VITC：情報流解析による高安全Cコンパイラ

古瀬　淳 ＊1(元)，＊2　米澤　明憲 ＊1

＊1 東京大学大学院情報理工学系研究科
＊2 Jane Street Global Trading, LLC

● A ● B ● S ● T ● R ● A ● C ● T ●

我々は，メモリ脆弱性を突いた攻撃を受けても，一定の安全性を保ちながら動作を継続する実行プログラムを生成する新しいC言語のコンパイル方法を実装した．メモリ安全なC言語コンパイル技術を使い，不正なメモリアクセスを検知し，バッファオーバフローを回避しつつプログラム実行を継続する方法はすでに存在する．しかし，その継続後の実行が安全かどうかは，その「安全」の定義も含め，あまり議論されてこなかった．そこで我々の枠組みでは，情報流解析による機密保持性を導入し，高機密性データが低機密領域に流出しないことを保証することで，継続後実行の安全性を保証した．

一般的にCプログラムに型キャストが存在すると，情報流解析を静的に行うことは難しい．そこで我々は静的解析が不可能な場合，型キャスト周辺でのみ動的解析を行う方法を開発した．

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．たとえば，Webサーバのプログラムでは，誰でも閲覧できるページ内容のデータと，ユーザから入力されたパスワードとは，データ構造上は同じ文字列であるが，その機密度は異なる．機密情報であるパスワードは間違ってもブラウザ上に表示されてはならない．それを防ぐためにはページ内容とパスワードは個別に管理され，混同を避けなければならないが，通常はそれはプログラムを書く者の責任である．そしてプログラムが正しく機密情報を管理できているかどうか，自動的に確証する方法はない．もし間違ってパスワードがページ内容に上書きされるようなプログラムを書いてしまい，誰もそのミスを見つけなければ，そのWebサーバは機密漏洩を起こしてしまう．

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る．こうして安全性が保証できるプログラムのみをコンパイルすることにすれば，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自動的に防ぐことができる．

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言語に対してはバッファオーバフローに象徴される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，このC言語の最大の障害であったメモリ安全性の欠如は後述のメモリ安全化Cコンパイラ研究の成果を利用することで解決できる．

そこで，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．VITCはCプログラムを検査し，機密漏洩が起こる可能性を認めた場合には危険なプログラムとみなしコンパイルを拒否する．逆に，VITCが安全であるとしてコンパイルしたC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

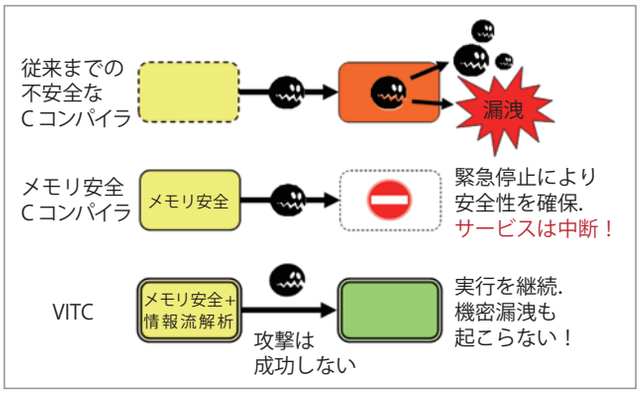

図 -1　既存 C コンパイラシステムとの比較

強い耐性を持つプログラムを生成することができる．攻撃によってメモリ内容が改竄され，プログラムが予期せぬ動作に陥ったとしても，そこから機密情報が漏洩してしまうことはないのだ（**図 -1**）．

VITC の基礎アイディア

【 メモリ安全化 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セキュリティホールがたびたび指摘されている：

```
void f(char *p){
  char buf[100], *q = buf;
  int i;
  while(*p){ *q = *p; q++; p++; }
  ...
```

たとえば，通常の C コンパイラでは，長さ 100 バイト以上の文字列ポインタに上記の関数を適用すると，配列 buf の正しい定義域を越えてメモリの書き込みが行われてしまう（バッファオーバフロー）．これを悪用すれば，プログラム中のデータを改竄できるだけでなく，悪意を持ったコードを注入し，プログラム内で実行させることが可能で，このようなメモリ管理の脆弱性はコンピュータ・ウィルスや不正侵入の恰好のターゲットとなってきた．

この問題に対して，言語研究の分野からはメモリ安全化コンパイラの提案が行われてきた（たとえば文献 3））．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を検知した場合は，そのアクセスを許さず，実行を中断することで，メモリ脆弱性による誤動作，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 中段）．

先ほどの例ではポインタを介したメモリアクセス *p と *q に対し，チェックコードが挿入される．もし，配列 buf 内を指すポインタ変数 q が buf の定義域を越えた値を持っているときに q を介したメモリ書き込みを行うと，チェックコードはその不正アクセスを検知し，プログラムの実行を中断する．

メモリ安全化コンパイラにより，多くの既存 C アプリケーションが変更な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 】

エラー忘却型計算 [1] では，このメモリ安全化コンパイルのアイディアをさらに推し進め，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，たとえば，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．

前記の例では，buf の定義域を外れた q への書き込みがあった場合，エラー忘却型計算では，そこでメモリエラーとして実行を中断するのではなく，たとえば，書き込み自体をなかったことにして実行を継続するといった対策を自動的に行う．

Web 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もの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．

【 情報流解析 】

情報流解析 [4] 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．次は情報流解析でよく使われる例を VITC の文法で書いたものである：

```
int^H h;
int^L l;
...
if(h) { l = 42; }
...
```

プログラム中の変数には，それぞれ機密度が設定され，その変数が持ち得る情報の機密度を表す．ここでは H(*igh*) が高機密，L(*ow*) が低機密とすれば，初めの2行は，変数 h と l がそれぞれ，高機密変数と低機密変数であることを宣言している．最後の行では h の値によって l への代入が行われるかどうかが決定されている．そのため，この行を実行後，h の値が非ゼロかという高機密度の情報は，l の値が 42 であるかどうかという低機密情報から推測できてしまう☆1．情報流解析では，たった1つの真理値であっても，このような高機密から低機密への情報流のあるプログラムは機密を漏洩する可能性があるのでエラーになる．Web のユーザ認証に失敗した場合，その理由として入力されたユーザ名がユーザデータベースに存在しないという真理値をメッセージとして知らせる Web サイトがあるが，これも機密度の高い真理値が漏洩している例である．このようなサイトでは何度も認証を繰り返すことで，データベースに存在する有効なユーザ名を推測することができ，それを利用すればサーバへの不正ログインの難度を大きく下げることができる．情報流解析を使って，ユーザデータベース情報が高機密であることが分かっていれば，ユーザ名が存在しないことを表示することは危険であることがコンパイル時に検出できる．

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[2), 5)]．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ないようである．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得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．

☆1 もともとの l の値が 42 であるときには機密漏れは起きないが，型を使った情報流解析ではこのような特殊な場合は考えない．また，フロー依存な解析ではこの後に続くプログラムの内容によってこの行を安全とみなせることもある．

【 VITC の主張 】

メモリ安全化 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，メモリ安全化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：

```
int^H h;
...
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としよう．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[3)]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，2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ことから h が真であったことが推測できるため，機密漏洩してしまう．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検証できない，もしくは非常に難し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，そのすべてに対し1つ1つプログラマがエラー処理を書くことを要求されるとすれば，それは恐ろしい労力となるだろう．これを自動化するとすれば，必然的にエラー忘却計算が必要となる．

メモリアクセスエラーが起きた際のエラー忘却計算の「勝手な」エラー処理の副作用で，プログラムは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，そのような状況であっても，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こと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 下段）．

以上のように，VITC ではＣのメモリ安全化コンパイル技術を基礎として，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言語プログラムを容易に情報流安全なプログラムへとコンパイルすることができる（**図 -2**）．

これが我々の高安全コンパイラ VITC の基本的アイディアである．以降は簡単にではあるが，VITC の情報流型システムについて説明する．

VITC の型システム

【 型推論 】

$\texttt{int}^H$ h や $\texttt{int}^L$ l, $\texttt{int}^H$ $\texttt{*}^L$ p（高機密度整数を指す，値自体は低機密（Low）なアドレス変数）などの各変数の取り得る機密度 L, H, ... はＣ言語のアトリビュートとして表記する．この機密度情報は基本的には，後述の動的型チェックの所を除き，型推論により自動的に推論される．そのため，VITC でＣ言語プログラムを情報流解析する場合には，最も単純な場合，プログラム中のどのデータが高機密／低機密かを指定する最低限の機密度の仕様を与えるだけでよい．どうしても静的型システムではチェックできない部分がある場合は，動的型チェックを行うように指示する修正を行う必要があるが，それ以外の変数の機密度情報を逐一ユーザが与える必要はない．

情報流解析を行うために，すべての変数宣言や関数宣言に機密度情報をユーザが記載していく必要があるとすれば，そのコストは膨大なものとなる．既存のＣアプリケーションをより安価に安全化するためには精度の高い型推論が必要不可欠である．

より理論的には，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 では ML での静的情報流型システム [5] の基本デザインを踏襲できる．各機密度は束を形成し，束の半順序によって型のサブタイプ関係が定義される．関数は情報流に関する多相型を持ち，HM(X)[6] で説明されている制約型システムおよび推論を適用している．

【 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，静的型を隠蔽し，動的検査を行うために動的型も導入している：

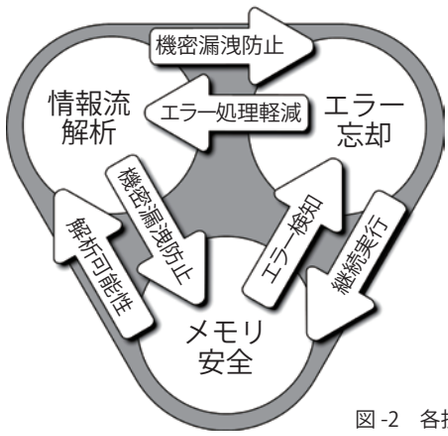

図 -2　各技術間の相互関係

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

たとえば上のソースコードを VITC によって情報流解析すると，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混同している可能性があるとして型エラーになる（型推論から見れば単一化しようとして失敗する）．

このような場合，動的型を使ってエラーを回避することができる：

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

VITC では特殊な動的機密度型 dyn をキャスト中に使うことで，p への代入に際して問題のある型 H と L を隠蔽し，型エラーをひとまず解消することができる．ただし，後に，変数 p を使ったポインタアクセスの際に，指示先がどんな機密度を持つか，実行時動的型検査を行う必要がある．この動的検査もキャストと機密度を使って記述する．上の例では，もし p に $\texttt{int}^L\texttt{*}$ のポインタが入っている場合はテストに成功し，ポインタアクセスによって L の値を取り出すことができる．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．

【 単純な型システムと型キャストの問題 】

ML での情報流解析型システム [5] と異なり，VITC での型理論で最も問題となるのは，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，$\mathtt{int}^{\alpha}\ *^{\beta}$ という型を持つ．α, β は型変数である．このとき，xの型を**単純に**考えると，pの持つアドレス値が代入されるため $\mathtt{int}^{\beta'}$ where $\beta \leq \beta'$ という型であろうと推測できるが，これでは，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たとえば以下のように修正しなければコンパイルできない：

```
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，この単純な型システムではそれを検知できない．

我々の実用的なCプログラムを情報流解析する実験では，初期のVITCが採用していた，このような単純な型システムを使った場合，キャストによって失われてしまう機密度回復操作が予想以上に多かった．つまり，安全にもかかわらず，動的検査を行うためのソースプログラムの修正が必要となる個所が多かったのである：

```
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，たとえば struct S^H s などといちいち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．たとえば：

```
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得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，キャストで機密情報が失われてしまうような単純な型システムでは，関数からアドレスが脱出し得るすべ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．

【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，VITCの最新型システムでは，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ている．つまり，上の例のxの型は，この型システムでは $\cdot^H\ \mathtt{int}^{\alpha}$ where $L \leq \alpha$ 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 $\cdot^H\ \mathtt{int}^{\beta}$ where $\alpha \leq \beta$ 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，この最新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 ℓ，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（**図-3**）．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 $\ldots{*}\lambda{*}\ldots{*}\lambda$ を表す λ^{ω} が用意されている．このアイディアは我々は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をせずに静的に解析できる．

$$
\begin{array}{lll}
l & ::= & \ell \mid \alpha \\
\lambda & ::= & l \mid dyn \\
\tau & ::= & \tau * \lambda \mid a \mid \lambda^{\omega} \mid \sigma * \lambda \\
\sigma & ::= & \forall \overline{a\alpha}\,[K]\,.\,\overline{\tau}^{\lambda} \rightarrow \tau \\
K & ::= & k, \cdots, k \\
k & ::= & l \leq l \mid \alpha
\end{array}
$$

図-3　VITC の型

実装と実験

VITC コンパイラは，基本となるメモリ安全化 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 [3] を採用し，エラー忘却計算と，情報流解析型システムを拡張することで実装されている．また，その開発は，日立製作所（C 言語のポインタ解析），とめ研究所（情報流解析ライブラリの作成），A.N. Lab（フロー依存型システムの開発）の各企業の協力も得て行われた．

また，本稿では触れることはできなかったが，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 thttpdWeb サーバに対して VITC による安全化実験を行った．まず，thttpd にメモリ脆弱性を意図的に付け加え，通常の C コンパイラでコンパイルしたこの改変サーバに対して，一定長を越える入力を Web ブラウザから与えるとバッファオーバフローにより，サーバ乗っ取りを行ったり，パスワードファイルなどの機密情報をブラウザにダウンロードさせることが可能となることを確認した．そして，メモリ安全化，エラー忘却（+ メモリ安全化），VITC による一部モジュールの情報流解析，の 3 段階の安全化を行った改変サーバに対し同じ攻撃を行い，それぞれを比較した：

- Fail-Safe C によるメモリ安全化版サーバでは，メモリ安全性によりプログラムは攻撃を探知するため，サーバ乗っ取りやデータ改変は不可能ではあったが，攻撃によりサーバはダウンしてしまった．
- メモリ安全化にエラー忘却計算を組み合わせたコンパイルでは，攻撃によるウィルスコードの注入は不可能である．サーバは攻撃を受けた後も，エラー忘却により実行の継続が可能であったが，攻撃によるメモリ改変による誤動作から，機密ファイルをブラウザに送信させてしまうことが可能であった．
- VITC 版サーバでも，攻撃を受ければ予期せぬメモリ改変が行われる．しかしそのような状況でも情報流解析による安全性は保たれているため，機密漏洩は起こらない．機密ファイルをブラウザへ送信しようとする際には，動的型チェックが機密情報を公開することを事前に発見，動的型エラーとして報告する．このエラーはエラー忘却機構により処理され，最終的にブラウザには機密ファイルの内容ではなく，無害な空文字列が送信された．

なお，VITC で生成される実行プログラムの速度効率は Fail-Safe C のそれとほぼ同じである．通常の C コンパイラと比べて数倍遅くなるが，このオーバヘッドのほとんどはメモリ安全化のために挿入されたチェックコードによるものである．VITC ではそれに加えて動的型検査のコストが必要☆2 だが，機密度の等値比較を行うだけなのでメモリ安全化コストに比べればほとんど無視できる．

☆2 静的型システムによって保証される安全性については実行時に検査する必要がないため，まったくオーバヘッドはない．

結論

VITC コンパイラではメモリ安全化 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) Rinard, M. et al. : Enhancing Server Availability and Security Through Failure-obliviouscomputing, In *Proceedings of the 6th Symposium on Operating Systems Designand Implementation San Francisco, CA* (Dec. 2004).
2) Myers, A. C. : JFlow : Practical Mostly-static Information Flow Control, In *Symposium on Principles of Programming Languages*, pp.228-241 (1999).
3) Oiwa, Y., Sekiguchi, T., Sumii, E. and Yonezawa, A. : Fail-safe ANSI-C Compiler : An Approach to Making C Programs secure (progress report), In *Lecture Notes in Computer Science*, Vol.2609 (Feb. 2003).
4) Sabelfeld, A. and Myers, A. : Language-based Information-flow Security, In *IEEE Journal on Selected Areas in Communications, 21(1), 2003* (2003).
5) Simonet, V. : Flow Caml in a Nutshell, In *Proceedings of the First APPSEM-II Workshop*, pp.152-165 (Mar. 2003).
6) Sulzmann, M., Odersky, M. and Wehr, M. : Type Inference with Constrained Types, In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)* (1997).

（平成 20 年 8 月 5 日受付）

古瀬　淳　jfuruse@janestcapital.com
INRIA，およびパリ第 7 大学にて計算機科学修了（Ph.D）．東京大学，産学官連携研究員を経て，2008 年より Jane Street にてクオンツ業務．

米澤　明憲（正会員）　yonezawa@yl.is.s.u-tokyo.ac.jp
MIT 計算機科学修了（Ph.D），1988 年より東京大学コンピュータ科学専攻教授．日本ソフトウェア科学会理事長，ドイツ国立情報学研究所（GMD）科学顧問歴任．現在，東大情報基盤センター長．ACM Fellow，2008 年国際オブジェクト技術協会 Dahl-Nygaard 賞受賞．