VITC: 対攻撃耐性コード生成コンパイラ

VITC: Safe C code compilation against attacks

古瀬 淳　　米澤 明憲
Jun FURUSE　Akinori YONEZAWA

東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
{furuse,yonezawa}@yl.is.s.u-tokyo.ac.jp

我々は、近年研究されてきたメモリ安全な C 言語のコンパイル技術と、型システムによる情報流解析を組み合わせて、バッファ・オーバーフローなどのメモリ脆弱性を突いた攻撃を受けても安全に動作を継続する事のできるコードを生成する新しい C のコンパイル技術 VITC を提案する。攻撃後の実行継続は、もともとのプログラムの意味論から逸脱した行為であるが、機密漏洩の可能性の無いことを保証することで、実行継続を正当化できる。

1 はじめに

近年、しばしば報道されるコンピュータ・プログラムの脆弱性発見やそれを利用した攻撃による被害の多くは、攻撃対象となるプログラムの多くが C 言語で書かれていることに原因がある。効率性を重視した C 言語では、メモリ管理の一部は言語内ではなく、プログラマに任されているため、バッファオーバーフロー等のメモリ管理のバグによる脆弱性が生じ易く、結果としてプログラムは攻撃者に乗っ取られ、機密漏洩という最終的被害にまで発展する。

また、この機密情報漏洩による被害は、メモリ脆弱性攻撃にのみ特有なものではない。C 言語には、機密保護のための機構が存在しないため、メモリ脆弱性とは関係なく、機密を漏洩してしまうバグを持つプログラムは多数存在すると思われる。

これら C 言語におけるメモリ脆弱性と機密漏洩の二つの問題の解決策として、我々はメモリ操作の安全化と情報流解析による機密漏洩検知を備えたコンパイル手法を提案する。これらの機構は互いに重要な役割を果たす。従来、C 言語では非常に困難であった情報流解析はメモリ安全化技術と組み合わせることで可能になる。また、メモリ安全化により、各メモリブロックの有限の定義域を取り払い、全アドレス空間に対して (仮想的に) 有効とすることによって、オーバーフローアクセスにもプログラムを中断すること無く実行を継続することが可能となるが、これは C 言語の未定義部分の挙動の大幅な拡張であるため、プログラマの漠然とした予想、つまり、通常使用している既存の C コンパイラでの挙動と異なる恐れがある。このような挙動の違いによる意図しない機密漏洩の可能性を情報流解析によって排除することで、継続実行の安全性を正当化することが出来る。

我々が製作しているコンパイラ、VITC[1]では、以上の機能により、プログラムは機密情報を漏洩しないことを保証しつつ、メモリ脆弱性攻撃に対して実行を安全に継続することができる、対攻撃耐性を備えた C プログラムのコンパイルが可能である。

以下、本稿では、2 節と 3 節で関連研究紹介を交えつつ我々のアイデアを解説する。その後 4.3 節で C 言語における型システムによる情報流解析の問題とその解決の提案を行い、その後の節で実装と、これからの課題を説明する。

2 C 言語のメモリ安全化と対攻撃耐性

2.1 既存の緊急停止によるメモリ安全化とその問題

C 言語のメモリ脆弱性問題に対しては既に多数の解決策が提案されているが、主に次の三つのグループにわけることができる:

1. StackGuard[2] をはじめとする、Canary value を使ったメモリの不正アクセスの監視
2. NX-bit などハードウェアによる注入コードの実行拒否
3. CCured[5], Fail-Safe C[6] などの C 言語のメモリ管理の完全な安全化

1 では、メモリ脆弱性攻撃が注入コード実行の手がかりとなる、関数リターンアドレスの改変を、そ

[1] Vulnerability and Intrusion Tolerant Compilation の略

の前後に Canary と呼ばれる値を挿入し、Canary の改変を攻撃の証拠とみなす。

2 は、データ領域上でのコードの実行を拒否することで、注入される攻撃コードの実行を防ぐ。

3 の方法では C 言語を、その仕様を保持したまま、ML 等と同等のメモリ安全性を付加するものである。原理的には各メモリブロックへのアクセスを監視し、ブロックの定義域をオーバーするアクセスを拒否する。1,2 の方法と違い、メモリアクセスに対し実行時検査を行うため、効率が落ちるが、近年の型理論の発展や、ポインタ解析技術の向上 [9] により、実用的に問題のない効率化を達成しつつある。

以上の手法は、内部での解決法こそ違うが、不正なアクセスを検知した際には全て同じ対応を取る。プログラムは、問題発生時にその実行を緊急停止することで安全性を確保するのである。この緊急停止による安全性 (*fail-safety*) は、そのまま実行を継続した場合に引き起こされるプログラムの乗っ取りや、データの改変を防ぐという点では、脆弱性攻撃への有効な防御策といえる。

しかしながら、この fail-safety はサーバー等の継続的に動作すべきプログラムの安全性としては不十分である。なぜなら、攻撃によりプログラムの提供するサービスは中断されてしまうからである。Fail-safety では、メモリ脆弱性を利用した攻撃は、乗っ取りや情報漏洩などは引き起こさないが、 DoS アタックとして機能してしまう。これは、メモリ脆弱性攻撃の対象とされるプログラムにサーバーが多いということを考えると無視できない問題である。これを解決するためには、プログラムが安全なだけでなく、さらに**対攻撃耐性**を持って継続動作することが望まれる。

2.2 Fail-safety を越えて対攻撃耐性へ

メモリ脆弱性攻撃によるプログラムの実行中断を回避するためにはバッファオーバーフローが原理的に起こらない、すなわち、各メモリブロックの有限の定義域を取り払い、全アドレス空間に対して (仮想的に) 有効とすることによって、全てのメモリアクセスを成功させる方法がある。この *boundless memory block* は、現実にはブロックを必要に応じて伸長するといった形で実装できる。この方法は、メモリ攻撃が成功し、データが破壊されてからその攻撃を検知する、2.1 節で挙げた 1 や 2 の方法では、原理的に不可能であるが、3 では、不正なメモリアクセスは、それが実際に発生する前に検知されるため、直前に配列の伸長を行うことが可能である。この方式では既に Rinard らが Safe C コンパイラ [1] に基づいた実験を行っている [3]。

Boundless block を使えば、バッファオーバーフロー問題は元のプログラムの意味を自然な形で拡張しつつ解決することができる。例えば以下の関数ではユーザ名とパスワードを受け取り、その組を `buf` に作成する:

```
f(char *user, char *pass)
{
  char buf[256];
  sprintf(buf, "%s:%s", user, pass);
  ... /* use of buf */
```

ここで、入力文字列が `buf` の長さ、256 文字を越える場合、バッファオーバーフローが発生するが、boundless block を使うと、メモリブロックの伸長が行われ、あたかも `buf` のサイズが元々十分にあったかのように継続動作する。脆弱性攻撃を防ぎつつ、攻撃を意図してはいないが、想定よりも長すぎる入力に対しても「正しく」動作する点で、この方法は非常に有望そうに見える。

2.3 安全な実行継続

しかしながら、この方法では、ブロックの伸長を行って実行を継続することで、思わぬ機密漏洩が起こる可能性が生ずる。次の例を考える:

```
f(char *user, char *pass)
{
  char buf[256];
  sprintf(buf, "%s:%s", user, pass);
  ... /* use of buf for secret data */
  bzero(buf, 256);
  ... /* use of buf for public data */
```

前の例に加えて、バッファをクリアする命令が付加されている。クリア前はバッファはパスワードを含む機密情報を格納しているが、クリア後は、機密とは関係ない目的に使用されているとする。バッファ伸長された場合においても、クリアされる部分は先頭の 256 文字のみで、伸長部分に存在する機密情報は残され、以降の操作によってはそれが公開部分に漏洩する恐れがある。この例に限れば、バッファ伸長とともにクリアサイズも動的に変更することで問題を回避できるが、これを一般化することは難しい。

対攻撃耐性 C 言語プログラムの生成では、元来未定義であった不正なメモリアクセス発生後のプログラムの動作をいわば勝手に決定することで継続動作

を実現するが、この継続動作はプログラマの意図を外れ思わぬ情報漏洩を引き起こす可能性がある。その点では、従来の fail-safety では問題発生時点でプログラムを緊急停止させることから、この問題は存在しなかった。その点では、boundless block を使った不用意な継続実行は fail-safety よりも不安全である。

攻撃発生後のプログラムの継続実行が安全であると正当化するためには、情報洩れが起こらないことを保証する必要があり、プログラムの情報流解析が必要である、これが本稿での我々の主張である。情報流解析による安全性は、メモリ安全性とは異なる新たな安全基準であるが、そもそも、メモリ脆弱性攻撃への従来の対策の目標が、攻撃によって引き起こされる機密漏洩の防止であったことを考えれば、この新基準の導入は不自然な事ではない。

3 C 言語における情報流解析

安全型システムによる静的情報流解析 (全体像が [8] によくまとめられている) では、情報の流れを型システム上で表現し、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 に適用した JFlow[4] や ML の拡張である Flow Caml[10] などが知られている。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。理由は明白で、C 言語自身が提供するメモリ管理は Java や ML に比して不完全で、情報流を追跡することは非常に困難だからである。

しかし、[5] や [6] のような完全なメモリ安全化技術を導入すれば、C 言語はメモリ安全な imperative な言語とみなすことができるため、情報流解析を行うことが可能となる。

4 C 言語のための安全型システム

以上のように、C 言語におけるメモリ安全性と情報流解析は相補的であり、対攻撃耐性を持った C プログラムを作成するためには両者が不可欠である。メモリ安全性は既存の技術により確保されているものと仮定し、以降は情報流解析について論ずる。

我々の C 言語のための安全型システムは、ML での情報流解析のための型システム [7] を基としている。ただ、(安全ラベル多相性を除く) 従来の ML 型システムでの多相型や例外は必要ないため、より単純化されている。また、ポインタはキャスト部分を除いてリファレンスと同様の型付けルールを使用できる。本稿では、紙面の関係上、安全ラベル多相性と、関数定義は扱わない。さらに、情報流解析で特徴的な条件分岐によって引き起こされる実行フローが持つ機密度、いわゆる pc についても省略した。VITC では、これらの機能はいずれも従来の情報流解析と同じ手法で実現されている。

$$\text{Types} \quad t ::= \text{int}^{\ell} \mid t\ \text{ptr}^{\ell}$$

e	::=		expressions
	|	$n : t$	integer
	|	$x : t$	variable
	|	$*e : t$	dereference
	|	$*e = e : t$	update
	|	$(t)e : t$	cast
	|	$e + e : t$	addition
	|	$\text{new}(t) : t$	allocation
	|	$\text{let } x : t = e \text{ in } e : t$	let binding

図 1: 型と式

4.1 型付けの目的

安全型システムでは、正しく型付けされたプログラムは、機密漏洩を起こさないことを保証する。これは、より正確には non-interference、つまり、高い機密度を持つ情報を変更して同じプログラムを動作させても、その変化が低い機密性を持つ式の値に影響を与えないという性質で定義できる。これを保証するために、安全型システムでは情報の流れを追跡し、情報の流れた先の式は、その源流の式よりも必ず同等かより高い機密度を持つ、という型付けを行う[2]。

4.2 型と式

型と式は実装のベースである Fail-Safe C[6] の物を基にした (図 1)。型には機密度を表す安全ラベル ℓ が付加されている。ラベルの集合 $\mathcal{L}$ は半順序 $\leq$ の元で束をなす。本稿では、$\mathcal{L} = \{L, H\}, L \leq H$ という例を用いる。例えば int^{H} は高い (High) 機密度の整

[2]我々の型システムの健全性、つまり、型付けされたプログラムが本当に non-inteference を満たすかどうか、の議論は本稿では行わない。

$$\Gamma \vdash n : int^{\ell} \qquad \frac{t \in \Gamma(x)}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e : t' \quad t' \leq t}{\Gamma \vdash e : t}$$

$$\frac{\Gamma \vdash e : t' \text{ ptr}^{\ell} \quad t' \leq t \quad \ell \lhd t}{\Gamma \vdash *e : t}$$

$$\frac{\Gamma \vdash e_1 : t \text{ ptr}^{\ell} \quad \Gamma \vdash e_2 : t \quad \ell \lhd t}{\Gamma \vdash *e_1 = e_2 : t}$$

$$\frac{\Gamma \vdash e_1 : \text{int}^{\ell} \quad \Gamma \vdash e_2 : \text{int}^{\ell}}{\Gamma \vdash e_1 + e_2 : \text{int}^{\ell}}$$

$$\frac{\Gamma \vdash e_1 : t \text{ ptr}^{\ell} \quad \Gamma \vdash e_2 : \text{int}^{\ell} \quad \ell \lhd t}{\Gamma \vdash e_1 + e_2 : t \text{ ptr}^{\ell}}$$

$$\Gamma \vdash \text{new}(t) : t \text{ ptr}^{\ell}$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma[x \mapsto t] \vdash e_2 : t'}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t'}$$

$$\frac{\ell \leq \ell'}{\ell \lhd \text{int}^{\ell'}, \quad \ell \lhd t \text{ ptr}^{\ell'}}$$

図 2: キャスト以外の型付け規則

数、int^L ptr^L は低い (Low) 機密度を持つ整数が格納されたメモリブロックへのポインタの型である。

ラベル間の半順序 $\leq$ はサブタイプ関係へと拡張される:

$$\frac{\ell \leq \ell'}{\text{int}^{\ell} \leq \text{int}^{\ell'}} \qquad \frac{\ell \leq \ell'}{t \text{ ptr}^{\ell} \leq t \text{ ptr}^{\ell'}}$$

ポインタの中身が invariant なのはリファレンスのサブタイピングと同様である。また、最外ラベルを取得する関数 l を定義しておく:

$$l(\text{int}^{\ell}) = \ell \qquad l(t \text{ ptr}^{\ell}) = \ell$$

式 new(t) は要素の型が t であるような新しいメモリブロックを作成する。Boundless block を仮定しているため、初期領域サイズは省略されている。

4.3 型付け規則

型判定 $\Gamma \vdash e : t$ は、型環境 Γ において、式 e が型 t を持つことを意味する。キャスト以外の型付け規則は図 2 の通りで、ごく自然である。

最も問題となるのは C 言語特有の機能であるキャストの型付けである。キャストは型を変更することで実際の値と元の型との結び付きを断ち切ってしまうため、静的解析だけでは安全性は保証できず、動的な型検査が必要である。型判定を拡張し $\Gamma \vdash e : t \rightsquigarrow e' : t$

$$\text{int}^{\ell} \preceq \text{int}^{\ell} \qquad \text{int}^{\ell} \preceq t \text{ ptr}^{\ell}$$

$$\frac{t \preceq t'}{t \text{ ptr}^{\ell} \preceq t' \text{ ptr}^{\ell}}$$

$\text{check}(v, int^{\ell}) = v$
$\text{check}(ptr(t_0), t \text{ ptr}^{\ell}) = ptr(t_0)$ when $t \preceq t_0$
$\text{check}(v, t)$ fails, otherwise

図 3: キャストのための動的型検査

として、e 中の実行時検査コードを埋め込んだ式を e' とすれば、キャストの型付け規則は、

$$\frac{\Gamma \vdash e : t' \rightsquigarrow e' \quad l(t) = l(t')}{\Gamma \vdash ((t)e) : t \rightsquigarrow \text{check}(e', t) : t}$$

となる。この規則では最外ラベルさえ一致していれば、静的にはいかなる自由なキャストをも許すが、実行時に式 e を評価し、その結果値 v が型 t にキャスト可能かどうかを判定する。我々は fat pointer[6] を使用するため、結果値 v がポインタ $ptr(t)$ か純整数 n かの判別が可能である:

$$v ::= n \mid ptr(t)$$

ポインタ値 $ptr(t_0)$ からは、ポインタが指すメモリブロックの要素型 t_0 — new(t_0) でブロックを作成した際の型引数 — を得ることが出来る[3]。判定は check 関数 (図 3) によって行われる。結果 v がポインタ $ptr(t_0)$ の場合、実際のブロックの型 t_0 ptr^{ℓ} に存在するラベルを消去し、単純化する方向にのみキャストを許可する。例えば、

$$\begin{array}{llllll} t_0 = & \text{int}^H & \text{ptr}^H & \text{ptr}^L & \text{ptr}^L \\ t_1 = & & \text{int}^H & \text{ptr}^L & \text{ptr}^L \\ t_2 = & & & \text{int}^L & \text{ptr}^L \\ t_3 = & & & \text{int}^H & \text{ptr}^L \end{array}$$

とすれば、次の t_0 は t_1,t_2 へとキャスト可能である。ラベルが一致していなかったり (例えば t_0 から t_3)、逆に型を複雑化するようなキャスト (t_2 から t_0) 等はエラーとなる。

この条件の利点は、各キャスト時のチェックを通りさえすれば、その後のメモリアクセスの安全性は保証されるため、その後のポインタを介したメモリアクセス毎に動的検査を行う必要がなく効率的である

[3] ポインタにはさらに、ポインタベースアドレス値とそのオフセットも格納されているが、ここでは重要では無いため省略する。

ことである。また、ブロック作成時の型より複雑な型へとキャストを行うこと、例えば純整数からポインタへのキャスト等は、それ以外の場合そ比べれば少数であると思われる。また、不幸にしてそのような場合が存在する場合でも、比較的簡単な書き直しで、check の条件に適合するキャストに分解できる事が多い。例えば、

```
int x;
int p[3];
p[0] = &x;
((int**)p)[0];
```

というプログラムでは、`((int**)p)[0]` が `p` の型 int ptr (ラベルは省略する) を int ptr ptr にキャストするため我々のシステムでは即時に失敗するが、これを `(int*)(p[0])` という同じ結果を返す式に書き換えれば、`p[0]` の評価結果は *ptr*(int ptr) であるから、キャスト検査は check(*ptr*(int ptr), int ptr) という、複雑性の向上しない、好ましい形に変更できる。

5 実装

我々は現在、メモリ安全なコードを生成する ANSI-C 準拠 Fail-Safe C コンパイラ [6] の研究を元に、情報流解析を行うプロトタイプコンパイラ VITC を作成中である。fat pointer や、各メモリブロックの型情報など、情報流解析で必要となる機能は Fail-Safe C と共通する部分が多く、かなりの部分の共有が可能である。

6 これからの課題

多くの C 言語プログラムに対し、情報流解析を行うためには、4.3 節で導入した安全型システムにおけるキャスト条件の緩和が必要である。ただし、check の条件を通らないキャストの安全性判定は、キャスト式部分で判定することは不可能であり、各メモリアクセスにおいて動的型検査を行わなければならないと予想される。その際に発生するであろうオーバーヘッドの低減が将来の課題である。

また、キャスト検査の失敗は実行時エラーを起こしプログラムは緊急停止する。残念な事に、これは対攻撃耐性の「問題があっても動きつづけるプログラム」という、そもそもの目標と矛盾する挙動である。この問題を克服するため、機密漏洩エラーを感知した場合、漏洩しようとする機密データを公開データに置き換えることで漏洩を防ぎ、安全に実行を継続するシステムを構築する予定である。

7 まとめ

我々は、攻撃を受けても情報を漏洩すること無く、かつ安全に実行継続する C 言語プログラムの生成に向けて、メモリ安全化と情報流解析による機密情報保護を組み合わせた対攻撃耐性コンパイラのアイデアを提案した。この二つの安全性は相補しあうことによって、C 言語プログラムの対攻撃耐性を実現する。

また、我々は C 言語における型キャストに関する情報流解析を静的型付けと動的検査を開発した。キャスト間の型付けの制限により、動的検査は各キャストで行うだけでよく、各メモリアクセスでの検査は不要となる。また、この型付けの制限はそれによって得られる効率性に比べ小さく、多くの場合簡単なキャストの書き換えによって吸収されることを示した。

参考文献

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[2] Crispan Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.

[3] Martin Rinard et al. Enhancing server availability and security through failure-oblivious computing, December 2004.

[4] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[5] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

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

[7] Francois Pottier and Vincent Simonet. Information flow inference for ML. In *Symposium on Principles of Programming Languages*, pages 319–330, 2002.

[8] A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.

[9] Tatsurou Sekiguchi. A practical pointer analysis for the C language. In *Computer Software, vol. 21, no. 6*, pages 34–49, Nov 2004.

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