電気学会論文誌 D（産業応用部門誌）
IEEJ Transactions on Industry Applications
Vol.139 No.2 pp.111–118　DOI: 10.1541/ieejias.139.111

論　文

難読化ツール oLLVM を用いたハードウェア難読化手法の評価

非会員　松岡　佑海*　非会員　藤枝　直輝**　正　員　市川　周一**a)

Evaluation of Hardware Obfuscation Techniques using Obfuscation Tool oLLVM

Yuumi Matsuoka*, Non-member, Naoki Fujieda**, Non-member, Shuichi Ichikawa**a), Member

（2018年3月15日受付，2018年9月16日再受付）

Obfuscation is a method to conceal the structure and function of software. Obfuscator-LLVM (oLLVM) is a set of tools for software obfuscation, which is implemented as middle-end passes in an LLVM framework. This study investigates the hardware obfuscation scheme, where the C-codes obfuscated by oLLVM are converted into logic design by using an HLS (High Level Synthesis) tool. CHStone benchmark suite is used for evaluation, where each application is processed by oLLVM, C-backend, and Xilinx Vivado HLS to generate the corresponding obfuscated logic design. On average, the logic scales of obfuscated designs became 1.1 times larger (by bogus control-flow), 1.7 times larger (by control-flow flattening), and 1.2 times larger (by instruction substitutions). In some applications, the obfuscations added by oLLVM were canceled by the analysis and optimization of HLS.

キーワード：知的財産，耐タンパ性，LLVM
Keywords: intellectual property, tamper resistance, LLVM

1. まえがき

近年，産業用システムの知的財産流出が大きな問題となっている。産業用システムに組み込まれた技術やノウハウは多大な時間と労力の成果であり，企業にとっては守るべき貴重な財産である。これらの財産が盗用されると，競合製品の出現など企業活動に大きな障害となる。特にソフトウェアは複製や解析が容易であるため，剽窃や改変などの被害に遭いやすい。この様な脅威からソフトウェアを保護することは重要な課題である。

ソフトウェア保護には，暗号化，鍵認証，難読化など多様な手法が存在する[1]。暗号化では暗号鍵を用いてプログラムやデータの内容を判別できなくする。安全性を確保するには，暗号鍵が盗まれないように管理する必要がある。それでも，暗号化された情報は利用時に復号されるため，盗用のリスクはゼロではない。鍵認証はロックを解除する「鍵」による保護機能である。パスワードやライセンスコードのチェックが鍵認証の典型例になる。鍵認証も，鍵の流出や，鍵認証をバイパスするようなコード改変には無力である。以下，本研究では，保護手法の1つである難読化を取り扱う。

難読化とは，プログラムや回路などの保護対象を機能的に等価かつ構造や動作が複雑になるように変換し，解析や改竄を阻害する技術である[1]。難読化手法として Opaque Predicates[2]や制御フロー平坦化[3]などがある。Opaque Predicates は恒真または恒偽な条件判定を行い，実行されない分岐を追加する。制御フロー平坦化は分岐を用いて制御構造を平らにする難読化である。難読化の問題点として，コードサイズや速度面のオーバヘッドが挙げられる。

ソフトウェア保護に際しては，コストと安全性の兼ね合いだけでなく，その実装に要する時間も重要である。安全性を向上するために開発期間が大幅に伸びると，開発コストが上昇するだけでなく，完成前にシステムが陳腐化する可能性がある。そのため実用的には自動でソフトウェアを難読化するツールが必須である。

本研究では，難読化ツール oLLVM[4]を採用する。oLLVM

a) Correspondence to: Shuichi Ichikawa. E-mail: ichikawa@ieee.org
* 豊橋技術科学大学大学院 電気・電子情報工学専攻
〒441-8580　愛知県豊橋市天伯町雲雀ヶ丘 1-1
Electrical and Electronic Information Engineering Course, Graduate School of Toyohashi University of Technology
1-1, Hibarigaoka, Tempaku-cho, Toyohashi, Aichi 441-8580, Japan
** 豊橋技術科学大学 電気・電子情報工学系
〒441-8580　愛知県豊橋市天伯町雲雀ヶ丘 1-1
Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology
1-1, Hibarigaoka, Tempaku-cho, Toyohashi, Aichi 441-8580, Japan

はオープンソースのプロジェクトで，プログラミング言語や実装プラットフォームによらず利用可能である。本研究ではC言語プログラムを評価対象とする。

本研究の目的の一つは，oLLVMを用いて難読化されたソフトウェアについて，サイズや実行時間のオーバヘッドを評価することである。Cプログラムを oLLVMで難読化すると，難読化された中間表現（Intermediate Representation, IR）が得られる。この中間表現からマシンコードを生成すれば，プロセッサで実行することができる。また中間表現をC言語に変換するバックエンドを用いることで，難読化されたC言語プログラムを得ることもできる。

難読化はソフトウェアだけでなくハードウェア（論理回路）にも適用可能な概念である[5]。本研究の第二の目的は，難読化されたC言語プログラムを高位合成（High Level Synthesis, HLS）を用いて論理回路化し，難読化ハードウェアにおける回路規模と遅延時間を評価することである。

本論文の構成は以下の通りである。2章で関連研究を示して本研究の位置づけを明らかにする。3章で oLLVMの機能を概観し，4章でソフトウェア難読化の評価結果，5章でハードウェア難読化の評価結果を示す。最後に6章で，結論と今後の課題を述べる。なお本論文は電気学会次世代産業システム研究会で発表した原稿[12]に大幅な加筆修正を施したものである。

2. 関連研究

著者らは制御プログラムの知的財産権を保護するため，制御プログラムの一部をハードウェア化して隠蔽する研究[6]，プログラムから難読化した論理回路を生成する研究[7]等を進めてきた。しかし適用対象ごとに難読化ツールを自作する必要があり，実用化へのハードルが高かった。汎用の（できればオープンソースの）難読化ツールが利用できれば，開発労力が削減され実用化への大きな一歩となる。

本研究では，ソフトウェア難読化ツールと高位合成技術を組合わせて，難読化論理回路が実現できるか検討する。2013年3月に oLLVMのリポジトリが GitHubに作成され，2015年に論文発表されるまで，一般に利用可能な難読化ツールは存在しなかったと思われる。高位合成技術も古くから研究されていたが，商用ツールが普及したのは2000年代半ばからである[8]。そのため，ソフトウェア難読化ツールと高位合成を実際に組み合わせて難読化制御回路を生成したという研究は，著者の知る限り本研究が初めてである。

難読化と高位合成はいずれも複雑な処理であり，組合せて正常に動作するかどうか自明ではない。仮に難読化論理回路が生成できたとしても，生成された回路が実用に耐えるかどうかも不明である。本研究では，生成された難読化論理回路の規模と速度も評価している。

改竄（tampering）に対する耐性を「耐タンパ性」とよぶ。難読化による耐タンパ性向上は重要な評価項目だが，これを定量的に評価することは容易ではない。Collberg ら[2]は複雑なプログラムほど重要な情報があると考え，プログラムの複雑さを定量化し，理解の困難さを評価した。しかし中村ら[10]は，プログラムの複雑さとプログラムの理解に必要な時間との間には，必ずしも相関がないことを示した。二村ら[11]はエントロピーによる命令出現頻度のランダム性の評価を行ったが，難読化によるエントロピーの増大は必ずしも起こらないと述べた。ソフトウェアの解析手段が多様であるため，耐タンパ性にも標準的あるいは単一の指標は存在せず，目的や手段毎に選択する必要がある。

本研究では，oLLVMを使用することによるオーバヘッドを測定・評価することを目的とする。既製の難読化ツール oLLVMを使用するため，耐タンパ性の定量的評価は今後の研究課題とし，本研究の評価対象外とする。

3. Obfuscator-LLVM

LLVM[9]は2000年にイリノイ大学で開発が開始されたコンパイラ基盤である。LLVMはフロントエンドである Clangや LLVM Core，LLDBなど様々なサブプロジェクトから構成されており，これらのサブプロジェクトを組み合わせる事によってコンパイラを作成することができる。LLVMでは，ソースコードは言語フロントエンドで中間表現（LLVM IR）に変換される。生成された IRは，“パス”と呼ばれるモジュールによって解析や最適化が行われる。このパスはユーザが任意に組み合わせることができ，独自のパスを実装することも可能である。

oLLVM[4]は Pascalらにより開発された難読化ツールである。oLLVMは LLVM IRに対して難読化を行うパスとして実装されているため，プログラム言語に依存せず，ターゲットアーキテクチャからもほぼ独立しているなどの利点がある。現在の oLLVMには，偽の制御フロー，制御フロー平坦化，命令置換の3種類の難読化が実装されている。

偽の制御フロー（Bogus Control Flow：B）は Collbergらが提案した Opaque Predicates[2]を拡張した難読化である。Opaque Predicatesは実行されない分岐を追加する事で，プログラムの複雑化を実現する。oLLVMの`-mllvm -bcf`オプションで，偽の制御フローを付加することができる。本論文でアプリケーション名の後ろに“_B”を付加した場合，偽の制御フローが付加されたアプリケーションを表すこととする。例えば ADPCMというアプリケーションに偽の制御フローを施したものは，ADPCM_Bと表記する。

制御フロー平坦化[3]（Control Flow Flattening：F）は制御構造を分岐により平坦化し，アルゴリズムの特徴を読み取りにくくする手法である。本文中では，アプリケーション名の後ろに“_F”を付加することにより，制御フロー平坦化が施されたアプリケーションを表す。制御フロー平坦化は oLLVMの`-mllvm -fla`オプションで利用可能である。

命令置換（Instructions Substitution：S）は，算術または論理演算子を機能的に等価な命令列に置き換えることである。本論文でアプリケーション名の後ろに“_S”を付加した場合，命令置換が適用されたアプリケーションを表すこととする。命令置換は oLLVMの`-mllvm -sub`オプションで

利用可能である。この様な難読化は比較的単純であり，生成されたコードを最適化する事によって容易に回避されうる。そのため，命令置換が難読化として機能するためには，最適化を施した後に置換を施す必要がある。与えられた演算子を無作為に等価な式に変換をすることは，結果としてコードの多様化をもたらす。コードの多様化により，プログラムを解析する際の命令パターン検索が難しくなる事も期待される。

4. ソフトウェア難読化

〈4·1〉 評価方法　4 章では，oLLVM を用いたソフトウェア難読化の結果を示す。まず，Fig. 1 にソフトウェア評価の手順を，Table 1 に使用した評価環境をまとめる†。

C プログラムを clang でコンパイルした実行ファイル（オブジェクト）を，以下 Original と称する。oLLVM を用いることにより，C プログラムを難読化して実行ファイルを生成することができる。難読化された実行ファイルは，oLLVM の難読化オプションに応じて On（難読化なし），Bn（偽の制御フロー），Fn（制御フロー平坦化），Sn（命令置換）と称する。

難読化された実行ファイルでは，難読化の結果がコンパイル後の命令列となり，難読化の状況を簡単に把握できない。そこで oLLVM と共に C バックエンド（CBE）を使用し，難読化された C プログラムを生成してから，それをコンパイルして難読化された実行ファイルを作成する。こうして生成された実行ファイルは，それぞれ難読化オプションに応じて O，B，F，S と呼ぶことにする。

CBE は LLVM IR を C 言語に変換するバックエンドツールで，旧バージョンの LLVM では正式サポートされていた。Version 3.1 以降の LLVM では CBE が正式サポートされていないため，本研究では JuliaComputing が作成した CBE [14] を使用する。CBE と oLLVM は同じ LLVM バージョンにする必要があるため，今回は oLLVM 3.5.2 に合わせて CBE 3.5.0 を採用した。CBE を利用することにより，oLLVM による難読化を目視で確認できる。また難読化したプログラムを容易に他の環境に移植できる。例えば多くの組込みシステムでは GCC を基盤とする開発システムが用いられており，LLVM 環境が提供されていないが，難読化された C プログラムであれば容易に移植可能である。

本研究では，評価対象としてベンチマークプログラム CHStone [13] を使用する。CHStone は原らによって作成された C ベースの高位合成用ベンチマークであり，様々な応用領域から選択された 12 のプログラムから構成されている（Table 2）。CHStone は標準の C 言語で記述されているため，ソフトウェアの評価基盤としても利用可能である。テストベクトルが自己完結しており，外部ライブラリが不要であるなど，本研究の評価で利用しやすい特徴を持つ。各プログラムの詳細については引用文献 (13) を参照されたい。

難読化はプログラムの一部分に施すことも可能だが，本研究では各プログラムの全体に対して難読化を施すこととする。実験における扱いを容易にするため，前処理として各プログラムを 1 つの C ファイルに結合してから用いた。

各アプリケーションに対して，Fig. 1 に示す 9 通り（Original，On，Bn，Fn，Sn，O，B，F，S）の実行ファイルを作成し，それぞれの実行時間とコードサイズを求めた。実行時間は各測定を 5 回行い，最大と最小の測定値を除いた 3 回の平均から求めた。コードサイズは，readelf コマンドで出力される `.text` セクションのサイズとする。

〈4·2〉 実行時間　CHStone の各アプリケーションについて，〈4·1〉節で示した 9 通りの変換を施し，その実行時間を測定した。本研究では難読化のオーバヘッドを調査したいので，絶対的な実行時間は問題でなく，Original の

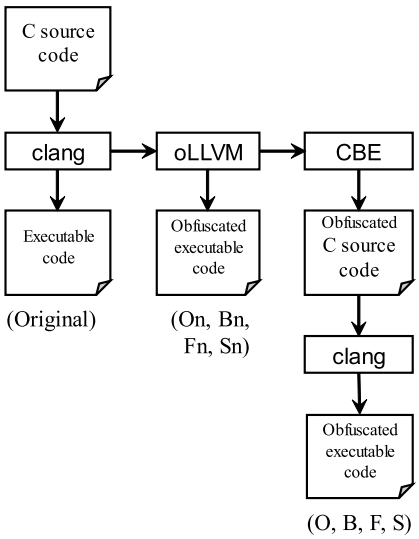


Fig. 1. Flowchart of software obfuscation.

Table 1. Evaluation environment.

OS	Ubuntu 16.04.2 LTS
CPU	Intel(R) Core(TM) i5-4440 @ 3.10 GHz
Memory	8GB
GCC	5.4.0
oLLVM clang [4]	3.5.0
llvm-cbe [14]	3.5.2
Optimization option	-O2

Table 2. CHStone programs [13].

Program	Design Description
DFADD	Double-precision floating-point addition
DFMUL	Double-precision floating-point multiplication
DFDIV	Double-precision floating-point division
DFSIN	Sine function for double-precision floating-point numbers
MIPS	Simplified MIPS processor
ADPCM	Adaptive differential pulse code modulation decoder and encoder
GSM	Linear predictive coding analysis of global system for mobile communications
JPEG	JPEG image decompression
MOTION	Motion vector decoding of the MPEG-2
AES	Advanced encryption standard
BLOWFISH	Data encryption standard
SHA	Secure hash algorithm

† clang が GCC のライブラリを利用しているため，GCC のバージョンも示した。C 言語プログラムのコンパイルには clang だけを用いる。

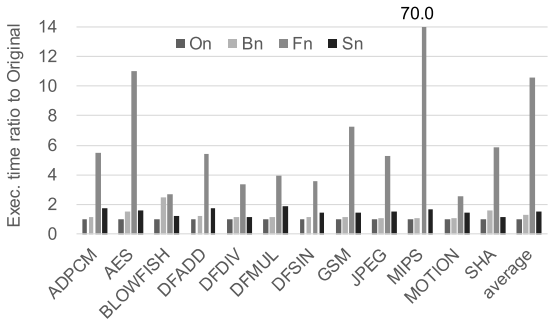


Fig. 2. Ratio of execution time to the original (On, Bn, Fn, Sn).

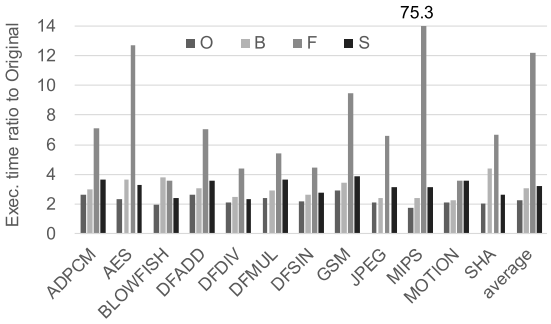


Fig. 3. Ratio of execution time to the original (O, B, F, S).

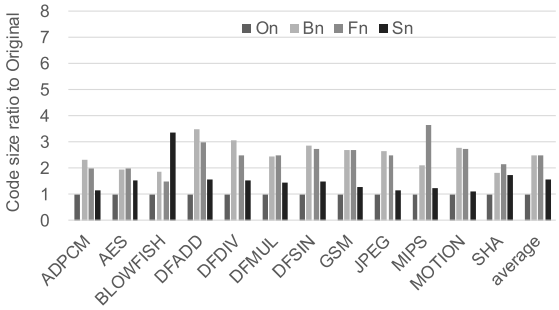


Fig. 4. Ratio of code size to the original (On, Bn, Fn, Sn).

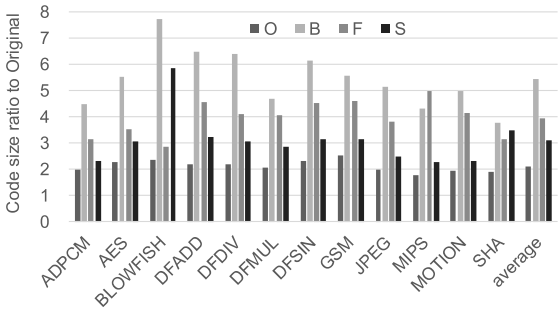


Fig. 5. Ratio of code size to the original (O, B, F, S).

実行時間に対する難読化プログラムの実行時間の比を検討する。CBE を用いない場合の結果を Fig. 2 に，CBE を用いた場合の結果を Fig. 3 にまとめる。12 本のプログラムのそれぞれの結果に加えて，右端に算術平均（average）を示した。

Fig. 2 において，On の結果は誤差の範囲内で 1.0 となっている。即ち oLLVM を使っても，難読化を指定しなければオーバヘッドは無視できる。従って，Fig. 2 における Bn，Fn，Sn の値は，難読化のオーバヘッドそのものと解釈できる。偽の制御フロー（Bn）を適用すると，実行時間は 1.1 倍（MIPS）～2.5 倍（BLOWFISH）となるが，平均では 1.3 倍程度に留まる。命令置換（Sn）では，1.2 倍（SHA）～1.9 倍（DFMUL），平均 1.5 倍程度である。ところが制御フロー平坦化（Fn）を施すと，実行時間は 2.6 倍（MOTION）～70 倍（MIPS），平均 10.5 倍と，非常に大きなオーバヘッドが発生する。

さらに CBE を使って実行ファイルを生成すると，オーバヘッドが増大する。Fig. 3 において，O の結果は 1.8 倍（MIPS）～2.9 倍（GSM）で，平均 2.3 倍となる。B と S のオーバヘッドは平均で 3 倍程度，F が 12 倍程度と，傾向は Fig. 2 に一致しているが，CBE のオーバヘッドが加わって更に実行時間が大きくなっている。ここでも MIPS_F のオーバヘッド（75 倍）は突出している。MIPS はプログラムの殆どが平坦化対象（switch 文）で記述されているため，平坦化により大きな影響を受けているものと思われる。同様に AES_F のオーバヘッド（13 倍）が大きいのは，switch 文を使用する関数 KeySchedule の実行時間が 25 倍程に増大するためである。

〈4･3〉 コードサイズ コードサイズの増大は，難読化によるプログラムの複雑化が原因であると考えられる。従って難読化においては避けがたいコストであるともいえるが，現実的制約は常に存在する。ユーザの立場では，各手法におけるコードサイズのオーバヘッドを事前に把握しておきたい。

Fig. 4 において，On の結果は誤差の範囲内で 1.0 となっている。即ち oLLVM を使っても，難読化を指定しなければコードサイズは変わらない。実行時間も変わらないので，On のコードは Original と（ほぼ）同じであると推測できる。偽の制御フロー（Bn）を適用すると，コードサイズは 1.8 倍（SHA）～3.5 倍（DFADD），平均では 2.5 倍になる。制御フロー平坦化（Fn）でも，1.5 倍（BLOWFISH）～3.7 倍（MIPS），平均 2.5 倍に留まる。最大は MIPS の 3.7 倍だが，実行時間の増大と同様に，平坦化対象（switch 文）の多用が原因であると考えられる。命令置換（Sn）では，1.1 倍（MOTION）～3.4 倍（BLOWFISH），平均 1.6 倍程度である。

CBE を使うとコードサイズが増えることも，実行時間と同じ傾向である。Fig. 5 において，O の結果は 1.8 倍（MIPS）～2.5 倍（GSM）で，平均 2.1 倍である。これが CBE により生成されたコードの増分であると解釈できる。命令置換（S）のオーバヘッドが，他の手法（B と F）より小さいことは，Fig. 4 と変わらない。BLOWFISH_S の 5.9 倍はやや突出しているが，ADD と XOR を多用している関数 BF_encrypt の影響と思われる。制御フロー平坦化（F）では，2.9 倍（BLOWFISH）～5.0 倍（MIPS），平均 4.0 倍と，これまで

の結果と矛盾しない。偽の制御フロー（B）ではオーバヘッドが大きくなり，3.8 倍（SHA）～7.8 倍（BLOWFISH），平均 5.4 倍となる。BLOWFISH_B のコードを調べると，関数 BF_set_key に for 文と if 文が多用されており，偽の制御フローによる影響が大きく出ていた。

5. ハードウェア難読化

〈5･1〉 評価方法 5 章では，oLLVM を用いたハードウェア難読化の方法と評価結果を示す。

ハードウェア難読化の手順を Fig. 6 にまとめる。C 言語ソースから高位合成（HLS）ツールで生成した論理回路を，本章では Original と称する。さらに 4 章と同様に，clang，oLLVM，CBE を使用して難読化した C 言語ソースを生成し，それを HLS で処理して難読化された論理回路を生成した。難読化後の論理回路は，oLLVM の難読化オプションに応じて以下 O，B，F，S と呼ぶ。HLS ツールには Xilinx Vivado HLS 2017.2 を使用し，ターゲットデバイスには Xilinx Zynq-7000（xc7z020clg484-1）を指定した。クロック制約は試行結果に基づいて 100 MHz と設定し，CAD のオプションは全てデフォルトのままとした。

4 章の結果から，CBE を用いると難読化のオーバヘッドが増加することが予想される。oLLVM の出力（LLVM IR 形式）から直接 HLS で論理回路を生成できれば理想的である（Fig. 6 の破線）。しかし Vivado HLS では LLVM IR を直接処理する方法がない（公開されていない）ため，本研究では CBE を経由する方法をとった。他の HLS ツール，例えば LegUp[15] は LLVM IR から Verilog HDL を生成できるため，破線の経路で処理できる可能性がある。しかし本研究では，著者らが Xilinx のデバイスとツールに習熟していること，メーカーサポートがあることから，Vivado HLS を評価に使用した。オープンソースの LegUp は研究上興味深い HLS ツールではあるが，破線の経路による評価は今後の課題とする。

評価対象には，4 章と同じ CHStone[13] を用いる。CHStone は元々 C ベース高位合成のためのベンチマークなので，本研究の評価対象として最適である。CHStone は YXI 社の高位合成ツール eXCite で動作が確認されているが，Vivado HLS では GSM，JPEG，MOTION について合成に失敗する。これは，Vivado HLS でポインタのサポートに制約があるためである。そこで本研究では GSM，JPEG，MOTION に限って Vivado HLS で合成できるようソースコードを修正した。具体的な修正点[16] は紙面の都合で省略するが，Vivado HLS と CHStone の組合せに関しては先行研究[17] でも問題が指摘されている。

以下本章では，合成された各論理回路について，論理規模と実行時間を評価する。論理規模は CAD の出力する使用資源量で見積もることができる。また CHStone ではテストベクタがソースコードに含まれているため[13]，実行時間は一意に定まる。従って，サイクルアキュレートな論理シミュレータで計測すれば，実行サイクル数は求まる筈である。しかし各プログラムの実行サイクル数は大きく，それを 12 個のプログラム，5 種の論理回路（Original，O，B，F，S）の全ての組合せについて計測することは現実的に困難である。そこで本研究では，Vivado HLS の出力する見積サイクル数を実行時間の指標とする。

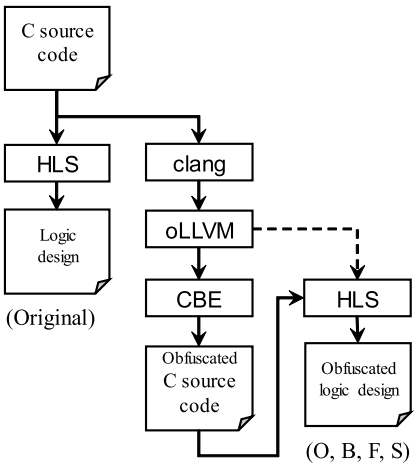


Fig. 6. Flowchart of hardware obfuscation.

Vivado HLS は合成時に回路の最大・最小レイテンシ（クロック数）を見積って出力する。ただしループ回数が可変である場合，レイテンシ不明として「?」を出力する。本研究でも多くの場合にレイテンシ不明となったが，CHStone ではループ回数は一意に決まっているので，ソフトウェアによる試行でカウントした各ループの実行回数を `pragma HLS loop_tripcount` でソース内に指定することにより，最大・最小レイテンシの出力を得た。

〈5･2〉 回路規模 Table 3 に，12 応用 × 5 種の合成結果をまとめる。応用名の後の O, B, F, S は難読化手法，無印は Original を表している。回路規模は回路要素全て（LUT, FF, DSP, BRAM）で決まるが，ここでは紙数の関係上 LUT についてだけ結果を検討する。

Fig. 7 は，LUT 数の比を棒グラフで示したものである。全体として平坦化（F）では論理規模が増大することがわかる。幾つかの応用（たとえば ADPCM）では，B や S で論理規模が増大しない（O とほぼ同じ）。

ADPCM について HLS の出力する VHDL 記述を調べたところ，ADPCM_B と ADPCM_S は信号名を除き Original とほぼ同じであった。そのため ADPCM_O，ADPCM_B，ADPCM_S の論理規模が同等になったものと思われる。しかし ADPCM_B，ADPCM_S の C 言語ソースでは確かに難読化が施されており，ソフトウェア評価では実行時間もコードサイズも増加している。すなわち HLS の解析と最適化により難読化（B と S）が除去されたと考えられる。念のため 50 行ほどの小さなプログラムで追試したが，そこでも B と S が無効化される現象が観測された。

AES や MOTION では，O の回路規模が Original より小さくなっている。これは CBE が変数に _attribute_ を与えて，最適化のヒントを与えているためと推測される。例えば C

Table 3. Summary of Obfuscated hardware.

Design	LUT	FF	DSP 48	BRAM 18K	cycle time [ns]	min. latency [cycle]	max. latency [cycle]
ADPCM	11542	25729	164	14	8.64	2.14E+04	3.25E+04
ADPCM_O	10473	20729	136	12	8.70	2.00E+04	2.96E+04
ADPCM_B	10473	20729	136	12	8.70	2.00E+04	2.96E+04
ADPCM_F	14413	28948	117	13	8.70	1.97E+03	2.64E+07
ADPCM_S	10746	21613	135	13	8.70	2.13E+04	3.09E+04
AES	9955	7119	8	14	8.60	2.57E+03	5.66E+03
AES_O	6832	4629	0	13	8.60	3.24E+03	3.32E+03
AES_B	7618	4738	0	12	8.60	5.12E+03	5.44E+03
AES_F	19573	16243	24	13	8.59	4.03E+03	2.58E+05
AES_S	7145	4693	0	13	8.60	3.24E+03	3.32E+03
BLOWFISH	14694	15928	0	19	8.58	7.63E+04	3.68E+05
BLOWFISH_O	14745	16032	0	19	8.58	6.12E+04	3.79E+05
BLOWFISH_B	14640	15906	0	19	8.58	6.12E+04	3.79E+05
BLOWFISH_F	18159	22697	0	19	8.51	7.49E+04	1.07E+10
BLOWFISH_S	35435	37356	0	19	8.58	6.25E+04	3.94E+05
DFADD	8591	8686	0	7	8.75	3.23E+02	1.01E+03
DFADD_O	8591	8686	0	7	8.75	3.23E+02	1.01E+03
DFADD_B	8714	8973	0	7	8.75	3.23E+02	1.01E+03
DFADD_F	11016	10748	0	7	8.54	2.81E+02	3.17E+05
DFADD_S	10298	8844	0	7	8.75	3.23E+02	1.06E+03
DFDIV	7254	10263	24	7	8.75	1.33E+02	3.76E+03
DFDIV_O	7254	10263	24	7	8.75	1.33E+02	3.85E+03
DFDIV_B	7938	11622	24	7	8.75	1.33E+02	3.83E+03
DFDIV_F	9208	12497	32	7	8.54	5.00E+00	3.99E+05
DFDIV_S	9051	10750	24	7	8.75	1.33E+02	3.85E+03
DFMUL	4291	5360	16	7	8.52	1.21E+02	5.21E+02
DFMUL_O	4275	5165	16	7	8.52	1.21E+02	5.21E+02
DFMUL_B	4223	5165	16	7	8.52	1.21E+02	5.21E+02
DFMUL_F	5642	6622	16	7	8.64	1.25E+02	1.71E+04
DFMUL_S	5046	4925	16	7	8.75	1.21E+02	5.21E+02
DFSIN	20703	25414	44	8	8.75	1.08E+03	9.42E+04
DFSIN_O	20687	25219	44	8	8.75	1.08E+03	9.60E+04
DFSIN_B	20467	25296	44	8	8.75	1.08E+03	9.60E+04
DFSIN_F	27898	30823	52	8	8.66	2.21E+02	2.94E+07
DFSIN_S	24055	25396	44	8	8.75	1.12E+03	9.64E+04
GSM	6455	10729	49	12	11.36	2.61E+03	4.04E+03
GSM_O	6459	10729	49	12	11.36	2.61E+03	3.69E+03
GSM_B	7036	12496	49	12	11.36	2.61E+03	4.08E+03
GSM_F	11399	22226	48	9	9.70	1.98E+03	6.64E+07
GSM_S	6535	10925	49	12	11.36	2.61E+03	4.94E+03
JPEG	34840	50504	154	52	9.22	4.27E+04	1.71E+06
JPEG_O	35078	51312	154	48	9.59	4.26E+04	3.69E+08
JPEG_B	37166	53650	154	46	9.59	4.41E+04	3.26E+10
JPEG_F	52496	79054	155	48	9.81	1.59E+05	6.07E+14
JPEG_S	36332	52587	154	46	9.59	4.27E+04	7.92E+15
MIPS	1956	2458	8	4	8.69	2.63E+03	3.85E+03
MIPS_O	1928	2314	8	4	8.69	2.64E+03	3.86E+03
MIPS_B	4895	2781	8	3	8.69	3.35E+03	6.40E+03
MIPS_F	4998	2938	8	4	8.69	1.68E+04	3.36E+04
MIPS_S	2086	2426	8	4	8.69	2.64E+03	3.86E+03
MOTION	4555	6235	0	5	10.01	9.60E+01	1.64E+05
MOTION_O	3997	4784	0	5	10.19	9.60E+01	1.64E+05
MOTION_B	4018	4779	0	5	10.19	9.60E+01	1.64E+05
MOTION_F	12488	13352	0	8	10.19	2.25E+02	6.49E+10
MOTION_S	4354	5087	0	5	10.19	9.60E+01	1.64E+05
SHA	3246	4072	0	20	8.63	1.11E+05	1.12E+05
SHA_O	3246	4072	0	20	8.63	1.03E+05	1.12E+05
SHA_B	3246	4072	0	20	8.63	1.03E+05	1.12E+05
SHA_F	6502	7439	0	22	8.75	1.05E+03	3.24E+06
SHA_S	3759	4439	0	20	8.75	1.03E+05	1.12E+05

言語の変数がハードウェアの必要幅より大きい場合，属性があると最適なサイズに削減できることがある。

BLOWFISH_S の論理規模も大きいが，これはソフトウェア評価のコードサイズでも見られる傾向で，ハードウェア固有の事情ではない。原因を調べると，BLOWFISH の S-box 演算で多くの演算子が置換され，一時変数への代入文の数は 2 倍ほどに増大していた。BLOWFISH の制御構造は，演算子置換による難読化に特に適合しているといえる。

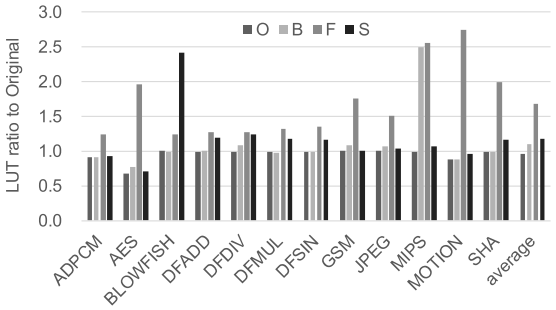


Fig. 7. Ratio of LUT to the original.

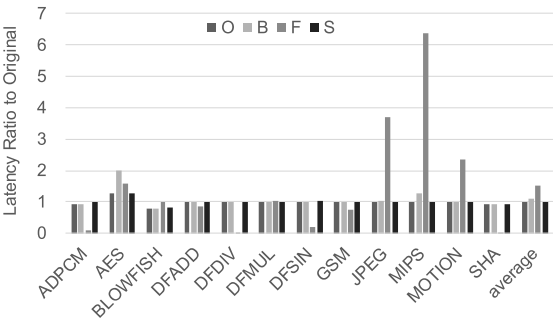


Fig. 8. Ratio of minimum latency to the original.

MIPS_B の LUT 数も際立って大きい。調べてみると，レジスタやメモリへのアクセスが MIPS_O では 32 ビット単位になっているのに対し，MIPS_B では 8 ビット単位になっていた。そのため MIPS_B では論理が複雑になり LUT 数が増加したものと思われる。メモリアクセス幅の変更が，偽の制御フローで難読化するために本質的に必要な処理であるか自明ではない。従ってこの論理規模増大は，実装上の問題であって本質的な変化ではないかもしれない。

〈5・3〉 実行時間 Table 3 に，12 応用 × 5 種の実行サイクル数の見積値（最小，最大）をまとめる。Fig. 8 は，実行サイクル数（最小）を Original との比で示したものである。幾つかの応用（ADPCM，SHA 等）では，平坦化（F）後の最小サイクル数が Original より小さい。これは平坦化手法の直接的帰結である。平坦化されたプログラムでは一つの処理を複数回に分けて実行するため，各回の処理は簡単になるかわり実行サイクル数が増加する。しかし HLS の最小遅延見積では，サイクル数の増加が正しく反映されていないと思われる。

一方，HLS の見積もる最大実行サイクル数は桁違いに大きく，現実離れした値になる（Table 3）。Fig. 9 は，実行サイクル数（最大）を Original との比で示したものである。値の桁が大きく違うため，このグラフだけ縦軸を対数表示にしている。Fig. 9 で平坦化後のサイクル数が大きいのは，最小値の見積と逆に，サイクル数の増加が見積値に過大に反映されているためと考えられる。このように HLS の見

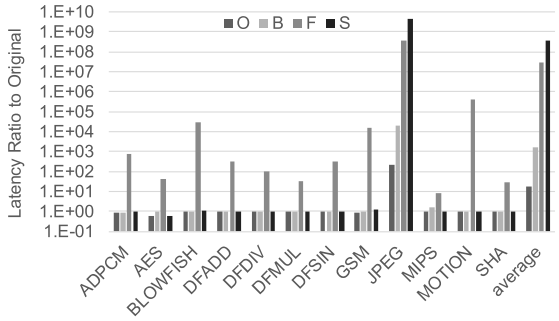


Fig. 9. Ratio of maximum latency to the original.

積が不正確になるということは，ある意味で難読化が成功しているという解釈もできる。

6. むすび

本研究では oLLVM を用いて CHStone ベンチマークに 3 種類の難読化を施し，生成されたソフトウェアとハードウェアのオーバヘッドを評価した。

ソフトウェア難読化の評価では，oLLVM 自体のオーバヘッドは無視でき，難読化による本質的オーバヘッドが現れた。実行時間は，偽の制御フロー（B）で平均 1.3 倍，平坦化（F）で平均 10.5 倍，命令置換（S）で平均 1.5 倍となった。コードサイズは，それぞれ平均で 2.5 倍（B），2.5 倍（F），1.6 倍（S）となった。ソフトウェア難読化で CBE を経由した場合，実行時間とコードサイズに相当なオーバヘッドが生じることも分かった。難読化しない場合（O）でも，CBE を経由するだけで実行時間とコードサイズはそれぞれ約 2 倍に増加する。CBE 経由で難読化した場合，実行時間は平均 3.0 倍（B），12.2 倍（F），3.2 倍（S）となり，コードサイズは平均 5.4 倍（B），4.0 倍（F），3.1 倍（S）になる。

ハードウェア難読化の評価では，論理規模（LUT）は平均 1.1 倍（B），1.7 倍（F），1.2 倍（S）であったが，一部の応用について難読化 B や S が HLS によって無効化されることが観測された。HLS による実行サイクル数の見積値は，難読化により不正確になることが確認できた。

今後は，HLS による最適化を回避する難読化手法，複数の難読化手法を組合せる手法，などを検討し，より実用的な難読化ハードウェア生成手法を検討する必要がある。本研究ではソフトウェア難読化ツール oLLVM をハードウェア難読化に流用したが，ハードウェア難読化に固有の技術を実装・評価することも今後の課題である。

難読化ハードウェアの性能オーバヘッドについては，さらに精密で実用的な評価が望まれる。最小サイクル時間を合成結果から見積り，実行サイクル数を論理シミュレーションにより測定すれば，各応用の実行時間を見積もることできるが，応用と難読化手法の組合せが多くシミュレーション時間も長くなって現実的ではない。むしろ FPGA を用いて難読化回路を実装し，実ハードウェアで性能を評価することが重要である。今後は実システムによる評価を進めてゆきたい。

謝辞

本研究の一部は JSPS 科研費 16K00072 の支援による。

文献

（1） A. Monden and C. Thomborson: “Recent Software Protection Techniques—Software-only Tamper Prevention—”, IPSJ magazine, Vol.46, No.4, pp.431–437 (2005) (in Japanese)
門田暁人・Clark Thomborson：「ソフトウェアプロテクションの技術動向（前編）—ソフトウェア単体での耐タンパー化技術—」, 情報処理, Vol.46, No.4, pp.431–437 (2005)

（2） C. Collberg, C. Thomborson, and D. Low: “A Taxonomy of Obfuscating Transformations”, Department of Computer Science, The University of Auckland, Technical Report No.148 (1997)

（3） T. László and Á.Kiss: “Obfuscating C++ Programs via Control Flow Flattening”, Annales Universitatis Scientiarum Budapestinensis de Rolando Etvs Nominatae. Sectio Computatorica, Vol.30, pp.3–19, Hungary (2009)

（4） P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin: “Obfuscator-LLVM-Software Protection for the Masses”, Proc. International Workshop on Software Protection (SPRO 2015), pp.3–9 (2015)

（5） D. Forte, S. Bhunia, M.M. Tehranipoor (Eds.): Hardware Protection through Obfuscation, Springer International Publishing (2017)

（6） N. Fujieda, S. Ichikawa, Y. Ishigaki, and T. Tanaka: “Evaluation of the hardwired sequence control system generated by high-level synthesis”, Proc. 26th IEEE International Symposium on Industrial Electronics (ISIE 2017), pp.1261–1267 (2017)

（7） Y. Ishigaki, N. Fujieda, Y. Matsuoka, K. Uyama, and S. Ichikawa: “An Obfuscated Hardwired Sequence Control System Generated by High Level Synthesis”, Proc. Fifth International Symposium on Computing and Networking (CANDAR 2017) (2017)

（8） G. Martin and G. Smith: “High-Level Synthesis: Past, Present, and Future”, IEEE Design & Test of Computers, Vol.26, No.4, pp.18–25 (2009)

（9） C. Lattner and V. Adve: “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”, Proc. 2004 International Symposium on Code Generation and Optimization (CGO 2004), pp.75–86 (2004)

（10） M. Nakamura, A. Monden, T. Itoh, K. Matsumoto, Y. Kanzaki, and H. Satoh: “Queue-based cost evaluation of mental simulation process in program comprehension”, Proc. International Software Metrics Symposium, pp.351–360 (2003)

（11） A. Futamura, A. Monden, H. Tamada, Y. Kanzaki, M. Nakamura, and K. Matsumoto: “Evaluation of Software Obfuscation Based on the Randomness of Instructions”, Computer Software, Vol.30, No.3, pp.18–24 (2013) (in Japanese)
二村阿美・門田暁人・玉田春昭・神崎雄一郎・中村匡秀・松本健一：「命令のランダム性に基づくプログラム難読化の評価」, コンピュータソフトウェア, Vol.30, No.3, pp.18–24 (2013)

（12） Y. Matsuoka, N. Fujieda, and S. Ichikawa: “Evaluation of software obfuscation techniques by obfuscation tool oLLVM”, The papers of technical meeting on Innovative Industrial System, IEE Japan, IIS-18-002 (2018) (in Japanese)
松岡佑海・藤枝直輝・市川周一：「難読化ツール oLLVM を用いたソフトウェア難読化手法の評価」, 電気学会次世代産業システム研究会, IIS-18-002 (2018)

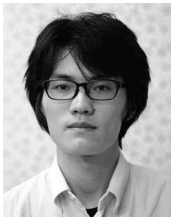
（13） Y. Hara, H. Tomiyama, S. Honda, and H. Takada: “Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis”, IPSJ Journal, Vol.17, pp.242–254 (2009)

（14） JuliaComputing: “llvm-cbe-fork (LLVM 3.5), github.com”,https://github.com/JuliaComputing/llvm-cbe-fork (accessed 2017-07-25)

（15） A. Canis, et al.: “LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems”, ACM Trans. Embedded Computing Systems, Vol.13, No.2, Article 24 (2013)

（16） Y. Matsuoka: “Preliminary evaluation of hardware obfuscation with LLVM and high level synthesis tool”, Master’s thesis, Department of Electrical and Electronic Information Engineering, Toyohashi University of Technology (2017) (in Japanese)
松岡佑海：「LLVM と高位合成ツールによるハードウェア難読化の試験評価」, 修士論文, 豊橋技術科学大学大学院電気・電子情報工学専攻 (2017)

（17） A. Dubey, A. Mishra, and S. Bhutada: “Comparative Study of CHStone Benchmarks on Xilinx Vivado High Level Synthesis Tool”, International Journal of Engineering Research & Technology, Vol.4, No.1, pp.237–242 (2015)



松 岡 佑 海（非会員） 2014 年和歌山工業高等専門学校電気情報工学科卒業。2016 年豊橋技術科学大学電気・電子情報工学課程卒業。2018 年同大学大学院工学研究科電気・電子情報工学専攻修士課程修了。電子情報通信学会会員。

藤 枝 直 輝（非会員） 2013 年東京工業大学大学院情報理工学研究科計算工学専攻博士後期課程修了。博士（工学）。同年より豊橋技術科学大学電気・電子情報工学系助教。プロセッサアーキテクチャ，FPGA 応用，組み込みシステム，セキュアプロセッサの研究に従事。情報処理学会，電子情報通信学会，IEEE 各会員。

市 川 周 一（正員） 1985 年東京大学理学部卒業。1987 年同大学大学院理学系研究科修士課程修了。1987 年新技術事業団創造科学推進事業（ERATO）後藤磁束量子情報プロジェクト研究員。1991 年三菱電機（株）LSI 研究所，システム LSI 開発研究所勤務。1994 年名古屋大学工学部助手。1997 年豊橋技術科学大学工学部講師。同助教授，准教授を経て，2010 年豊橋技術科学大学大学院工学系研究科准教授。2011 年沼津工業高等専門学校制御情報工学科教授。2012 年より，豊橋技術科学大学大学院工学系研究科教授。現在に至る。理学博士。並列計算機，並列処理，および専用計算システムアーキテクチャの研究に従事。IEEE（senior member），電子情報通信学会（シニア会員），ACM，情報処理学会，各会員。