Vol.2017-OS-141 No.4
2017/7/26

C言語からHaskellへの変換によるプログラム難読化

大羽 史将[†1,a)]　加藤 和彦[†1,b)]　阿部 洋丈[†1,c)]　長谷部 浩二[†1,d)]

概要：プログラムの難読化とは、同じ動作のまま元のプログラムを人間や解析ツールが解読しづらい形式に変換する技術である。そのうちの 1 つに、Wang らが提案した「言語変換による難読化」という方式がある。Wang らは C 言語のプログラムを論理型言語 Prolog のものに変換し、各言語間の実行モデルの違いを利用することで難読化を実現した。しかし、変換後のプログラムの実行時間が平均約 150 倍増加するという問題点があった。そこで本研究では、実行性能の低下を抑えつつ言語変換による難読化を実現するために、C 言語プログラムを純粋関数型言語 Haskell のそれに変換する手法を提案および実装した。この際、既存の C 言語ライブラリを利用するためにメモリの相互運用性は重要な課題である。我々はこれを解決する手法を考案した。そして評価実験により、Wang らの手法と比較して実行性能の低下を抑えられることを示した。

Ooba Fumiyuki[†1,a)]　Kato Kazuhiko[†1,b)]　Abe Hirotake[†1,c)]　Koji Hasebe[†1,d)]

1. 導入

プログラムの難読化は、同じ動作のまま元のプログラムを人間や解析ツールが解読しづらい形式に変換する技術である。その目的は、プログラムのリバースエンジニアリングや不正な改変を防止することである。また、難読化によってプログラムの動作を攻撃者が理解しづらくなることにより、攻撃者によってそのプログラムに含まれる脆弱性が発見されてしまうのを遅らせることができるという副次的な役割もある。

このような目的のために、これまで様々な難読化手法が提案されてきた。近年利用されている有効な難読化技術に、プロセスレベルの仮想マシン（プロセス VM）を用いる手法 [8] がある。これは、対象の機械語プログラムに独自のプロセス VM を組み込み機械語命令の一部をそのバイトコードへ変換するため、この難読化が行われたプログラムは既存の静的解析ツールによる解析が困難になるというものである。しかし最近の研究で、この手法を用いた難読化を効率良く取り除く手法が提案され [3][9]、これにより、プロセス VM を用いる多くの難読化手法は無効化されてしまう可能性が出てきた。

†1 筑波大学 (Uniersity of Tsukuba)
a) ooba@osss.cs.tsukuba.ac.jp
b) kato@cs.tsukuba.ac.jp
c) habe@cs.tsukuba.ac.jp
d) hasebe@cs.tsukuba.ac.jp

これを受けて Wang らは、あるプログラミング言語によって記述されたプログラムを異なるパラダイムの言語のプログラムに変換し、言語間の実行モデルの差を利用して難読化を行う「言語変換による難読化（translingual obfuscation）」という新しい難読化手法を提案した [12]。この手法は、同じ動作をするプログラムであってもそれを記述するプログラミング言語ごとに実行モデルは異なり、生成される機械語プログラムも異なる実行手続きを示すという点を利用する。また、変換前後のプログラミング言語の実行モデルの差が大きいほど、元のプログラムのソースコードは推測されにくくなるという特徴がある。

Wang らは、言語変換による難読化の概念実証として、C 言語によって記述されたプログラムを論理型のパラダイムを持つ Prolog 言語のプログラムへ変換する手法を実現している。一般に C 言語プログラムは、その制御構造やデータ構造の抽象度が低いため、CPU の動作を厳密に制御するのには適している。しかし、これから生成された機械語プログラムからは比較的容易に元のソースコードが推測でき、その動作が解析されてしまう。そこで、Wang らは C 言語プログラムの単純な制御構造やデータ構造を Prolog の高い抽象度による表現に変換し、Prolog の処理系を利用して実行ファイルを生成することで難読化を達成した。Prolog コンパイラは、Prolog の動作モデル（抽象機械）を CPU 上でシミュレートするようなコードを生成する。そのため生成された実行ファイルを解析すると、抽象機械を

構成する個々のサブルーチンの動作は推測できるかもしれないが、それらがどのように相互に関係し、入力となったPrologプログラムの動作を実現しているのかを理解するのは困難となる。

Wangらは、このようにして得られたプログラムの難読化の効果や実行性能を評価した。この結果、言語変換による難読化は有効な手法であると結論付けた。しかし、変換後のプログラムの性能が大きく低下してしまうという問題があった。

本研究では、C言語によって記述されたプログラムを関数型言語へ変換することにより難読化を行う手法を提案する。関数型言語の実行モデルでは、関数は機械語の実行コードとその実行時に利用する環境の情報の組み合わせとして表現され、これによりC言語プログラムを、関数ポインタによる間接アクセスが多用されるプログラムに変換できると考えられる。一般に、静的に実行ファイルを調べるだけでは、ポインタによる間接アクセスが発生する場合、正しい解析結果を得ることは難しいとされている。そのため、関数型言語への変換による難読化は有効な手法であることが期待できる。本研究では、関数型言語の一例としてHaskell言語を利用し、コンパイラ基盤LLVMを介することにより、C言語のプログラムをHaskellのそれに変換する手法を提案する。また、提案手法による難読化の効果と実行性能の評価を行った。この結果、提案手法は難読化として有効であるということを示せた。また、Wangらの手法よりも実行性能の低下を抑えることができた。

本稿は、以下の章によって構成される。第2章では、本研究の先行研究であるWangらの手法とプロセスVMを用いた難読化手法について説明する。第3章では、本研究においてC言語の変換先となるプログラミング言語Haskellとその処理系、そしてコンパイラ基盤であるLLVMについて説明する。第4章では、提案手法及びその実装について説明する。第5章では、提案手法による難読化の効果および実行時の性能を測定する実験と、その結果について説明する。第6章では、結論と今後の課題について述べる。

2. 関連研究

2.1 プロセスVMを用いた難読化

プロセスVMを用いた難読化手法[8]では、難読化対象の実行ファイルにプロセスレベルの仮想マシン（インタプリタ）を組み込む。さらに、元の機械語命令の一部を、このインタプリタが解釈できるバイトコードに変換することで難読化を行う。機械語を独自のバイトコードに置き換えることにより、既存の実CPUの機械語向けに作られた解析ツールが使えなくなり、その結果実行ファイルの解析が困難になる。

言語変換による難読化では、プログラムのコンパイル時に変換先言語の実行モデルを実現するコードが生成される。そのため、バイトコードのような中間表現やそれを解釈するインタプリタを実行時に必要としないという点で、プロセスVMを用いた難読化と大きく異なる。

2.2 C言語からPrologへの変換による難読化

Wangらは、プログラミング言語の実行モデルの差を利用して難読化を行う「言語変換による難読化」というアイデアを主張し、その概念実証としてC言語プログラムを論理型のパラダイムを持つProlog言語のプログラムへ変換する手法を提案した[12]。彼らの手法では、C言語プログラムの制御構造とメモリ構造をそれぞれ、Prologの特徴であるバックトラックと単一化を用いた処理へ変換することにより難読化を行う。これにより、変換後のプログラムの制御フローグラフは複雑なものとなり、また、難読化前後の実行ファイルを用いたバイナリ比較では、構文的・意味的な類似度は低くなり、彼らの手法は難読化として有効であると結論付けられた。一方、変換後のプログラムは実行時のオーバーヘッドが大きく、性能が低下するという問題があった。具体的には、ベンチマークに利用したプログラム群について、各プログラムのソースコード中の全関数のうち50%をPrologに変換すると、実行時間が平均150倍増加した。そこで、彼らはプログラム全体をPrologに変換するのではなく、必要な部分のみを選択して難読化することで、大きな性能低下を回避しつつ難読化の効果を得られると主張した。

3. 準備

3.1 プログラミング言語Haskell

Haskell[7]とは、値が必要になるまでその計算を行わない遅延評価という特徴を持つ、純粋関数型な汎用プログラミング言語である。本研究では、C言語のプログラムをHaskellのそれに変換することにより難読化を行うので、Haskellプログラムから生成される機械語プログラムの構造が重要な点となる。

本研究では、Haskellコードから実行ファイルを生成するコンパイラとしてGHC[11]を利用する。GHCにより生成される実行ファイルでは、Haskellの各関数は小さな粒度の実行コードのブロックに分割され、関数ポインタにより互いを参照し合うような構造に変換される[5]。これによって、プログラムの制御構造が複雑になり、解析の難易度が上がる。また、Haskell抽象機械の動作についての事前知識がない限り、各ブロックがプログラムの実行に与える影響を理解するのは困難であり、このことは既存の静的解析ツールによる解析を行っても、意味のある結果が得られないことを示している。このようにしてプログラムの構造の理解を妨げることは、実行時における特定のメモリアドレスに対する不正な読み書き操作を困難にすることにもつながる。さらにGHCのランタイムにおける、Haskellの

遅延評価を実現するコードやデータ構造、ガーベジコレクションのようなプログラムのアルゴリズムとは本質的には関係ない処理によって、デバッグのような動的解析の難易度も上がる。本研究では、これらの点を難読化の要素として利用する。なお、GHC は後述する LLVM を利用して実行ファイルを生成している [10]。

3.2 コンパイラ基盤 LLVM

LLVM[6] は、コンパイラの機械語生成処理を担ったり、プログラムを解析するために用いられるフレームワークである。プログラミング言語に依存しない、独自の抽象機械用の中間表現（LLVM アセンブリ言語）を提供している。LLVM を利用するコンパイラは、入力となるソースコードをこの中間表現に変換することにより LLVM の機能を利用できる。本研究の変換手法では、C 言語プログラムを Haskell のものに変換する過程で LLVM を利用するため、これに関する LLVM の機能について述べる。

LLVM は抽象機械としてレジスタマシンを提供している。使用できるレジスタの数に上限はないが、レジスタへの値の設定については静的単一代入（static single assignment, SSA）形式を取っているため、レジスタへの値の再代入は禁止されている。そのため、更新が必要なデータはこの抽象機械が提供する仮想的なメモリ空間に保存する。

次に、LLVM アセンブリ言語について説明する。例として、リスト 1 に示す、ループにより数をカウントし出力する[*1]単純な C 言語プログラムを、Clang[1] という LLVM を利用した C コンパイラに入力する。これによって生成される LLVM プログラムは図 1 のようになる。本来 LLVM プログラムは、アセンブリ言語と同様に抽象機械に対する命令を羅列したものであり、可読性が低い。そこで、このプログラムを制御フローグラフとして可視化したものが図 1 である。

この LLVM プログラムはリスト 1 の main 関数に対応しており、LLVM の関数は複数のノード（基本ブロック）によって構成される。各基本ブロックは LLVM アセンブリ命令によって構成され、基本ブロックは必ず別の基本ブロックへのジャンプまたは条件分岐（br 命令）、もしくはそれが属する関数からの復帰（ret 命令）により終端されるという規則がある。

リスト 1 中の変数 i は、図 1 では%i という変数に対応している。alloca 命令によりこの変数の値を保存するための領域が LLVM 抽象機械の仮想的なメモリ上に割り当てられ、値が変更されないレジスタ変数%i によりその領域を参照する。このメモリへのアクセスは、レジスタ変数の値の書き込みを行う store 命令と、レジスタ変数への値の読み込みを行う load 命令によって実現される。

[*1] 出力関数として printf ではなく printf1 という関数を使っているが、現在の実装では可変長引数関数に対応できていないため、引数の数を固定したラッパ関数を使っている。

リスト 1 変換前の C プログラム

```
1 void printf1(void *s, size_t x1);
2 int main (int argc, char *argv[]) {
3     long i;
4     for(i = 0; i < 10000000; i++) { }
5     printf1("%ld\n", i);
6     return 0;
7 }
```

最後に、LLVM には型システムが導入されており、各命令ごとに扱う値の型を正確に明示する必要がある。今回の例には記載されていないが、C 言語における値の型変換処理は、LLVM 上では専用の命令を用いて行われる。

4. 提案手法

本研究では、C 言語プログラムを Haskell のものに変換し、それから実行ファイルを生成することにより難読化を実現する。しかしこれら 2 つの言語は違いが大きく、例えば、C 言語では変数の値は自由に書き換えできるが、Haskell ではそれはできない。このギャップを埋めるために LLVM を利用する。LLVM では、値の更新ができないレジスタと可能なメモリが使い分けられているため、Haskell のようなプログラミングモデルを持つ言語と相性が良い。そこで、C 言語から一度 LLVM へ変換し、それから Haskell へ変換することにより、C 言語から Haskell への変換を実現する。変換処理全体の流れをまとめると、まず C 言語から LLVM への変換は、既存の Clang C コンパイラを利用することで実現できる。その次に行われるのは LLVM から Haskell への変換であり、本研究の貢献はこの手法の提案である。最後に、生成された Haskell プログラムから実行ファイルへの変換は、既存の GHC を利用することによって実現される。

LLVM から Haskell への変換は、各 LLVM 命令をそれと同等な処理を行う Haskell のプログラムに変換することにより、実現される。LLVM プログラムはアセンブリ言語のように逐次的に記述および解釈され、また、Haskell プログラムと既存の C 言語ライブラリ間での相互運用性を高めるために、変換後の Haskell プログラムは IO モナドと do 構文を利用した構造となる。

4.1 メモリモデル

ここでは、Haskell に変換されたプログラムと既存の C 言語ライブラリとの間で、メモリレイアウトの互換性を保つための方法について述べる。これを実現するためには、Haskell 内での C 言語データの表現方法と扱い方が重要となり、この点を説明する。これに関連して、C 言語のポインタを Haskell ではどのようにして表現するのかという点

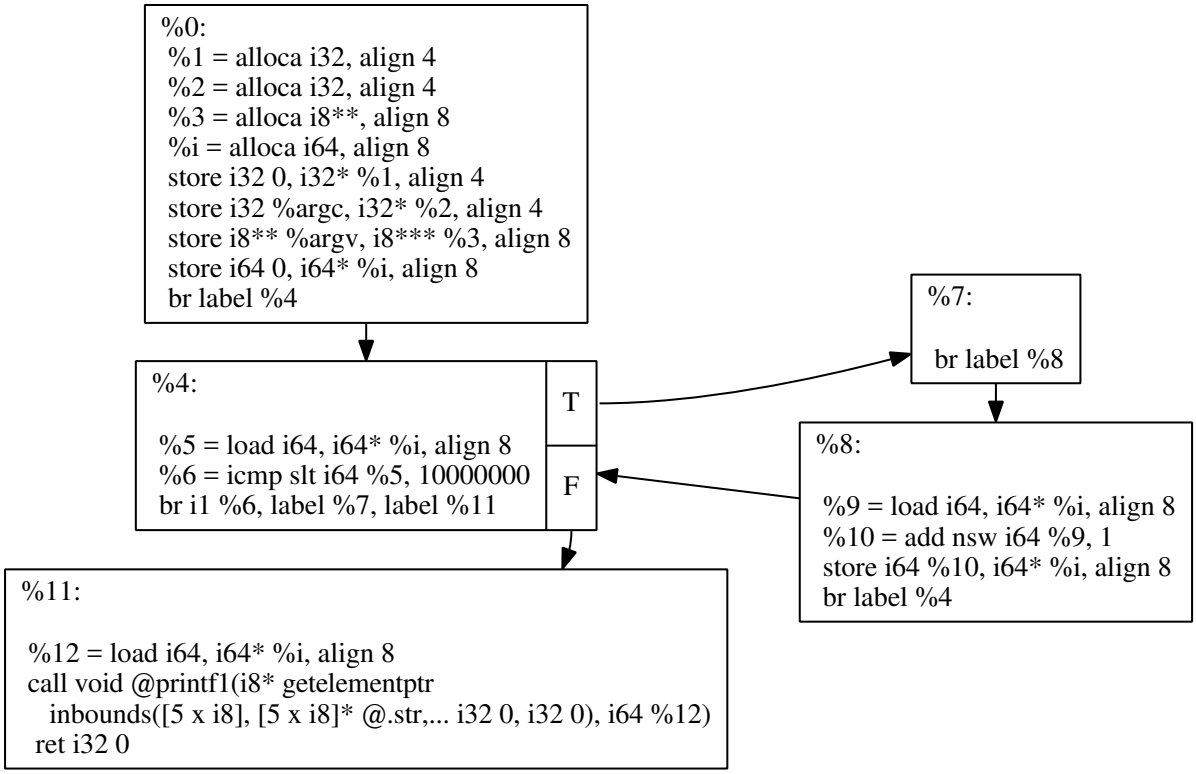

図 1 リスト 1 に対応する LLVM プログラムの制御フローグラフ

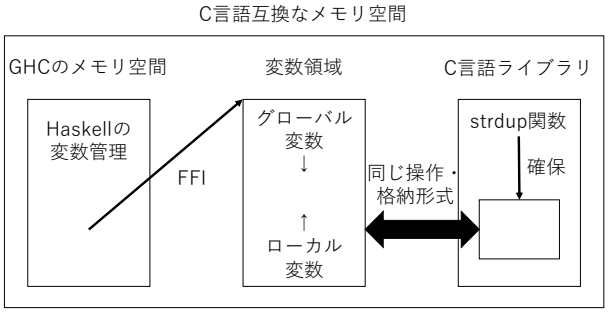

図 2 メモリモデル

についても述べる。

4.1.1 メモリの相互運用性

一般的に、C 言語を用いてプログラムを開発するときは、外部のライブラリを利用する必要がある。標準ライブラリにある関数を例に挙げると、例えば strdup 関数は引数として与えられた文字列を、この関数内で動的に確保された領域に複製し、それを指すポインタを返却する。この関数の呼び出し側は、返却された領域に対して操作を行う。また逆に、strcpy 関数も文字列をコピーする関数であるが、この関数の呼び出し側が用意した領域に strcpy 関数がコピー元の文字列を書き込む。このように C 言語を用いたプログラム開発では、ライブラリ内で割り当てられた動的領域を呼び出し側に利用させたり、逆にライブラリ関数の呼び出し側で用意した領域をライブラリ関数に操作させるといったことがよくある。そのため、このようにして記述された C 言語プログラムを Haskell へ変換した際には、Haskell 側が操作するメモリ領域と、外部のライブラリが操作するメモリ領域は相互に利用できなくてはならない。

そこで本変換手法では、図 2 に示すようなメモリモデルを利用する。図中の GHC ランタイムが管理する Haskell 変数用のメモリ領域とは別に、C 言語と互換性のあるメモリ領域を確保し、そこに変数用の領域の割り当てと操作を行うという方針を取る。この領域を以後「変数領域」と表記する。変数領域は C 言語と互換性があるのでライブラリ関数から操作することができ、逆にライブラリ関数内で動的に確保した領域についても、変数領域と互換性があるので特別扱いすることなく操作できる。変数領域は、GHC が提供する外部関数インタフェース（foreing function interface, FFI）を利用することにより、その確保と操作を実現できる。

このメモリモデルは、次に示すように LLVM のメモリモデルと似ているため、LLVM から Haskell への変換の簡単化に繋がる。まず、LLVM における値の割り当て方に着目すると、演算の対象となる値は書き換え不可能なレジスタ変数に割り当てられ、計算が行われる度にその結果は別の新しいレジスタ変数に割り当てられる（レジスタ変数は無

限に作れる)。値の更新が必要なものは、書き換え可能なメモリ上に領域が確保され、専用の命令(store 命令)を利用してこの領域に値を保存し、値を利用するときにここから読み取って(load 命令)利用する。そして、この領域への参照がレジスタ変数に保存され、これは上書きされない。次に Haskell の変数に着目すると、一度値を割り当てられた変数は、その値は書き換わらない。このことは、LLVM におけるレジスタ変数に対応しているとみなせる。また、LLVM におけるメモリは、先述した変数領域に対応し、その内部の特定の箇所への参照を Haskell の変数が保持するというように対応する。そして、LLVM の load, store 命令に相当する機能は、Haskell の FFI による変数領域へのアクセスとして実現される。

最後に個別の変数に割り当てる領域に関して、グローバル変数は変数領域の先頭から順番に確保される。また変数領域の末尾側はスタックとして利用され、末尾から先頭方向に向かって成長する。これを利用して、関数のローカル変数は、このスタック上に確保される。一方で、C 言語における malloc 関数などによる動的領域は、変数領域の外に確保されるが、先述したメモリの相互運用性により問題にはならない。そのため、ヒープ領域に該当する領域は存在しない。

4.1.2 ポインタ

まず、変数領域に確保された個別の変数を参照するためのポインタの表現方法について説明する。ポインタは、Haskell プログラムにおいては値として利用されるので、適切な型を決定する必要がある。ここで、C 言語における型 T に対応する Haskell の型を $\tau(T)$ と表記することにする。T は C 言語における int 型などのプリミティブ型に加え、ポインタ型も含まれる。C 言語における配列型はポインタを用いて表現でき、また構造体は、LLVM ではその構造体の領域の先頭ポインタとメンバのオフセットの組み合わせにより表されるので、ここでは考えない。

例として C 言語における int 型を T として考えると、int 型のポインタは Haskell では以下のような値として表現される。

$$\begin{aligned} &\lambda m.(castPtr\ (m\ +\ offset)\ ::\ Ptr\ \tau(int)) \\ &\qquad\qquad ::\ \ Ptr\ \tau(char) \to Ptr\ \tau(int) \end{aligned}$$

このように、C 言語におけるポインタは、Haskell ではラムダ式を用いて表現される[*2]。ここで、m は、変数領域の先頭を示す $Ptr\ \tau(char)$ 型の値(C 言語では void*型の値に相当)であり、*offset* は参照する領域の、変数領域の先頭からのオフセットを表す。変数領域の先頭を指す値をこのラムダ式に適用することにより、実際のアドレスを取得できる。このような設計にした理由は、変数領域は Haskell プログラムの実行開始時に確保され、その開始アドレスは実行ごとに異なるからである。オフセットについては、C 言語のグローバル変数に相当するものであれば、LLVM から Haskell への変換時に静的に計算できる。ローカル変数についても、関数ごとに必要な領域の総サイズとその中での各変数のオフセットは静的に計算できる。そのため、変換後の Haskell プログラムで関数呼び出しの度に、ローカル変数の確保を開始するアドレスを持ち回すことにより、上記の *offset* を適切に計算できる。

ポインタ自身を変数領域に保存する際には、整数に変換して保存する。しかし、GHC では Ptr a 型の値が保持するアドレスを整数に変換したりその逆の操作を行う関数は提供されていない。そこで、これらの処理を行う C 言語の関数を作成し、FFI で呼び出すことにより対応する。

次に、変数領域へのアクセス方法を示す。GHC が提供する以下の関数を用いて変数領域に対して読み書きを行う[*3]。

$$\begin{aligned} &peekArray\ \ ::\ \ Storable\ a \Rightarrow Int \to Ptr\ a \to IO\ [a] \\ &pokeArray\ \ ::\ \ Storable\ a \Rightarrow Ptr\ a \to [a] \to IO\ (\ ) \end{aligned}$$

peekArray 関数は LLVM の load 命令に対応し、読み取るバイト数と変数領域へのポインタを渡すことで、読み取り結果がリストにより返却される。また、pokeArray 関数は store 命令に対応し、ポインタが示す場所に与えられたデータを書き込む。

4.2 制御フロー

次に、LLVM の関数やそれを構成する制御フローを、Haskell での表現に変換する方法について述べる。まず、LLVM コードから Haskell コードへ変換する際に、両者の間で対応する点について述べる。次にこれを用いて、制御フローの変換方法を述べる。

4.2.1 変換の概要

図 1 に示された LLVM プログラムを例として、変換処理を説明する。このプログラムは、リスト 2 のような Haskell コードに変換される。Haskell への変換は、LLVM の関数単位で行われる。LLVM の関数(main)は Haskell の関数(v_main)に変換され、さらに、図 1 に示す基本ブロックがそれぞれ v_main 内でローカルに定義される関数に変換される。そして、図 1 の基本ブロック内の LLVM 命令ごとに、対応する Haskell プログラムが生成される。ここで、先述のメモリモデルの実現には IO モナドの利用が要求される。また、LLVM プログラムはアセンブリ言語のように手続き的に記述されているため、変換処理を簡単化するために、変換後の Haskell プログラムの構造は do 構文を利用

[*2] Foreign.Ptr モジュールの Ptr a は GHC が提供する型構成子で、GHC で FFI を利用する際に Haskell から C 言語の領域をポインタとして参照するために利用される。a のところに CInt(= $\tau(int)$) などのような、C 言語での int 型に対応する Haskell の型が入る。また、castPtr は上記の Ptr a 型の値を別の型へ変換するために用いられる。

[*3] 共に Foreign.Marshal.Array モジュールの関数である。

したものとなる。以後便宜上、LLVM の main 関数に対応する Haskell の v_main 関数のように、ソースコード中のトップレベルに現れ、他の関数から呼び出せる関数のことをグローバル関数と表記する。逆に、main 関数内の基本ブロックのように、v_main 関数内にローカルに定義される関数のことをローカル関数と表記する。

4.2.2 関数の変換方法

LLVM のグローバル関数は、let 式を用いた Haskell プログラムに変換される（1 行目）。let 部に、基本ブロックに対応するローカル関数（2〜38 行目）と、変数領域上に確保したローカル変数を参照するラムダ式（39〜47 行目）を宣言する。in 部には、このグローバル関数のエントリポイントとなるローカル関数の呼び出し式（48 行目）を記述する。これは、LLVM の関数で最初に実行される基本ブロック（図 1 中では%0 というラベルがある基本ブロック、リスト 2 では 2 行目のローカル関数）に相当している。

Haskell のグローバル関数は引数として、変換前の LLVM の関数の引数に加えて、2 つの引数を追加で受け取る。一つ目は変数領域の先頭を指すポインタ（引数 memory）である。これをローカル変数用のラムダ式に適用することで、実際に値が保存されている領域のアドレスを取得できる。二つ目は変数領域の先頭からのオフセット（引数 base）である。グローバル関数の呼び出しごとにこの値は小さくなり、関数呼び出しから復帰すると値は大きくなる。これにより、ローカル変数を保持するためのスタック構造を実現している。先述したように、変数領域の先頭を指すポインタ（memory）と、上記の base と、各ローカル変数の base からのオフセットにより、変数領域上に確保されているローカル変数の領域を特定をすることができる。

4.2.3 制御フローの変換方法

LLVM の基本ブロックは、ジャンプおよび分岐処理を行う br 命令や、関数から復帰する ret 命令といった、制御を他の基本ブロックへ移す命令により終端するという規則がある。そこで、このことを利用して LLVM の制御フローを Haskell で再現する方法を述べる。まず、br 命令により基本ブロックが終端されている場合、移動先のブロックも明示的に示されている。このため Haskell プログラムでは、移動先の基本ブロックに相当するローカル関数を末尾呼び出しすることにより、br 命令と同等な動作を再現できる。図 1 の例では、エッジの向きが制御の移動の方向を表しており、Haskell に変換後のプログラムでは、この方向でローカル関数を呼び出す。

次に、LLVM の ret 命令についてだが、この命令により返却される値を変換後のローカル関数の戻り値とする。ローカル関数同士の呼び出しは全て末尾呼び出しとなっているので、このようにすることで、最後のローカル関数の戻り値がグローバル関数としての戻り値となる。

リスト 2 変換後の Haskell プログラム

```
1  v_main memory base v_argc v_argv = let {
2    v_0x1cf4e70 arg = do {
3      pokeArray (v_0x1cf4f18 memory) [0];
4      pokeArray (v_0x1cf5598 memory) [v_argc];
5      pokeArray
6        (castPtr (v_0x1cf5638 memory) :: Ptr CLong)
7        [c_ptr2int
8          $ (castPtr (v_argv memory) :: Ptr CSChar)];
9      pokeArray (v_i memory) [0];
10     v_0x1cf5980 ();
11   };
12   v_0x1cf5980 arg = do {
13     v_0x1cf5ad8 <- peekArray 1 (v_i memory)
14       >>= (\ [x] -> return x);
15     let { v_0x1cf5b50 = v_0x1cf5ad8 < 10000000 };
16     if v_0x1cf5b50 then v_0x1cf5ba0 ()
17                    else v_0x1cf5c30 ();
18   };
19   v_0x1cf5ba0 arg = do {
20     v_0x1cf5d50 ();
21   };
22   v_0x1cf5d50 arg = do {
23     v_0x1cf5ea8 <- peekArray 1 (v_i memory)
24       >>= (\ [x] -> return x);
25     let { v_0x1cf5f20 = v_0x1cf5ea8 + 1 };
26     pokeArray (v_i memory) [v_0x1cf5f20];
27     v_0x1cf5980 ();
28   };
29   v_0x1cf5c30 arg = do {
30     v_0x1cf6068 <- peekArray 1 (v_i memory)
31       >>= (\ [x] -> return x);
32     let { v_0x1cf8ca8 = (\ m -> (v__str m)
33       `plusPtr` (1 * 0) :: Ptr CSChar) };
34     v_0x1cf66a8 <- v_printf1
35       memory (base -24)
36       v_0x1cf8ca8 v_0x1cf6068 ;
37     return 0
38   };
39   v_0x1cf4f18 = \ m ->
40     castPtr (plusPtr m $ base + (-4)) :: Ptr CInt;
41   v_0x1cf5598 = \ m ->
42     castPtr (plusPtr m $ base + (-8)) :: Ptr CInt;
43   v_0x1cf5638 = \ m ->
44     castPtr (plusPtr m $ base + (-16))
45                                       :: Ptr CSChar;
46   v_i = \ m ->
47     castPtr (plusPtr m $ base + (-24)) :: Ptr CLong;
48 } in do { v_0x1cf4e70 () }
```

4.3 LLVM 命令の変換方法

前節までで触れていない LLVM 命令のうち、主要なものについて説明する。まず、四則演算や比較演算などのように副作用が発生しない命令は、図 1 のレジスタ変数%6 へ比較結果の代入を行うコードを例にすると、これに相当す

る Haskell プログラムはリスト 2 の 15 行目となる。このように、let 構文を用いて演算結果をローカル関数内での局所的な変数に束縛し、以後の計算でこれを利用できるようにする。

次に、例のプログラムには現れていないが、LLVM には SSA 形式における Φ 関数を実現するための phi 命令という命令がある。Φ 関数は、その命令が属する基本ブロックに遷移してくる直前の基本ブロックに応じて、レジスタ変数に割り当てる値を切り替える命令である。例として、以下のような LLVM コードを考える。

```
1 B:
2     %r = phi i32 [%r0, %B0], [%r1, %B1]
```

この例では、phi 命令によってレジスタ変数%r に値を割り当てるが、基本ブロック B0 から基本ブロック B に遷移してきた場合はレジスタ変数%r0 の値を、基本ブロック B1 から遷移してきた場合は%r1 の値を割り当てる。このように phi 命令は、それが属する基本ブロックではない、別の基本ブロック内で割り当てられるレジスタ変数を参照することがある。このため、単純にレジスタ変数を Haskell の変数に変換するだけでは、スコープ外の変数の参照を行う不正なプログラムとなってしまう。そこで、遷移前のローカル関数が phi 命令で使用する値（この例では基本ブロック B0 に相当する関数では%r0 の値、B1 に相当する場合は%r1 の値）を引数として遷移先のローカル関数に渡すことにより解決する。この値を、各ローカル関数は arg という名前の仮引数で受け取る。そして phi 命令が現れた場合は、上記の四則演算命令の場合と同様に、let 構文を使って arg の値を新しい変数にそのまま割り当てる。

4.4 本手法で扱う LLVM アセンブリ言語

前章で LLVM アセンブリ言語の特徴について述べたが、LLVM アセンブリ言語には多くの言語機能があり、それら全てをサポートするには多くの労力が必要である。今回は、C 言語から Haskell への変換により生成されるプログラムは、単に C 言語から生成されたものよりも複雑な構造のため難読化として利用できるということを示せれば良く、そのために本稿では、最低限必要な LLVM の機能のみを取り扱うこととする。本来の LLVM アセンブリ言語に以下のような制限を加えたサブセット言語を定義し、これに対して提案手法を実装した。制限の内容は、LLVM の型システムにおいて扱える型の種類および関連する命令の種類の限定、そして、レジスタ変数のスコープの変更である。これらについて、以下で順に説明する。

4.4.1 型と命令の制限

LLVM には、厳密な型システムが存在する。例えば getelementptr という命令は、配列の要素や構造体のメンバのアドレスを厳密に計算する多機能な命令である。この命令は、オペランドの型によって処理内容が変化し、本手法で扱える型が多いほど、変換処理の実装が複雑になる。また、C 言語のキャストに相当する命令についても、型の種類が多いほど変換処理の組み合わせが増加してしまう。

そこで、本研究で使用する LLVM のサブセット言語では、扱える型の種類を制限する。例えば、本来の LLVM アセンブリ言語では、整数を任意のビット長ごとに別の型として扱うが、それを 8, 16, 32, 64 ビットの符号付き整数のみに制限する。実際の C 言語の処理系でも、整数はこれらのビット長単位で扱われるので、これらで十分事足りると考えられる。本手法で扱う型は、これらの整数型に加えて void 型、ポインタ型、配列型、構造体型のみとする。先述の通り、取り扱う型の種類を制限したため、それに伴ってサブセット言語で利用できる命令の種類も限定される。

4.4.2 レジスタ変数のスコープの変更

LLVM の仕様では、レジスタ変数のスコープはグローバル関数全体となっている。このため、ある基本ブロック内で値を割り当てられたレジスタ変数は、別の基本ブロック内からも参照可能であるが、これは Haskell プログラムへの変換処理においては不都合な仕様である。そこで、本提案手法で使用する LLVM のサブセット言語では、レジスタ変数のスコープを狭くすることでこの問題に対応する。具体的には、alloca 命令によって値を割り当てられる、ローカル変数用の領域を参照するレジスタ変数を除いて、各レジスタ変数のスコープはそれに値が割り当てられる基本ブロック内のみで完結させる。これにより、alloca 命令によって割り当てられる変数のみを特別扱いすれば良くなり、変換処理の実装が単純になる。

このことは Clang による LLVM プログラムの最適化処理にも関連している。Clang によって最適化されたプログラムは、上記の制限を満たしていないことがある。そのため本変換手法では、Clang が出力するプログラムのうち、最適化が行われていないもののみを変換対象として扱う。

4.5 変換後のプログラムの構成

先述したように、GHC によって生成された実行ファイルでは、実行コードのブロック同士が関数ポインタによって互いを参照し合う構造に変換される。変換後の main 関数を構成する LLVM コードについて、その内部での関数ポインタの参照関係を図 3 に示す。この図において、各ノードは実行コードに相当し、根に相当するノードは main 関数のエントリポイントである。また、エッジの方向が参照の方向である。

C 言語で記述された main 関数は 1 つの実行コードのみから構成されており、変換前はこのような関数ポインタによる参照関係は存在しなかった。しかし、提案手法の変換により複数の実行コードのブロックから構成されるようになった。このように、プログラムが実行コードの間接アク

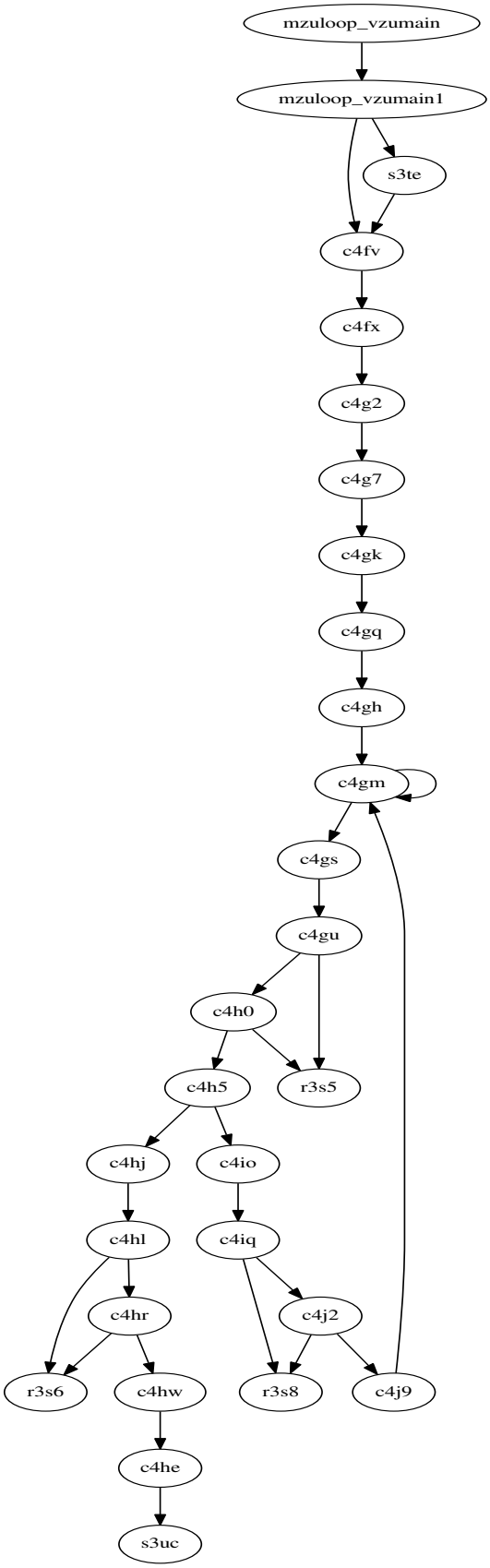

図 3 変換後のプログラム中の関数ポインタの参照関係

表 1 実験結果

種類	グラフ間の距離	変換前に対する変換後の実行時間
メモリアクセス	0.857	22.3 倍
竹内関数	0.954	33.6 倍
クイックソート	0.778	69.3 倍

セスによって構成されるため、変換後のプログラムの解析は複雑になると思われる。

5. 評価

提案手法による難読化の効果、および実行時の性能を測定する実験を行った。その実験内容と結果について述べる。実験環境は、OS には Linux 4.4(Ubuntu 16.04 LTS, x86_64) を利用し、コンパイラ基盤として LLVM 3.8 を、C 言語フロントエンドとして Clang 3.8 を利用した。Haskell 処理系としては、GHC 7.10 を利用した。

5.1 評価対象

リスト 1 のメモリアクセスを繰り返すプログラムに加え、竹内関数の計算、配列のクイックソートを行う、3 種類のプログラムを対象として実験を行った。竹内関数とは、以下の式で定義される関数である。

$$t(x,y,z) = \begin{cases} y & (x \leq y) \\ t(t(x-1,y,z), t(y-1,z,x), \\ \quad t(z-1,x,y)) & (otherwise) \end{cases}$$

この式の通りに再帰呼出しを用いて関数 t を実装し $t(12,0,11)$ を計算すると、関数呼び出しの回数は約 412 万回となる。これにより、関数呼び出しのオーバーヘッドを調べる。またクイックソートのプログラムでは、10 万要素の整数配列をソート対象とする。

5.2 難読化の評価

一般に、難読化によってプログラムの複雑さがどれだけ増したかは、そのプログラムの解析者の技量や主観に依るところが大きいため、厳密に定義することが難しい。そこで本実験では、マルウェアの亜種の特定などに応用される、グラフ理論に基づいて 2 つの実行ファイル間の違いを計算する手法 [4] を参考にして難読化の評価を行う。グラフ理論に基づいた手法では、まずグラフ間の距離を定義し、次にプログラムをグラフ構造とみなして距離を計算することにより、プログラムの違いを定量的に評価する。ここではグラフ間の距離を、2 グラフ間の最大共通部分グラフを利用して、以下の式のように定義する [2]。

$$dist(G_1, G_2) = 1 - \frac{|G_3|}{max(|G_1|, |G_2|)} \quad (1)$$

ここで、G_1, G_2 は比較対象となるグラフを、G_3 は G_1 と

G_2 の最大共通部分グラフを、$|G|$ はグラフ G のノード数を示している。この式の計算結果は 0 以上 1 以下の値となり、値が 1 に近いほどグラフ間の共通部分が少ないことを意味する。つまり、プログラムの構造は似ていないと解釈できる。

本実験では比較対象のプログラムとして、変換前の実行ファイルには Clang が生成するものを、変換後の実行ファイルには GHC が生成するものを利用した。この際、LLVM 命令は個別の CPU アーキテクチャ（本実験では x86_64）の命令とほぼ対応しているという点から、バイナリ形式のプログラムではなく、LLVM プログラムを利用して近似することにより評価を行った。また、式 (1) の計算に用いるグラフとして、変換前のプログラムのグラフ G_1 には LLVM プログラムの制御フローグラフを、変換後のプログラムのグラフ G_2 には LLVM プログラム中の関数ポインタによる参照関係を表したグラフを利用した。ここで、G_2 として関数ポインタの参照関係を利用している理由は、変換後プログラムは複数の関数から構成され、これらから正確な制御フローグラフを構成するのが困難であったためである。また、各関数の制御フローグラフは単純な形をしており、関数単体では難読化への貢献が少ないからである。そこで、各関数の参照関係に着目した、マクロな視点から見たグラフを用いて評価を行った。

各実験対象プログラムに対する評価結果を、表 1 の 2 列目に示す。変換前後のグラフ間の距離は 1 に近い値を示しており、先に述べたように、値が 1 に近いほどグラフ間の構造の差は大きい。そのため、プログラムの変換の結果、その制御構造が大きく変わったと考察できる。このことから、提案手法によって難読化として意味のある効果が得られたと言える。

5.3 実行性能の評価

本研究の難読化を行うことにより、変換前後のプログラムの実行性能がどのように変化するのかを調べた。3 つのプログラムについて、変換する前と後での実行時間を 5 回計測し、それらの平均値を求めた。この値を元に、実行時間が何倍に増加するのかを計算した。実験結果を表 1 の 3 列目に示す。

実験結果から、本研究の難読化を施すことによりプログラムの実行時間が、20〜70 倍増加していることが分かる。性能低下の大きな原因として、変数領域へのメモリアクセスが考えられる。Haskell プログラムではこの領域へアクセスするために peekArray, pokeArray 関数を用いており、これらは、メモリから読み込んだデータやメモリへ書き込む値を Haskell のリストとして表現している。これにより、無駄な変換処理が発生し、その分実行時間が増加するのだと考えられる。また、各 LLVM 命令の動作が Haskell のプログラムで表現されることにより、GHC のランタイム中ではこれらに関連するオブジェクトの生成やガーベジコレクションによる回収が多発していると思われる。このオーバーヘッドも、実行時間が増加する原因になると思われる。

Wang らの手法と比較すると、彼らの手法ではベンチマークに利用した各プログラムの全関数のうち 50%を Prolog に変換すると、実行時間が約 150 倍増加した。しかし、本手法ではプログラムを全て Haskell に変換しても Wang らの手法ほどは性能は低下しなかった。本実験では Wang らのものと同じプログラムを利用したわけではないので一概には言えないが、本提案手法を同じベンチマークを利用できるように拡張したとしてもこの傾向は変わらないと思われる。よって、実行性能については本提案手法によって改善されたと言える。

6. 結論

6.1 まとめ

本稿では、C 言語によって記述されたプログラムを LLVM プログラムに変換し、それを Haskell のものに変換することによって難読化を行う手法を提案した。この際、LLVM の各命令の動作に相当する Haskell プログラムを生成することで、LLVM から Haskell への変換を達成する。また、C 言語のライブラリ関数とのメモリモデルの相互運用性を保つために、Haskell の外部関数呼び出しの機能を利用して変数を管理する手法も提案した。

そして、本提案手法の評価を行った。難読化という観点においては、変換前後のプログラムの制御フローグラフの比較結果から、元のプログラムとの類似性が低くなることを確認した。また、C 言語の 1 つの関数が、Haskell では関数ポインタによって参照し合う複数の実行コードブロックの集合によって表現されることも確認した。この点は、難読化として有用であるといえる。また、Wang らの手法よりも実行性能の低下を抑えることができた。

6.2 今後の課題

本研究の変換手法では、LLVM アセンブリの言語機能のうち、一部しか対応できていない。例えば、Clang による最適化が行われた LLVM プログラムを適切に処理できないなどの問題点がある。そこで今後の課題として、まずは LLVM アセンブリ言語の全ての機能を適切に Haskell コードへ変換可能にする必要がある。

また、本手法によって生成される機械語プログラムの性能を向上させる必要がある。詳細な性能評価によりどの部分がボトルネックになっているのかを特定し、実行効率の良い Haskell プログラムへの変換やランタイムの改良を行うことにより、高い実行性能を持つ機械語プログラムを生成できるようにする必要がある。

参考文献

[1] Apple Inc. and others: clang: a C language family frontend for LLVM, (online), available from ⟨https://clang.llvm.org/⟩ (accessed 2017-06-30).
[2] Bunke, H. and Shearer, K.: A graph distance metric based on the maximal common subgraph, *Pattern recognition letters*, Vol. 19, No. 3, pp. 255–259 (1998).
[3] Coogan, K., Lu, G. and Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach, *Proceedings of the 18th ACM conference on Computer and communications security*, ACM, pp. 275–284 (2011).
[4] Gao, D., Reiter, M. K. and Song, D.: Binhunt: Automatically finding semantic differences in binary programs, *International Conference on Information and Communications Security*, Springer, pp. 238–255 (2008).
[5] Jones, S. L. P.: Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, *Journal of functional programming*, Vol. 2, No. 02, pp. 127–202 (1992).
[6] Lattner, C. and Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, IEEE Computer Society, p. 75 (2004).
[7] Marlow, S. et al.: Haskell 2010 language report (2011).
[8] Oreans Technology: Code Virtualizer: Total obfuscation against reverse engineering, (online), available from ⟨http://oreans.com/codevirtualizer.php⟩ (accessed 2017-06-30).
[9] Sharif, M., Lanzi, A., Giffin, J. and Lee, W.: Automatic reverse engineering of malware emulators, *Security and Privacy, 2009 30th IEEE Symposium on*, IEEE, pp. 94–109 (2009).
[10] Terei, D. A. and Chakravarty, M. M.: An LLVM backend for GHC, *ACM Sigplan Notices*, Vol. 45, No. 11, ACM, pp. 109–120 (2010).
[11] The Glasgow Haskell Team: Glasgow Haskell Compiler, (online), available from ⟨https://www.haskell.org/ghc/⟩ (accessed 2017-06-30).
[12] Wang, P., Wang, S., Ming, J., Jiang, Y. and Wu, D.: Translingual obfuscation, *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, IEEE, pp. 128–144 (2016).