するためにはその機密情報管理コードを開発者自身が書く必要があり，また，そのコード自身の正しさを検証することも困難だからである．この問題を解決するため，プログラミング言語自体に情報の機密度を表現できる機構を導入し，プログラム内の情報の流れをコンパイラが解析し，機密漏洩が起きないことを保証する，情報流解析による機密漏洩防止策が提案されてきた．情報の機密性とその伝搬を型システムで表現し，その型システム上で型付けできるプログラムは機密漏洩を起こさないことを数理的に保証すれば，アプリケーション開発者自身は機密情報管理コードを書く必要がなくなり，管理コードのバグによる機密漏洩という問題も防ぐことができる．

C 言語はその実行速度やハードウェアよりの記述能力から，Java などの新言語登場後の現在もシステム設計では最も一般的であり，多くのアプリケーションにも広く使われているが，それらの多くに情報漏洩の事例や危険性がある．そのため，情報流解析による機密漏洩防止策が取られるべきであるが，現在まで，情報流解析の研究や実装は，Java や ML など，理論的に厳正な静的型システムを与えることができた言語系に対してしか行われてこなかった．C 言語に対してはメモリ安全性の欠如をはじめとする種々の困難があると思われていたためか，これまで深く考えられてこなかったのである．しかし，今や C 言語の最大の問題であったメモリ安全性の欠如は我々が本プロジェクトで行ったメモリ安全 C コンパイラ研究の成果を利用すれば解決できる．我々の高安全 C コンパイラ，VITC の開発研究では，このメモリ安全な C 言語のコンパイルを仮定した上で，C プログラムに対して機密漏洩防止を保証する情報流解析による高度な安全性を与えることを目標とした．

3.2. 平成 17 年度からの成果

VITC コンパイラの研究開発は平成 17 年度より，メモリ安全 C コンパイラの研究成果を踏まえて開始された. 平成 17 年度に基礎的な理論研究を行い, プロトタイプを開発, 実験を行った. 平成 18 年度には, プロトタイプから得られた知見を元に, VITC の本実装を開始した. 現在 VITC は，基礎となるメモリ安全 C コンパイラの実装として，産業技術総合研究所で開発が行われている Fail-Safe C[4]を採用し，それを拡張することで実装されている．

平成 18 年度半ばから，最終的な目標である，既存の実用 C アプリケーションの VITC による

[3] 産業技術総合研究所情報セキュリティ研究センターが公開

コンパイル実験を開始した.この過程において，研究開発初期に想定していた理論に基づいたコンパイル法では問題のあるプログラムパターンが発見された．この問題の解決を通して，フロー依存型解析[2]やポインタ解析などの既知の技術の適用，実装と，ポインタに関する情報流の型解析[9]や，リソース識別子に対する情報流解析に関する新しい理論的発展[10]が得られた.これらの成果により，C 言語で書かれた現実的なアプリケーションである thttpd ウェブサーバを VITC によりコンパイルし，誤ってサーバ上の機密情報を外部に漏らさないことを確認した.

VITC でコンパイルされたプログラムは，情報流解析による安全性を備えており，誤って機密情報を漏洩しないことが保証される．また，この安全性は，通常動作中だけでなく，メモリ脆弱性を突いた攻撃を受けた後にも保証される.その結果，VITC ではメモリ脆弱性攻撃に対して非常に強い耐性を持つプログラムを生成することができる．(図 1)

3.3.VITC の基礎アイデア

3.3.1. メモリ安全 C コンパイラ

C 言語プログラムでは，言語仕様自体に起因するメモリ管理の不備による，バッファー・オーバーフローなどのセキュリティーホールが度々指摘されている．この問題に対して，言語研究の分野からはメモリ安全コンパイラの提案が行われてきた(例えば[4]):これらのコンパイラでは各メモリアクセスが適正である場合にのみアクセスを許可しないよう言語仕様を拡充し，実際にコンパイラはメモリアクセスにチェックコードを埋め込んだ実行プログラムを生成する.不正なメモリアクセスがあった場合は実行を中断することで，メモリエラーによる誤動作，そしてそこから生ずる可能性のある機密漏洩を防止する(図 1). これらのコンパイラにより，多くの既存Cアプリケーションが変更無しにメモリ安全なプログラムにコンパイルできる．エラー忘却型計算[1]では，このアイデアをさらに押し進め，不正メモリアクセスを検知した後にもなんとか実行を継続させる．不正アクセスをそのまま実行するのは危険であるため，例えば，不正書き込みは単純に無視するなどの回避を行う.この方式は一見無謀に見えるが，多くのプログラムを案外「それとなく」実行できることが報告されている[1]．ウェブサーバなど，連続動作を期待され，かつ個々のセッションがある程度独立しているため，あるセッションでのエラー忘却型計算が他に影響を与えないと仮定できるソフトウェアなどに有望な技術である.ただし，その継続された実行が安全な物であるかは議論されていない:エラー忘却型計算は攻撃者のプログラム注入などは防ぐことができるが，その適当な実行継続はプログラマが想定しているプログラムの意味と大きくかけ離れ，誤って機密情報を攻撃者に開示してしまう危険性がある.

3.3.2. 情報流解析

情報流解析[5]では主に型システムによって情報の流れを表現し，追跡することでプログラムの機密漏洩を起こす可能性を判定する．この情報流解析を実用的な言語に適用した例では Java や ML の拡張が知られている[3,6]．それに対して，C 言語で書かれたプログラムの多くで情報漏洩が発生し，大きな問題になっているにもかかわらず，我々の知る限り C 言語での情報流解析の例は無いようである．その理由は，C 言語本来の仕様では，そのメモリ管理は不完全で，かなりの部分がプログラマの責任に任されており，不正なメモリアクセスが起こりうる状況において情報流を追跡することはほぼ不可能だからである．また，この分野の全体の傾向として，既存のソフトウェアシステムに対して情報流解析を行い，安価に高安全なプログラムを生成する研究はあまり行われていない．既存ソフトウェアは，そもそも情報機密度の概念がない言語で，情報流を意識せず書かれている．そのため，各情報の機密度の仕様を与えたとしても，そのままでは情報流安全なプログラムにコンパイルすることは不可能であることが多い.既存システムに対して情報流解析による安全性

を安価に提供するのであれば，コンパイラ・システムはこの点も考える必要があるだろう: 既存ソフトウェアを情報流解析の対象とするのであれば，解析を行うために必要となる対象ソースコードの変更はできるだけ少なくすむよう，情報流型システムを設計するべきである．

3.3.3. VITC の主張

メモリ安全 C コンパイラをはじめとする，高安全 C コンパイラのそもそもの動機は C ソフトウェアシステムのセキュリティー問題, つまり，機密漏洩問題を解決することにある．C 言語ではメモリ管理が不十分であるため，不正メモリアクセスが攻撃の主要手段であり，メモリ安全コンパイラはこれを不可能とすることで，間接的に機密漏洩問題の一部を解決している．しかし，メモリ安全といえども，機密を漏洩してしまうバグを含むプログラムは存在する. よって，さらなる高安全性を C プログラムに求めるならば，そもそもの動機である機密漏洩自体を防止できる，情報流解析を行う高安全コンパイラが求められるはずである．逆に，情報流解析の視点から見れば，C 言語での情報流解析には，プログラムのメモリ安全性が保証されていて，情報流の追跡が可能となることがまず必要である. ただし, 単純なメモリ安全性では不十分である:

```
printf("hello ");
if(h){ array[x] = 10; }
printf("world");
```

上の例では h は高機密な変数であり，その値は外部に漏れてはならない．もし h が真であった場合は配列アクセスが行われるが，もし添字 x が不正である場合，単純なメモリ安全性[4]ではプログラム実行が中断されてしまう．プログラムの実行中断は多くの場合(この場合, 二つ目の printf が実行されるかどうかで)外部から観測可能であり，中断された場合はその事から h が真であったことが推測できるため，h の情報が漏洩してしまう．プログラムを中断させないためには何らかのエラー処理が必要であるが，C プログラムではこのような安全性が静的にはまず検証できないメモリアクセスが普遍的に存在する．情報流解析を行うために，その全てに対し一つ一つプログラマがエラー処理を書くとすれば，それは恐ろしい労力となるだろう．これを自動的に行うとすれば，必然的にエラー忘却計算が必要となる．エラー忘却計算の「勝手な」エラーを処理のため，プログラムはエラーが起きた場合，プログラマの意図とは異なった奇妙な動きをするかもしれない．しかし，そのような状況でも，情報流解析による安全性により，機密情報が攻撃者に漏れる事は防ぐことができる．その結果，VITC ではメモリ脆弱性攻撃に対して非常に強い耐性を持つプログラムを生成することができる(図 1)．以上のように，VITC では C のメモリ安全コンパイラ技術を基礎として，情報流解析とエラー忘却計算を互いに補完させることによって，既存の C 言語プログラムを情報流安全なプログラムへとコンパイルする(図 2).

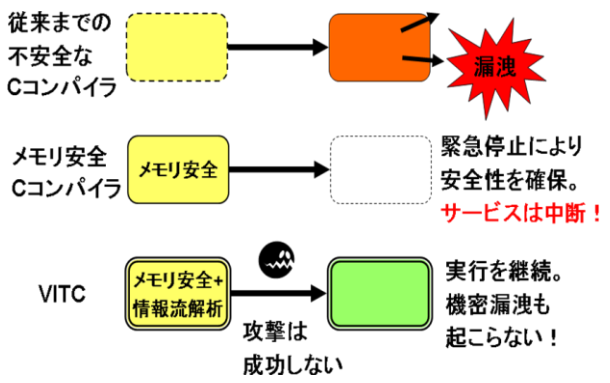

図 1: 各コンパイラと安全性

3.4.VITC の型システム

メモリ安全性を仮定した C は，かなり宣言的ではあるものの関数型言語に酷似した言語である．そのため，VITC の情報流型システムは基本的に ML のそれ[6]を踏襲する．各機密度は束を形成し，C の型に付加される．例えば int^H は高(High)機密度の整数，int^H $*^L$ は高機密度整数を指す, 値自体は低機密(Low)なアドレスの型である．束の半順序によって型のサブタイプ関係

が定義され，関数は情報流に関する多相型を持ち，HM(X)[7]で説明されている制約型システムおよび推論を適用できる．

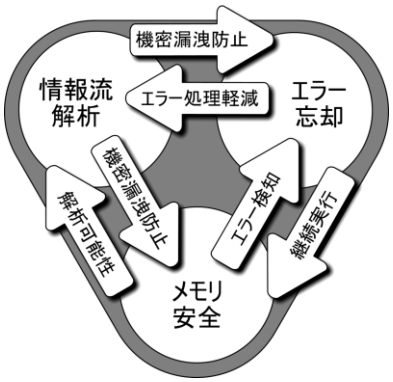

図 2: VITC の基本技術の相互関係

3.4.1. 動的型

VITC では，C 言語の既存アプリケーションが情報流を意識して書かれておらず，そのままでは静的に情報流型安全なプログラムとして静的に型検査できない場合に備えて，動的型を導入している[8]:

```
bool b;
int^H h;
int^L l;
int* p; p = b ? &h : &l;
```

例えば上のソースコードを情報流解析すると，最終行において，高機密情報 H，低機密情報 L を指すポインタ型を単一化しようとして型検査に失敗する．このような場合，動的型を使ってエラーを回避する:

```
p = b ? (int^dyn *)&h : (int^dyn *)&l;
*(int^L *)p
```

ここではキャストを使って，p への代入に際して型 H と L を動的型 dyn を使って隠蔽し，型エラーをひとまず解消する．後に，変数 p を使ったポインタアクセスの際に，指示先が L の型を持つかどうか実行時型検査を行なっている．指示先が H の場合はこの検査は失敗，エラー忘却計算により機密漏洩を起こさない適当な値が読み出される．

3.4.2. 型キャストの問題

VITC での型理論でもっとも問題となるのは，型キャスト周辺での情報流型の扱いである:

```
int* p;
int x = (int)p;
int* q = (int*)x;
```

ポインタ p は型推論により，int^{α} $*^{\beta}$ という型を持つ．α, β は型変数である．我々の平成 18 年度までの型システム[8]では x は p の持つアドレス値が代入されるため $\mathrm{int}^{\beta'}$ where $\beta \leqq \beta'$ という型を持つが，ポインタ値として指し示す中身の機密度 α のことは x の型には伝わらず，忘れられてしまう．そのため，この値をふたたびキャストにより int *へと戻した場合，指示先の機密度が不明であるため，q = (int^H *)x という具体的な機密度 H を使った明示的な実行時検査が必要となる．また，この実行時検査時には x をポインタとして見たときの指示先の機密度も具体的に知る必要がある．つまり，変数 p の指示先の機密度も，int^H * p などと指定する必要がある．つまり，プログラマは上記コードを例えば以下のように修正しなければコンパイルできなかった:

```
int^H * p;
int x = (int)p;
int* q = (int^H *)x;
```

この例では文面から明らかに(int*)x は p と同じ値であり，型 int^H * を持つはずで，実行時検査を行う必要もないはずだが，この型システムではそれを検知できない．ただ，このような特異な例は実際のアプリケーションではあまり出てこないと予測されたため，特段の対策は取られてこなかった．この予想に反して，この初期の型システムを使った実際のアプリケーション例の型検査の実験では，実はキャストの有無にか

かわらず，大量に明示的な機密度指定が必要になることが判明した．つまり，ソースプログラムの修正が必要となる箇所が予想以上に多かったのである:

```
void initialize_S(struct S *);
void f(){
  struct S s;
  initialize_S(&s);
}
```

関数 f は構造体 S をスタックに取り，そのアドレスを初期化関数 initialize_S に渡す. 類似コードは C では普遍的に見られるが，この変数 s の機密度は明示的に, 例えば struct S^H s などと一々指定しなければならない．なぜなら，関数 initialize_S 中で実行時検査がこの機密度 H に対して行われる可能性があるからである. 例えば:

```
void initialize_S(struct S *p) {
  int x = (int)p;
  ...
  struct S *q = (struct S^L *)x;
  ...
```

この実行時検査では，指定された機密度 L をローカル変数 s の機密度と比較する．そのため，先ほどの例の p と同じく s は具体的な機密度を用いて宣言されなければならない．一般的に，関数引数が関数内で実行時検査の対象にならなければ, アドレスが関数に渡される変数に対し，struct S^H s のような明示的機密度の指定は必要ない．しかし，引数が実行時検査されうるかどうかはこの型システムでは判別できない．結果として，関数からアドレスが脱出しうる全てのローカル変数で明示的に機密度を指定してやる必要があった.

3.4.3. 改良された型システム

実験から得られた初期の型システムの問題は, 次のようにまとめられる:

- キャスト周りの不必要な実行時検査
- 実行時検査の必要/不必要性が関数を越えて伝わらない

この問題を解決するため，平成 19 年度，我々は型システムを改良し，キャストが影響を与えるのは C 本来の型の部分のみで，情報流型は影響を受けないように改良された．前節はじめの例の x の型は, この型システムでは$\cdot^H$ int α where L $\leqq \alpha$ となって, H は失われない. そのため，最後のキャストにおいても q の型は $\cdot^H$ int β where $\alpha \leqq \beta$ と自動的に推論できる．C 言語ではどのような整数もポインタにキャストでき，それを通したメモリアクセスが(不正であるかもしれないが)可能であることを考えると，より一般的には，この新しい型システムでの情報流型 τ は，C 本来の型の部分(int など)を省略すると，基本機密度型 λ (機密度，機密度変数 α，そして動的型 dyn(後述))の無限リストと見なすことができる:

$$
\begin{aligned}
\lambda &::= l \mid \alpha \mid \mathrm{dyn} \\
\tau &::= \tau * \lambda \mid a \mid \lambda^{\omega} \mid \sigma * \lambda \\
\sigma &::= \forall a_1,\ldots a_m\, \alpha_1,\ldots,\alpha_n.\, \tau \rightarrow^{\lambda} \tau \\
K &::= k, \ldots, k \\
k &::= l \leqq l \mid \alpha
\end{aligned}
$$

無限リストを導入するための有限長の記法として，任意の τ へと具体化可能な型変数 a，一定の機密度を持つ無限リスト...* λ *...* λ を表す λ^{ω} が用意されている．このアイデアは初期の型システム[8]で問題となっていたキャストによる明示的動的検査の必要性を大幅に減らすことができるだけでなく，一定の制限下ではあるが，ループを持つデータ構造の情報流解析を実行時検査せずに静的に解析できる．これは既存の C アプリケーションを安価に安全化するために必要不可欠である.

3.5.実装と実験

最新の VITC の実装では，特にポインタ周辺の新しい型付けによって，比較的少量のプログ

ラム変更でソフトウェアの情報流を解析できることができるようになった．また，Unix のファイルシステムのファイルの権限情報を利用し，そのファイル内容の機密度を自動的に得ることにより，プログラムコード内だけの情報機密度だけではなく，ファイル上に存在する高機密データに対しても，機密漏洩を起こさないことを保証する機構を導入した．我々は thttpd ウェブサーバ(C ソースコード一万行，平成 20 年一月現在，世界で 2 万以上稼働中)に対して VITC による安全化を行った．まず，thttpd にバッファー・オーバーフロー脆弱性を付け加え，通常の C コンパイラでコンパイルしたこの改変サーバはメモリ脆弱性攻撃によりサーバ乗っ取りや，機密情報漏洩が可能となることを確認した．次に，VITC を使ってこの改変サーバをコンパイル，起動し，同じ攻撃を試みた．VITC 版サーバは，メモリ安全性によりサーバ乗っ取りが不可能であったばかりでなく，誤動作による機密漏洩も防御しながら，実行の継続が可能であることを確認できた．

3.6.結論

VITC コンパイラではメモリ安全 C コンパイルと情報流解析を組み合わせることで，安価に既存Cアプリケーションを安全化する手法を提案している．情報流解析されたエラー忘却計算により，VITC でコンパイルされたプログラムはメモリ脆弱性攻撃を受けた後も，プログラム中の機密情報を漏洩することなく安全な継続動作を行うことができる．

参考文献

[1] Martin Rinard et al. Enhancing server availability and security through failure-oblivious computing. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation San Francisco, CA, December 2004.

[2] Jun Furuse, Dzung Dinh-Khac, Viet Ha Nguyen. Flow Sensitive Information Flow Analysis for C Programs. In Proceedings of Japan-Vietnam Workshop on Software Engineering, 2007.

[3] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Symposium on Principles of Programming Languages, 1999.

[4] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure (progress report). LNCS 2609, Feb 2003.

[5] A. Sabelfeld and A. Myers. Language-based information-flow security. In IEEE Journal on Selected Areas in Communications, 21(1), 2003.

[6] V. Simonet. Flow Caml in a nutshell. In Proceedings of the first APPSEM-II workshop, pages 152-165, March 2003.

[7] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4), 1997.

[8] 古瀬 淳. VITC: 対攻撃耐性コード生成コンパイラ. 第 4 回ディペンダブルソフトウェアワークショップ (DSW'06-2). 2006.

[9] 古瀬 淳, 米澤 明憲. 安全なシステム記述言語および高信頼 OS 記述言語～ VITC: 情報流解析による 高安全 C コンパイラ. 情報処理学会誌『学と産の連携による基盤ソフトウエアの先進的開発』（掲載予定）. 2008.

[10] Ryozo Yamashita. Information Flow Analysis for Resources.修士論文,東京大学大学院情報理工学系研究科コンピュータ科学専攻, 2007 年度.

4. OS 用型付きアセンブリ言語の設計・実装

4.1. メモリ管理機構と型付けされた言語

近年，静的プログラム解析技術，特に型理論の発展により，多くのアプリケーションプログラムが型付けされたプログラミング言語（例: Java，C#，O'Caml など）を用いて作成されるようになった．これは，型付けされた言語で記述されたプログラムのメモリ安全性（バッファオーバーフローなどの脆弱性がないこと）を，型検査によって保証・検証できるためである．ちなみに，オープンソース系ソフト

ウェアの開発レポジトリである SourceForge（http://sourceforge.net）によれば，実に 6 割近いソフトウェアが，型付けされた安全な言語を用いて構築されている．例えば，Java などの安全なプログラミング言語を用いて構築されたウェブサーバやメールサーバが実際に存在する．

ところが OS は未だに，型付けされていない言語（例: C 言語，アセンブリ言語）を用いて作成されている．このため従来の OS は実行時にメモリエラーを生じるかもしれず，OS の安全性・信頼性を確保・検証することは困難であった．OS はコンピュータシステムを運用する上で最も基盤的なソフトウェアであり，その安全性・信頼性はコンピュータシステム全体の安全性・信頼性に大きく影響する．例えば，現実に広く用いられている OS である Windows や Linux は，ほぼ全て C 言語やアセンブリ言語で記述されているため，これらの OS の安全性や信頼性は事実上ほとんど検証されていない．その結果，コンピュータシステムの異常停止や外部の攻撃者による乗っ取りなど，現実に大きな問題が生じている．

このような安全性・信頼性の問題にもかかわらず，OS が型付けされた安全な言語を用いて作成されてこなかった理由の一つは，OS の重要な機能である「メモリ管理機構」（メモリ領域を確保したり解放したりするための機構）を，既存の型付けされた言語では記述できなかったという点にある．これは，既存の型付けされた言語は，外部のメモリ管理機構（例: ガーベジコレクション）に依存しているため，プログラム作成者に明示的にメモリ管理機構を記述させると，メモリ安全性が保証できなくなってしまうためである．OS はコンピュータシステムの資源（メモリや CPU）を管理するプログラムであるので，外部のメモリ管理機構に依存することは現実的ではない．

そこで本研究は，メモリ管理機構を記述することが可能な程度に柔軟かつ強力な，型付けされたプログラミング言語の実現方式について考察・検討を行った．その結果，複数のポインタが同一のメモリ領域を指しているかどうかを型システムで明示的に扱うこと，また可変長配列（プログラム実行時までサイズが分からない配列）を型システムで直接扱うことで，既存の多くのメモリ管理機構を記述できることが分かった[1]．例えば，最も基礎的なメモリ管理機構であるメモリスタックや，より複雑な機構である，いわゆる malloc/free のような機構も記述可能となることが分かった．

この検討結果にもとづき，我々は新たな型付きアセンブリ言語 TALK[4]を，設計・実装した．また，この TALK を用いて実際に OS カーネル のプロトタイプ（TOS[5]）も実装した．TOS は非常に簡素な OS カーネルであるが，メモリ管理機構やマルチスレッド管理機構などの重要な機構が実装されており，またハードウェアを操作するための簡単なデバイスドライバも実装されているため，理論上は，現実の OS カーネルにかなり近いと言える．起動直後からほとんど全ての部分が型付けされたプログラムで記述された OS は（機能が大きく制限されているとはいえ）おそらく TOS がはじめてである．また，TALK を更に発展させ，割り込み機構を扱うことのできる型システム拡張 iTALK を設計した．なお，これら TALK と TOS の実装（ソースコード）は，プロジェクトのホームページで公開中である．

以下では主に TALK の詳細について説明する．

4.2. TALK（OS 用型付きアセンブリ言語）

図 1 に TALK の抽象機械の構文を，図 2 に TALK の型の構文を示す（実際の TALK はより複雑であるが，ここでは大幅に簡略化したものを示している．詳細は[2][3]を参照されたい）．図 2 における配列型 a により，前節で述べた可変長配列を表すことができる．より具体的には，型 t[i]は各要素が型 t を持つような配列で，サイズが i のものを表す．図 2 にあるように，TALK の型システムでは，サイズを表す整数型 i に型変数を取ることができるため，可変長配列（型検査時にはサイズが不明である配列）を表すことができる．例えば，型 int[α]は，整数値の配列で，そのサイズが型検査時には不明（α）であるものを表す．

図 1 において，split 命令と concat 命令を除いた命令は，通常のアセンブリ言語の命令と意味は同じである．ld 命令はメモリから値を読み込む命令である．

st 命令はメモリに値を書き込む命令である．jmp 命令はいわゆる分岐命令である．また，実際の TALK には他にも条件分岐命令や整数演算命令などが存在するがここでは説明を省略する．

$$
\begin{array}{lrcl}
(register) & r & ::= & r_1 \mid r_2 \mid r_3 \mid r_4 \\
(operand) & o & ::= & r \mid [r] \\
(instruction) & i & ::= & ld\ r = [r] \mid st\ [r] = r \\
 & & & \mid split\ o, o \mid concat\ o, o \\
 & & & \mid jmp\ o \mid \ldots \\
(instructions) & I & ::= & \cdot \mid i\ ; I \\
(memory) & M & ::= & \cdot \mid \{c \mapsto c\} M \\
(registers) & R & ::= & \{r_1 \mapsto c; \ldots, r_4 \mapsto c;\} \\
(state) & S & ::= & (M, R, c)
\end{array}
$$

図 1: TALK の抽象機械の構文

$$
\begin{array}{lrcl}
(type\ var) & \alpha & & \\
(type\ vars) & \Delta & ::= & \cdot \mid \alpha, \Delta \\
(int.\ type) & i & ::= & \alpha \mid c \mid \ldots \\
(word\ type) & w & ::= & i \mid \forall \Delta. C. \Phi. \Gamma \\
(tuple\ type) & t & ::= & \langle w, \ldots, w \rangle \\
(array\ type) & a & ::= & t[i] \\
(memory\ type) & \Phi & ::= & \cdot \mid \{i \mapsto a\}\ \Phi \\
(regs.\ type) & \Gamma & ::= & \{r_1 \mapsto w; \ldots, r_4 \mapsto w;\} \\
(cop) & cop & ::= & < \mid \leq \mid \ldots \\
(int.\ cstrts.) & C & ::= & \cdot \mid i\ cop\ i, C
\end{array}
$$

図 2: TALK の型の構文

なお，上述のように表現された可変長配列にアクセスするためには，整数間の制約も型システムで把握する必要がある．例えば，上述の可変長配列型 int[α]を考える．この配列の 3 番目の要素に安全にアクセスするためには, この配列のサイズが 3 以上, すなわち $\alpha \geqq 3$ でなければならない．このような整数制約を扱うため，TALK の型システムでは，図 2 の C で表されるように，整数制約を明示的に追跡する．なおメモリを表す型（Φ）の形から分かるように，全てのメモリは基本的に可変長配列として扱われるため，型システムで整数制約を追跡することは必要不可欠といえる．

また，可変長配列を用いてメモリ管理を実装するためには，配列を分割したり結合したりする必要がある（後述）．このため TALK には，一つの配列を二つの配列に分割する split 命令と，逆に二つの配列を一つの配列に結合する concat 命令が存在する．これらの命令をプログラム中に適宜記述することにより，柔軟なメモリ管理を実現することができる．なお，これらの命令（split と concat）は，純粋に型情報のみを扱う擬似命令であり実行時には実行されないため，多くの split 命令や concat 命令がプログラム中に存在しても実行時のオーバーヘッドは存在しない．

更に，メモリ安全なメモリ管理機構を実現するためには，複数のポインタが同一のメモリ領域を指しているかどうかを追跡する必要がある．例えば，あるプログラムが，あるメモリ領域にポインタが格納されていると思っているのに，他のプログラムがそのメモリ領域を誤って整数として扱ってしまうと，前者のプログラムのメモリ安全性が失われてしまう．

この問題を解決するために TALK の型システムは, メモリの状態を表す型（図 2 のΦ）において，各要素が重複しないようにすることで，同一のメモリ領域を指すポインタが常に同一の型を持つようにしている．TALK においては，いわゆる C 言語等で見られるような表現のポインタ型は存在せず，整数とポインタは双方とも整数型として表される. このため, ある整数型 i をもつレジスタ r をポインタとして使えるかどうかは，メモリ型がアドレス i に可変長配列の存在を示しているかどうかによる．

図 3 は split 命令の型付け規則である（これも実際にはより複雑であるが，ここでは簡略化している）．簡単にいえば，この型付け規則は，まず指定されたオペランド（o_1）が実際に配列を指すポインタであるかどうかを検査し，次いでその配列のサイズが，指定されたサイズ（o_2）よりも大きいか，もしくは等しいかどうか検査する．最後に実際に配列の型を二つに分割し，次の命令の型付けに移る．

$$
\frac{\begin{array}{c}
\Delta, C \models i_1 = get\_type(\Phi, \Gamma, o_1) \\
i_2 \equiv get\_type(\Phi, \Gamma, o_2) \\
\Delta \vdash \Phi = \{i_1 \mapsto t\,[i_3]\}\Phi' \\
\Delta, C \models i_2 \leq i_3 \\
\Phi'' \equiv \{i_1 \mapsto t\,[i_2]\} \\
\{i_1 + i_2 * sizeof\,(t) \mapsto t\,[i_3 - i_2]\}\Phi' \\
\Delta, C, \Phi'', \Gamma \vdash I
\end{array}}{\Delta, C, \Phi, \Gamma \vdash split\ o_1, o_2;\ I}\ (\textsc{split})
$$

$$
\begin{array}{l}
where \\
get\_type(\Phi, \Gamma, o) \quad = \quad \Gamma(r) \quad (if\ o\ is\ r) \\
\qquad\qquad\qquad\qquad\quad\ \ \Phi(\Gamma(r)) \quad (if\ o\ is\ [r])
\end{array}
$$

図 3: split 命令の型付け規則

$$\frac{\begin{array}{c}\Delta, C \models i_1 = get\_type(\Phi, \Gamma, o_1) \\ \Delta, C \models i_2 = get\_type(\Phi, \Gamma, o_2) \\ \Delta \vdash \Phi = \{i_1 \mapsto t\,[i_3]\}\{i_2 \mapsto t\,[i_4]\}\Phi' \\ \Delta, C \models i_1 + i_3 * sizeof(t) = i_2 \\ \Phi'' \equiv \{i_1 \mapsto t\,[i_3 + i_4]\}\Phi' \\ \Delta, C, \Phi'', \Gamma \vdash I\end{array}}{\Delta, C, \Phi, \Gamma \vdash concat\ o_1, o_2;\ I}\ (\textsc{concat})$$

図 4: concat 命令の型付け規則

図 4 は concat 命令の型付け規則である（split 命令同様, ここでは簡略化している）. この型付け規則は, 簡単にいえば, split 命令の型付け規則の逆である. まず指定された 2 つのオペランド（o_1 と o_2）が実際に同じ要素型を持つ配列を指すポインタであるかどうかを検査し, 次いでそれら 2 つの配列がメモリ上で連続しているかどうかを検査する. 最後に実際に配列の型を一つに連結し, 次の命令の型付けに移る.

$$\frac{\begin{array}{c}\Delta, C \models i_1 + i_2 = \Gamma(r_2) \\ \Delta, C \vdash \Phi = \{i_1 \mapsto t\,[1]\}\Phi' \\ \Delta, C \vdash t = \langle \ldots, w_{i_2}, \ldots \rangle \\ \Delta, C, \Phi, \Gamma\{r_1 \mapsto w_{i_2}\} \vdash I\end{array}}{\Delta, C, \Phi, \Gamma \vdash ld\ r_1 = [r_2];\ I}\ (\textsc{load})$$

図 5: ld 命令の型付け規則

図 5 は ld 命令の型付け規則である. この型付け規則は, まずレジスタ r_2 が長さ 1 の配列の唯一の要素（タプル）を指すポインタであるかどうかを検査する. そしてレジスタ r_1 の型をレジスタ r_2 が指す要素の型（図では w_{i2} で更新して, 次の命令の型付けに移る. この型付け規則から分かるように, TALK におけるメモリ読み込みは, 長さ 1 の配列に対してのみ可能である. このため, メモリ読み込みを行うためには, split 命令を用いて長さ 1 の配列を生成する必要がある.

$$\frac{\begin{array}{c}\Delta, C \models i_1 + i_2 = \Gamma(r_1) \\ \Delta, C \vdash \Phi = \{i_1 \mapsto t\,[1]\}\Phi' \\ \Delta, C \vdash t = \langle w_1, \ldots, w_{i_2}, \ldots, w_n \rangle \\ \Delta, C \vdash t' = \langle w_1, \ldots, \Gamma(r_2), \ldots, w_n \rangle \\ \Delta, C \vdash \Phi'' = \{i_1 \mapsto t'\,[1]\}\Phi' \\ \Delta, C, \Phi'', \Gamma \vdash I\end{array}}{\Delta, C, \Phi, \Gamma \vdash st\ [r_1] = r_2;\ I}\ (\textsc{store})$$

図 6: st 命令の型付け規則

図 6 は st 命令の型付け規則である. この型付け規則は, まずレジスタ r_1 が長さ 1 の配列の唯一の要素（タプル）を指すポインタであるかどうかを検査する. そしてレジスタ r_1 が指す要素の型を, レジスタ r_2 の型で更新して, 次の命令の型付けに移る. ld 命令と同様に, TALK におけるメモリ書き込みは, 長さ 1 の配列に対してのみ可能である.

TALK においてメモリ読み込み・書き込みを長さ 1 の配列に制限している理由は, TALK の型システムが, メモリの型を破壊的に更新できるようにしているためである（これは strong update と呼ばれる）. 例えば, TALK の型システムでは, 要素が全て整数値 0 であるような長さ 10 の配列の型は, 0[10]と表せるが, 直接この配列の 5 番目の要素を整数値 1 で更新（strong update）できるようにすると, 更新後の配列の型を自然に表現することが困難になってしまう. そこで, TALK の型システムでは, 直接配列の要素を更新できるのは長さ 1 の配列に制限するかわりに split 命令と concat 命令を導入した. 例えば上述の例では, まず split 命令で配列型 0[10]を 0[4], 0[1], 0[5]の三つの連続する配列型に分割し, 真ん中の長さ 1 の配列に対して更新を行う. この結果, 更新後の配列の型は, 0[4], 1[1], 0[5]という三つの配列型で表現される. なおこの制限のため, メモリ読み込み・書き込みを行うときには, プログラム中に適宜 split 命令と concat 命令を挿入しなければならないが, 前述の通りこれらの命令は型検査時にのみ使われ実行時には実行されないため, 性能には影響しない.

4.3. TALK を用いたメモリ管理の実現

TALK では, システム上の利用可能なメモリは, 可変長配列の集合として表現される. これは, 利用可能なメモリのサイズをシステム起動前に OS が知ることが一般に不可能であることを考えれば, 自然な表現である. その他のメモリの表現方法として, リスト(cons セル)等を用いた方法も考えられるが, そもそも実際の物理メモリが配列で扱われることを考えると, TALK のように可変長配列でメモリを表現する方がより自然であるといえる. 例えば, 物理メモリのデータの読み書きは基本的に定数オーダー（物理メモリのサイズによらない計算量）で実行されるが, これをリストで表現すると, 最悪の場合, 物理メモリ全てにアクセスする必要が生じる. これ

に対し，配列で表現すれば，物理メモリと同様に定数オーダーでメモリ操作を行うことができる．

実際のメモリ管理は，メモリを表す可変長配列を操作することで行われる．例えば，最も基礎的なメモリ管理機構であるメモリスタックは次のように実現できる．まずスタックそのものは，スタックが利用するメモリ領域を表す一つの配列と，スタックのどの位置まで既に利用されているかを表すポインタ（スタックポインタ）の組で表現できる．

このスタックからのメモリ領域の確保（いわゆるスタックへのプッシュ）は次の様に実現できる．まず，確保するメモリ領域のサイズ分だけスタックポインタをシフトする．これはTALKの整数演算命令によって実現される（前述の通り，TALKにおいては，整数値とポインタは型システムにおいては基本的に区別されない）．次にsplit命令を用いて新たに確保されるメモリ領域を残りのメモリ領域から分離する．スタックからメモリ領域を確保できるかどうかの検査は，配列の境界検査として実現される．なお，TALKの型システムは整数制約を明示的に追跡しており，配列の境界検査コードの正しさを検証できるため，結果としてスタックオーバーフローなどの問題を未然に防ぐことができる．

一方，メモリ領域の解放（いわゆるスタックからのポップ）は次のように実現できる．まず，解放されるメモリ領域のサイズ分だけスタックポインタをシフトする．これはスタックへのプッシュと同様，通常の整数演算命令によって実現される．次にconcat命令を用いて解放されたメモリ領域を，スタックが用いているメモリ領域に結合する．このconcat命令の型検査において，解放されるメモリ領域がとスタックの残りのメモリ領域とが連続しているかどうかが検査されるため，確保されたのと逆の順でメモリ領域が解放されていくことを保証できる．

また，いわゆるmalloc/freeのような，より複雑なメモリ管理は以下のように実現できる．まず，利用可能なメモリ領域（フリー領域）を，可変長配列の集合，例えばリストとして表す．また，このリストにはそれぞれの配列の長さを表す整数値も含めておく．

このフリー領域からのメモリ確保（malloc）は次のように実現できる．まず，フリー領域を表すリストを走査し，新たに確保されるメモリ領域のサイズよりも大きいサイズの配列を探す．各配列の長さはリストに保存されているので，この探索は通常の条件分岐命令（と整数演算命令）で実現できる．そして，見つけた配列をsplit命令によって分割し，新たに確保するメモリ領域をフリー領域から分離する．一方，メモリ領域の解放（free）は，解放するメモリ領域（配列）をフリー領域のリストに連結することで実現できる．

なお，上述のメモリスタックやmalloc/freeの仕組みによって確保されたメモリが，他のプログラム等から不正にアクセスされないことは，TALKの型検査によってプログラムの実行前に検証できる．

4.4. 実装

上述のTALKの設計にもとづき，実際にTALKのアセンブラと型検査器をIA-32用に実装した．TALKのアセンブラは，TALKで書かれたソースコードを入力として，型情報が埋め込まれたバイナリプログラムを出力する．一方，TALKの型検査器は，アセンブラによって出力されたバイナリプログラムを入力として受け取り，TALKの型システムの型付け規則に従って，そのプログラムの安全性（メモリ安全性と制御フロー安全性）を検証する．型検査の実装は，TALKの型付け規則をプログラムに変換するだけであるので，基本的に簡潔である．ただし，TALKの型システムは整数制約を明示的に追跡するので，この整数制約を解消するための制約解消器が必要であった．このために我々は，Omega testと呼ばれる良く知られたアルゴリズムをもとに，実際に制約解消器を一から実装した．ちなみにTALKアセンブラ・型検査器のソースコードはO'Caml言語で記述されており，双方合わせて約18000行弱である．

なお，実際のTALKアセンブラの入力対象は，TALKをIA-32アーキテクチャに対応させたものであるため，本報告書で説明したTALKとは文法・命令・仕様などが異なるが，理論的には大きな差はない．また，現在のTALKアセンブラは，IA-32の全ての命令･機能をサポートしているわけではないが，

OS の記述に必要となる命令・機能は全てサポートしている．

これに加え，TALK アセンブラを用いて実際に簡単な OS カーネル（TOS）を実装した．TOS は，OS 起動の最初期の部分を除いて，全て TALK で記述されている（TALK 以外で記述されているのは，IA-32 アーキテクチャ特有の特殊な CPU 操作が必要な起動時の処理のみである）．TOS は約 3000 行強の TALK ソースコードからなる．現状の TOS は非常に簡素ではあるが，メモリ管理機構（メモリの確保と解放)やマルチスレッド管理機構(スレッドの生成，切替，同期，終了)，ディスプレイとキーボードのデバイスドライバ等，OS の本質的な機能を提供しており，型付けされた言語で OS カーネルを作成できることを実践的に示している．メモリ管理機構は前節で説明したように実現されている．また，マルチスレッド管理機構は，スレッドをプログラムカウンタとメモリスタックの組で表すことにより，メモリ管理機構の問題に帰着することで実現されている．なお，デバイスドライバは，基本的にハードウェアとメモリの間でデータの転送を行うだけであるので，単なるメモリの読み書きとして実現できた．なお，TOS のバイナリプログラム全体の型検査に要する時間は，Pentium 4（3GHz）上で約 0.1 秒であった．これは，TALK の型検査による安全性の検査が現実的であることを示している．

これら TALK と TOS の実装は，[4][5]で公開中である．

参考文献

[1] 前田 俊行, 米澤 明憲. 強く型付けされたオペレーティングシステム. 日本ソフトウェア科学会第 22 回大会論文集. 2005.

[2] Toshiyuki Maeda and Akinori Yonezawa. Writing Practical Memory Management Code with a Strictly Typed Assembly .Language. In proceedings of the 3rd workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'06), 2006.

[3] Toshiyuki Maeda. Writing an Operating System with a Strictly Typed Assembly Language. 博士論文, 東京大学大学院情報理工学系研究科コンピュータ科学専攻, 2005 年度.

[4] TALK project: http://web.yl.is.s.u-tokyo.ac.jp/~tosh/talk

[5] TOS project: http://web.yl.is.s.u-tokyo.ac.jp/~tosh/tos

5. システムソフトウェアの形式的検証

5.1. 初めに

近年，これまで人間が行っていた重大な仕事の多くがコンピュータシステムに任されるようになってきている．そのような重要なシステムの多くは並行・分散または低レベルシステムであり，そのようなシステムの誤りにより悲惨な事態が起こる可能性を防ぐことが重要な課題となっている．

既存のシステムの安全性を保証するために，形式的論理に基づく仕様記述言語で仕様を書いた上で検証ツールを用いて形式的検証を行うことは有効な方法として知られている．しかし，記述言語の表現力によっては仕様を書くことが困難であることがあり，状態爆発などの問題で現実的なシステムの検証ができないことがある．

そこで，本研究では，高信頼システムソフトウェアの実現を目標としてモデル検査器と定理証明器によるシステムソフトウェアの形式的検証の研究を行った．

5.2. 形式的検証のための仕様記述言語

5.2.1. 状態爆発を抑える

並行プログラムが多くの状態を持ち，状態空間が爆発することは形式的検証の大きな困難として知られている．仕様記述言語の設計の仕方によって，検証のために探索すべき状態空間を削減する定理を得ることができる．

本研究では，空間論理と時相論理を扱う記述言語を設計し，"partial order reduction"という定理を証明した[1]．この記述言語を用いれば，場合によっては状態空間爆発という問題を抑えることができる．

5.2.2. 物理的なシステムを扱う

実世界とそれを対象とするコンピュータシステム

のための仕様を書くことは既存の時相論理では困難である．その理由の一つは，物理学における時間の無限仮定によって，物理的なシステムは離散的にモデル化しがたいことにある．

本研究では，既存の時相論理に序数の概念を導入した[2]．これにより，ロボットのようなコンピュータシステムの仕様記述と検証の負担を軽減できる．

5.2.3. 低レベルソフトウェアの仕様

低レベルのソフトウェアは細粒度のコンピュータメモリ操作を要するため仕様が書きにくいことがある．この問題を考慮して，よく知られている Hoare 論理を拡張して分離論理("separation logic")と呼ばれる論理が欧州で開発された．この記述言語に基づく形式的検証を目指して，オペレーティングシステムの形式的な検証を検討した[3].

5.3. モデル検査器と定理証明器によるシステムソフトウェアの形式的検証について

本研究では，以上の仕様記述言語を利用して，システムソフトウェアの検証をモデル検査器と定理証明支援系を用いて行った．

5.3.1. 分散アプリケーションを扱う

最初に，定理証明支援系を用いた現実的な並行プログラムの検証を可能にするためのアプローチを提案し，Coq 定理証明支援系を用いて評価を行った[4][5][6]．このアプローチでは，並行プログラムを直接定理証明支援系上にモデル化し検証するため 5.2.1.で提案された空間と時相論理を扱う記述言語を含む Coq ライブラリを設計し，実装を行った．このライブラリを用いて，現実的なアプリケーションとして Java 言語で実装された SMTP サーバの受信部分のプロトコル(図 1)の実装の正しさの検証実験を行った．

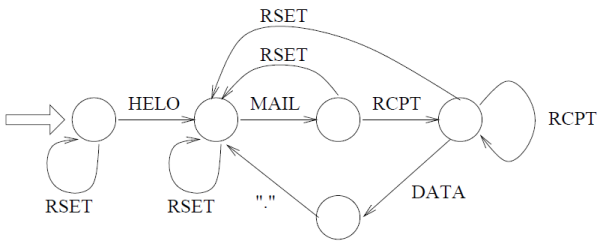

図 7 SMTP サーバの受信分のプロトコル

5.3.2. 低レベルソフトウェアの検証

次に，形式的検証の実用性を確かめるために，現実的なオペレーティングシステムに対して部分的に仕様を書いた上で検証を行った[7][8]．ここではネットワークデバイスのための組込みオペレーティングシステムである Topsy を対象とした．

最初に，バッファーオーバーフローなどのコンピュータのメモリ管理の過失で発生する安全性の問題を考慮して，オペレーティングシステムのカーネルのメモリ管理のソースコードの検証を行った．このために，C 言語に似た言語を Coq 定理証明支援系においてモデル化し，ライブラリとして実装を行った．また，分離論理の拡張もライブラリとして実装し，さらにこれらのライブラリを用いて Topsy のメモリアロケータを検証した． 検証の結果の一つとして，メモリリーク，つまりサービスのアベイラビリティに繋がる誤りを発見した(図 2).

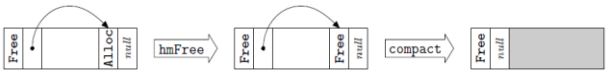

図 8 メモリ管理における誤り

このような誤りはポインタ演算に関係するので，定理証明支援系上で分離論理を用いることが誤り発見のために不可欠であったと考察される．

5.3.3. 組込みオペレーティングシステムの検証に向けて

以上の方法を用いれば他のオペレーティングシステムでも様々な低レベル機能の検証が可能であるが，オペレーティングシステムの複数の部分に基づく仕様，いわゆる高レベル機能仕様の形式的検証は定理証明支援系ではまだ困難である．例えば，タスクアイソエーションというオペレーティングシステムの仕様は，メモリ管理だけでなく，ハードウェアやコンテキスト切り替えなどに依存するので，古典的な分離論理だけで扱えない．

この問題を考慮して，Topsy オペレーティングシステムのハードウェアやカーネルやユーザアプリケーションを抽象化した上でモデル化し，これを用いて Spin モデル検査器において，メッセージ通信サービスとカーネルメモリ領域の保護に関する性質の検証を行った[9].

5.4. まとめ

本研究は，新しい仕様記述言語の性質を調査し，この新しい記述言語と既存の記述言語を定理証明支援系で実装した．この実装で得たライブラリを利用して，現実的なソフトウェア(メールサーバとオペレーティングシステムのカーネルの部分)の厳密かつ形式的な検証を行った．

成果としては，国際学会での発表や学会の論文集と学術雑誌での論文の記載と再利用可能なソフトウェア[10][11]などがある．

参考文献

[1] Reynald Affeldt and Naoki Kobayashi. Partial Order Reduction for Verification of Spatial Properties of Pi-Calculus Processes, In Proceedings of the 11th International Workshop on Expressiveness in Concurrency (EXPRESS 2004), London, UK, 2004.

[2] Ste'phane Demri and David Nowak. Reasoning about transfinite sequences (extended abstract). In Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA 2005), Taipei, Taiwan, 2005. LNCS 3707.

[3] Nicolas Marti, Reynald Affeldt and Akinori Yonezawa. Towards Formal Verification of Memory Properties using Separation Logic. 日本ソフトウェア科学会第 22 回大会論文集, 6 pages. 2005 年 9 月. 高橋奨励賞受賞.

[4] Reynald Affeldt and Naoki Kobayashi. A Coq Library for Verification of Concurrent Programs, In Proceedings of the 4th International Workshop on Logical Frameworks and Meta-Languages (LFM 2004), Cork, Ireland, 2004.

[5] Reynald Affeldt, Naoki Kobayashi, Akinori Yonezawa, Verification of Concurrent Programs using the Coq Proof Assistant: a Case Study, 2004 年並列／分散／協調処理に関する『青森』サマー・ワークショップ（SWoPP 青森 2004）.

[6] Reynald Affeldt, Naoki Kobayashi and Akinori Yonezawa. Verification of Concurrent Programs using the Coq Proof Assistant: a Case Study, IPSJ Transactions on Programming, 2005, Vol. 46. No. SIG 1 (PRO 24) , pp.110-120.

[7] Nicolas Marti, Reynald Affeldt and Akinori Yonezawa. Verification of the Heap Manager of an Operating System using Separation Logic. In Proceedings of the 3rd workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2006). 2006, pp.61-72.

[8] Nicolas Marti, Reynald Affeldt, Akinori Yonezawa. Formal Verification of the Heap Manager of an Operating System using Separation Logic. In Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006), 2006. LNCS 4260.

[9] Nicolas Marti, Reynald Affeldt, Akinori Yonezawa. Model-checking of a Multi-threaded Operating System. 日本ソフトウェア科学会第 23 回大会論文集, 6 pages. 2006 年 9 月.

[10] applpi project: http://staff.aist.go.jp/reynald.affeldt/applpi/

[11] seplog project: http://www.nongnu.org/seplog/