コンパイラ最適化がテイント解析に与える影響と解析性能向上

林 優香[1,a)]　稲吉 弘樹[1]　掛井 将平[1]　毛利 公一[2]　瀧本 栄二[2]　齋藤 彰一[1]

概要：個人情報やプライバシに関わる機密データを電子化して扱う機会が増え，その漏洩の危険も高まっている．機密データの漏洩の有無を確認する方法の 1 つにテイント解析がある．テイント解析では，データが変数間を伝播する道筋を追跡することで漏洩の判定を行う．一方で，ソースファイルから実行ファイルを生成する際には，プログラム実行時間の最小化，使用するメモリ量の最小化等を目的としたコンパイラ最適化を施すことが一般的である．最適化手法によっては変数の除去や定数化のような処理が含まれる場合があり，変数を辿るテイント解析に影響する可能性が考えられる．本稿では，テイント解析とコンパイラ最適化の関係性について調査を行い，解析に影響を与える最適化を明らかにした．また，コンパイラ最適化をテイント解析に活用する手法を提案し，一部の最適化手法に対して実際に提案手法の実装と評価を行った．

キーワード：テイント解析，コンパイラ最適化，LLVM，暗黙的フロー

Effects of Compiler Optimization on Taint Analysis and its Performance Enhancement

Yuka Hayashi[1,a)]　Hiroki Inayoshi[1]　Shohei Kakei[1]　Koichi Mouri[2]　Eiji Takimoto[2]
Shoichi Saito[1]

Abstract: Increasing computerization of privacy sensitive or confidential information has amplified risks of information leakage in recent years. In the circumstances, taint analysis is used to track information propagating among variables and detect information leakage. Meanwhile, in compilation process of a program, compiler optimization is commonly used to minimize processing times and the amount of memory used. However, some of compiler optimization techniques remove variables in the original source code and replace it with immediate values that can cause implicit information flows and make taint analysis impracticable. In this paper, we investigate and declare the effects of compiler optimization on taint analysis. Furthermore, we propose a method that leverages compiler optimization to enhance taint analysis. We implemented and evaluated it on programs containing implicit information flows.

Keywords: Taint Analysis, Compiler Optimization, LLVM, Implicit Flow

1. はじめに

個人情報やプライバシに関わる機密データを電子化して扱う機会が増え，その漏洩の危険も高まっている．機密データを扱うプログラムは，情報漏洩に対する安全性の保証が期待される．しかし，プログラム作成者が自身のプログラムの安全性を保証したり，ユーザがその保証の正しさを判定することは難しい [1]．

機密データの漏洩検知手法の 1 つにテイント解析がある．テイント解析では，データがプログラムの変数間を伝

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 立命館大学
Ritsumeikan University
[a)] y.hayashi.036@nitech.jp

播する道筋であるデータフローを解析する．データフローの解析により，機密データの漏洩を検知し，データの保護や利用制限に役立てる．テイント解析可能であることの保証は，情報漏洩に対するプログラムの安全性保証の第一歩である．しかし，データフローの中には暗黙的フローと呼ばれる追跡が難しいフローが存在する [1]．このため，テイント解析による安全性保証のためには，暗黙的フローへの対策が不可欠である．

一方，一般に，ソースコードから実行ファイルを生成する際には，プログラムの実行時間やメモリ量の最小化を目的としてコンパイラ最適化を施す．コンパイラ最適化には，ループ構造の展開，変数の除去や定数化，不要なコードの除去等の処理が含まれる．これらは，コードの構造を変更する処理であるため，データフローに影響を与える可能性がある．より具体的には，コンパイラ最適化により暗黙的フローが除去される可能性や，反対に暗黙的フローを生み出す可能性がある．

そこで，(1) テイント解析を妨げる暗黙的フローとコンパイラ最適化の関係性について調査し，(2) 暗黙的フローをコンパイラ最適化の段階で除去する手法の提案と実装を行った．なお，本稿では，ループによる暗黙的フローのみを対象とし，分岐による暗黙的フローを対象とした調査と実装は今後の課題である．

本稿の構成は次のとおりである．まず，第 2 章でテイント解析に関する先行研究について述べる．次に，第 3 章で調査と提案手法の実装に用いた LLVM について説明し，第 4 章でテイント解析とコンパイラ最適化の関係性について LLVM を用いて調査した結果について述べる．第 5 章では調査の結果を踏まえて提案手法の概要について述べ，第 6 章で LLVM を用いた実装について述べる．最後に，第 7 章で提案手法の正当性について評価した結果について述べ，第 8 章でまとめる．

2. 関連研究

本章では，テイント解析とデータフローの詳細，及びテイント解析を妨げる暗黙的フローの解析に取り組んだいくつかの先行研究について述べる．

2.1 テイント解析

テイント解析とは，動的なデータフロー解析手法の 1 つである．解析対象のデータにタグと呼ばれる属性情報を割り当て，データの移動先にタグを伝播させる．解析時には，データに付加されたタグを調べることでデータの起源を調べる [2]．

2.1.1 明示的フロー

データフローは，明示的フローと暗黙的フローの 2 種類に分類される．明示的フローとは，値の直接的な代入関係が見られ，明示的にデータの移動を確認できるデータフローである．また，解析対象のデータをオペランドとして使用する演算の演算結果等，一部でも解析対象のデータを含む場合は明示的フローとして扱う [3]．

```
int secret = 0;
int public = 0;
for (int i = 0; i < 10; ++i)
  if (i == secret) public = i;
```

図 1 暗黙的フローの例.
Fig. 1 An example of implicit flow.

```
int public = 0;
for (int i = 1; i < 10; ++i) // ループ変数の開始値が 1
  if (i == secret) public = i; // 0の場合のフローはない
```

図 2 初期化値 0 を利用し，0 の時のフローをループ中に発生させずに暗黙的フローとする例.
Fig. 2 Implicit flow when secret is zero, public is initialized to zero and the loop begins from one.

```
int public = 0;
if (secret == 1)
  public = 1;
// else public = 0; のようなフローはない
```

図 3 初期化値 0 を利用し，0 の時のフローを分岐中に発生させずに暗黙的フローとする例.
Fig. 3 Implicit flow when secret is zero, public is initialized to zero and is not assigned zero after it.

2.1.2 暗黙的フロー

暗黙的フローとは，データの直接的な代入関係はないが，解析対象のデータの値が間接的に移動するフローである．暗黙的フローの例を図 1 に示す．機密データが格納された変数を `secret`，公開の可能性があり機密データが移動してはならない変数を `public` とする．図 1 では，変数 `secret` から `public` への明示的な代入は存在しないが，`secret` の値は明らかに `public` に移動している．このようなデータの移動を暗黙的フローと呼ぶ．

2.2 暗黙的フローへの対処法と解析の困難性

2.2.1 暗黙的フローの発生期間に着目した対処法

文献 [3] では，インフォメーションフローの追跡と制御による情報漏洩防止手法を提案している．メモリやレジスタを拡張した領域を利用し，保護対象データに識別タグを付加する．データを移動する命令の実行に合わせてタグも伝播させることでデータフローを追跡する．ただし，これが可能であるのはデータフローが明示的な場合のみである．暗黙的フローには，保護観察モードと呼ぶ特殊な実行モードの導入により対処する．

暗黙的フローで特に問題となるのは，処理を行わないパ

スを通った場合もデータフローに含まれるという点である．図 2, 3 を例に説明する．変数 secret の値は 0 であるとする．図 2 のように変数 public の初期値が 0 でループ変数の開始が 1 の場合，ループ中の処理としての 0 の代入は，明示的フローとしても暗黙的フローとしても存在しない．同様に，図 3 のように public の初期値が 0 で，0 の場合の分岐条件がない分岐の場合，分岐中の処理としての 0 のフローは存在しない．しかし，どちらの例でも public の値は 0 となるため，明らかに secret から public へのデータフローは存在する．これらの例のような，直接的な代入がないだけでなくループや分岐のパスすら存在しない場合，暗黙的フローの検出はより難しいものとなる．

このような暗黙的フローにも対応するため，文献 [3] では，暗黙的フローそのものの追跡ではなく，暗黙的フローの発生期間に着目した対処法を提案している．暗黙的フローを正確に追跡することは困難であるが，暗黙的フローの発生期間は，分岐の発生から合流までの間に限定できる．そこで，この期間をすべて保護観察モードとし，保護観察モード中はデータの出力を制限することで機密データの漏洩を防ぐ．しかし，分岐の合流地点を検出する手段がないため，合流地点は手動で定義する必要がある．

2.2.2 プロセス単位で保護する対処法

文献 [4] は，インフォメーションフローのセキュリティに関する 30 年分の研究をまとめており，暗黙的フローについても言及している．分岐による暗黙的フローの対処の例として，分岐開始後にプログラムの残りのプロセスを機密度の高いものとし，変数から読み取られたすべての値を機密データとして扱うことで漏洩を防ぐ方法が述べられている．前述の文献 [3] と同様に，暗黙的フローが含まれる可能性のある箇所を保護する手法は，保護範囲の終了地点の検出が困難であるという問題を抱えている．

2.2.3 タグの過剰伝播への対処法

文献 [5] は，暗黙的フローに対処したテイント解析の提案と実装である．暗黙的フローの問題点であるタグの伝播不足に対処することを目的とする．暗黙的フローの発生に際し，その後のデータフローが発生する可能性のすべてを追跡する方針は他の文献と大差ないが，この手法の問題性について述べている．タグの伝播不足は当然問題であるが，それに対処するために過剰に伝播させることもまた問題であると指摘している．文献 [5] では，依存関係を解析し，タグを伝播させる暗黙的フローの範囲を絞ることでこれに対処する．ただし，一般的なプログラム中の偶発的な暗黙的フローのみを対象としており，分析の妨害を目的として意図的に暗黙的フローが組み込まれる場合は考慮していない．

2.2.4 暗黙的フロー解析の困難性

暗黙的フローの解析精度向上を目指し，様々な研究が行われてきたが課題も残る．それぞれの対処法で解決可能な暗黙的フローもあれば解決ができないものもあり，暗黙的フローのテイント解析は難しいと言える．そこで，本稿では，暗黙的フローを解析可能にする方法の提案ではなく，事前に解析対象のコードから暗黙的フローを取り除くことで解析を容易にする方法を提案する．これにより，解析ツール側での暗黙的フロー対策に依存しないテイント解析の精度向上を目指す．

3. LLVM

本章では，提案手法の実装に用いた LLVM について述べる．また，LLVM 組み込みのテイント解析機構について考察する．

3.1 概要

LLVM [6] は，様々な最適化機構や幅広いアーキテクチャに対するコード生成機能等を提供するコンパイラ基盤であり，LLVM Core，Clang，LLDB 等のいくつかのサブプロジェクトから構成されている．コンパイラ基盤とは，コンパイラ生成のためのフレームワークであり，フロントエンド，ミドルエンド，バックエンドそれぞれの開発手段や開発支援機構を提供する．LLVM プロジェクトでは，LLVM Core にコンパイラ基盤としての機能が含まれ，本稿では提案手法の実装でこれを利用した．また，LLVM Core をバックエンドとして使用した C/C++/Objective-C/Objective-C++ のフロントエンド実装が Clang であり，提案手法の適用に利用した．

3.2 コンパイラ最適化

LLVM は，コンパイル時に様々な最適化を施す [7] [8]．最適化は Pass と呼ばれる形式で実装されており，Clang で標準的に使用される組み合わせが用意されている他，任意に組み合わせて利用することも可能である．また，LLVM の提供する機能を用い，独自の最適化 Pass を追加することも容易である．本稿で対象とするループの最適化だけでも複数の最適化 Pass が用意されている．例えば，ループ構造の展開を行うことで分岐やループ変数の計算を削減し高速化を図る Pass や，ループ変数の都度計算を省くため，定数化して変数の値を埋め込むような Pass が存在する．

3.3 LLVM IR

LLVM では，言語やプラットフォームに依存しない独自の中間表現を規定しており，これを LLVM IR と呼ぶ [9]．LLVM IR は無限個の仮想レジスタを仮定した Static Single Assignment (SSA) 形式で表現され，最適化や解析に適している．本稿でもこの LLVM IR を対象として実装を行った．

表 1　調査環境.
Table 1　Environment.

OS	Ubuntu 18.04.2 LTS
CPU	Intel(R) Core(TM) i3-4130 @ 3.40GHz
アーキテクチャ	x86_64
CPU 操作モード	32-bit, 64-bit
LLVM	7.1.0
Clang	7.1.0
Triton	0.6

3.4　DataFlowSanitizer

3.4.1　概要

DataFlowSanitizer [10] は，LLVM 組み込みの動的データフロー解析機能である．専用の関数を用い，ソースコード中で解析対象の変数を指定し，変数にラベルを付与する．実行時にラベルの伝播の様子を確認することでデータフローの解析を行う．

3.4.2　テイント解析への影響

DataFlowSanitizer では，ラベル付けした変数に影響する最適化をスキップする．これにより，最適化のテイント解析への競合が防止されるため，最適化と併用しても解析結果に影響はない．ただし，スキップされる最適化が多くなると最適化効率が下がる点は問題である．例えば，ループ変数をラベル付けした場合にループの最適化が完全にスキップされる等，最適化効率が大きく低下する可能性が考えられる．したがって，最適化効率を意識しながら解析を行う場合には DataFlowSanitizer を併用することは適切ではない．また，DataFlowSanitizer を使用すると，DataFlowSanitizer の解析用コードが追加で挿入されるため，実行ファイルのサイズ及び実行時間の増加も考慮する必要がある．

4.　テイント解析とコンパイラ最適化の関係

3.2 節で述べたように，コンパイラ最適化によりコードの構造が大幅に変更される場合がある．変数の定数化により，データフローが途切れて解析が困難になる可能性や，ループ構造の展開により，暗黙的フローが明示的フローに変換される可能性が考えられる．本章では，このような最適化が暗黙的フローにどのように影響するのかを調査した結果を述べる．調査環境は表 1 に示す．

4.1　概要

コンパイラ最適化が暗黙的フローに与える影響を調査するため，暗黙的フローを含むテストコードを作成し，最適化適用後のコードを解析した．暗黙的フローには様々なパターンが存在するが，本節ではその中でも代表的な 4 種類に対して，ループ回数や `break` の有無により条件分けした全 20 パターンを対象に検証を行った．

```
int secret = 4;
int public = 0;
for (int i = 0; i < 10; ++i)
  if (i == secret) public = i;
```

図 4　機密データが数値で `break` がない暗黙的フロー.
Fig. 4　Implicit flow without a break in the for statement when secret is a number.

```
int secret = 4;
int public = 0;
for (int i = 0; i < 10; ++i)
  if (i == secret) { public = i; break; }
```

図 5　機密データが数値で `break` がある暗黙的フロー.
Fig. 5　Implicit flow with a break in the for statement when secret is a number.

```
int secret = 4;
int public = 0;
for (int i = 0; i < secret; ++i)
  ++public;
```

図 6　機密データが数値でインクリメントを利用した暗黙的フロー.
Fig. 6　Implicit flow using increment when secret is a number.

4.2　調査方法

Clang の `-O3` オプションによる最適化を有効にして，暗黙的フローを含むソースコードをコンパイルした．コンパイル後，ソースコードの暗黙的フローが実行ファイル中に残っているか否かを解析した．実行ファイルの暗黙的フローの解析には，動的バイナリ解析 (DBA) フレームワークの Triton [11] を使用した．

4.3　調査対象

4.3.1　機密データが数値の場合

1 種類目は機密データが数値の場合の暗黙的フローである (図 4 参照)．ループ変数を介することで，変数 `secret` から変数 `public` への直接的なフローの発生を回避する．このループについて，if 文でのチェック後に `break` がないパターン (図 4) とあるパターン (図 5) とで最適化後に生成されるコードが異なることから，両パターンを調査対象とした．さらに，ループ回数が少ない場合と多い場合では最適化後のコードの構造が異なるため，それぞれについてループ回数が 10 回の場合と 1000 回の場合とを区別して検証した．

2 種類目は，変数 `public` を変数 `secret` の値の回数分だけインクリメントすることにより代入を回避する暗黙的フローである (図 6 参照)．この場合は，ループ回数が変数 `secret` によって決まるため，ループ回数の大小についての区別は付けていない．

```
char secret[] = "hello";
char public[STRLEN];
for (int i = 0; i < strlen(secret); ++i) {
  for (char c = 'a'; c <= 'z'; ++c)
    if (c == secret[i]) public[i] = c;
}
```

図 7　機密データが文字列で Ascii コードを 1 文字ずつ走査することによる暗黙的フロー.

Fig. 7　Implicit flow scanning the ascii code character by character when secret is a string.

```
char ascii[] = "abcdefghijklmnopqrstuvwxyz";
char secret[] = "hello";
char public[STRLEN];
for (int i = 0; i < strlen(secret); ++i)
  public[i] = ascii[secret[i] - 'a'];
```

図 8　機密データが文字列で Ascii テーブルを利用した暗黙的フロー.

Fig. 8　Implicit flow using an Ascii table when secret is a string.

表 2　機密データが数値の場合の調査結果.

Table 2　Results when secret is a number.

機密データ	定数		変数	
ループ回数	少	多	少	多
break あり (図 5)	✓$_1$	✓$_2$	✗$_3$	✓$_4$
break なし (図 4)	✓$_5$	✓$_6$	✗$_7$	✗$_8$
インクリメント (図 6)	✓$_9$		✓$_{10}$	

✓: 明示的フローに変換された
✗: 暗黙的フローが残った

4.3.2　機密データが文字列の場合

3 種類目は機密データが文字列の場合の暗黙的フローである (図 7 参照). Ascii コードを順に走査し，一致した Ascii コードを代入することで直接的なフローの発生を回避する [12]. 4 種類目は，Ascii テーブルを介することで直接的なフローの発生を回避する暗黙的フローである (図 8 参照). 文字列についても数値の場合と同様に，break ありの場合となしの場合，及び文字列長が 8 の場合と 1000 の場合をそれぞれ検証した.

4.4　調査結果

4.4.1　概要

4.3 節の暗黙的フローの各パターンに対し，さらに，変数 secret が定数の場合と変数の場合についてそれぞれ検証を行った. 変数の場合の値は，引数として与える場合と入力値として与える場合の両方を試したが，どちらの場合でも暗黙的フローに与える影響に変化はなかったため変数としてまとめて扱う. 機密データが数値の場合の結果を表 2 に，文字列の場合の結果を表 3 に示す. 表中に示した通し番号は，以後の説明でそれぞれのパターンを指し示すために使用する.

表 3　機密データが文字列の場合の調査結果.

Table 3　Results when secret is a string.

機密データ	定数		変数	
文字列長	短	長	短	長
break あり	✓$_{11}$	✓$_{12}$	✓$_{13}$	✓$_{14}$
break なし (図 7)	✓$_{15}$	✓$_{16}$	✓$_{17}$	✓$_{18}$
Ascii 表 (図 8)	✓$_{19}$		✗$_{20}$	

✓: 明示的フローに変換された
✗: 暗黙的フローが残った

```
printf("%d\n", 4);
```

図 9　機密データが定数の場合の最適化後のコードと同等のコード.

Fig. 9　Equivalent code to the optimized code when secret is a constant.

```
for (int i = 0; i < 1000; ++i)
  if (i == secret) { public = secret; break; }
```

図 10　パターン 4 の最適化後のコードと同等のコード.

Fig. 10　Equivalent code to the optimized code of pattern 4.

4.4.2　機密データが定数の場合

表 2, 3 のとおり，最適化により暗黙的フローが明示的フローに変換されるパターンが存在する. まず，機密データが定数の場合について考察する. 定数の場合，数値であるか文字列であるかにかかわらず，コンパイル段階で最終的な結果の解析が可能である. このため，図 4 の変数 secret が 4 の時に変数 public を出力する場合を例とすると，図 9 のようなコードと等価なコードに変換され，暗黙的フローを生成していたループは完全に消失する. break がある場合 (図 5) とインクリメントの場合 (図 6) も同様である. また，その他の定数のパターンに関しても，多少の違いはあるが，本質的には結果を直接出力するような形式に変換される. よって，暗黙的フローが明示的フローに変換されたと言える.

4.4.3　機密データが数値の変数の場合

機密データが変数の場合，コンパイル段階では機密データの値が未確定である. このため，定数の場合ほどシンプルなコードに変換されるわけでない. しかし，パターンによっては，ループ変数を介していた代入が secret の直接的な代入に変換されるため，データフローが明示的になる場合もある. これに当てはまるのがパターン 4 であり，図 10 のコードをコンパイルした場合と同等のコードに変換され，明示的フローとなる.

同様にループ回数の多いパターン 8 であるが，これは break がないためにパターン 4 ほどにシンプルなコードへの最適化とはならない. 最適化後のコードもループ変数を介した代入のままであった. break のありなしによる違いは他でも同様で，break がないコードは secret と一致し

```
int tmp = secret - 1;
int public = 0;
if (0 <= tmp && tmp <= 9) public = secret;
```

図 11　パターン 3 の最適化後のコードと同等のコード．
Fig. 11　Equivalent code to the optimized code of pattern 3.

```
if (secret == 0) public = secret;
if (secret == 1) public = secret;
if (secret == 2) public = secret;
(中略)
if (secret == 9) public = secret;
```

図 12　パターン 7 の最適化後のコードと同等のコード．大まかに表現した例．
Fig. 12　Conceptual code of the optimized code of pattern 7.

```
if (secret == 0) public = public ^ public;
if (secret == 1) public.set(1);
if (secret == 2) public = secret;
(中略)
if (secret == 9) public = secret;
```

図 13　パターン 7 の最適化後のコードと同等のコード．より厳密に表現した例．
Fig. 13　Equivalent code to the optimized code of pattern 7 with accuracy.

た後もループが続くために，break があるコードに比べて複雑なコードとなる．

一方，ループ回数が少ないパターン 3 はループ構造が完全に除去され，比較の後に secret を直接代入するようなコードとなる．しかし，0 の場合は初期値を利用することで代入を省くような最適化が施されるため，図 11 のコードのような構造となる．このため，secret が 0 の場合は暗黙的フローのままである．

また，パターン 7 は，0 から 9 までのループ構造を完全に展開した形となり，図 12 のコードのようにループが除去された構造となる．これに伴いループ変数が消えるため，大部分は明示的フローとなる．ここで，アセンブリ命令を効率的に使用することにより，代入以外の方法で値をセットする手法が問題となる．図 12 をより正確に表現すると図 13 のようなコードとなる．最適化としては正しい挙動であるが，0 は xor 命令，1 は sete 命令を用いて間接的に値をセットすることが可能である．この特性のために，0 と 1 のみが暗黙的フローのままとなる．

最後に，インクリメントを利用するパターン 10 は，完全にループ構造が除去されて secret から public への代入のみとなるため明示的フローである．ループ変数を介さないために，ループの最適化がより効率的に働いた例である．

4.4.4　機密データが文字列の変数の場合

文字列が数値と異なる点は，データの格納にスタックが

```
for (int i = 0; i < strlen(secret); ++i)
  if ('a' <= secret[i] && secret[i] <= 'z')
    public[i] = secret[i];
```

図 14　機密データが文字列の場合の最適化後のコードと同等のコードの例
Fig. 14　Equivalent code to the optimized code when secret is a string.

利用される場合が多く，アドレスを介した受け渡しになるという点，及びループが二重になるという点である．この特性のために，ループ変数を介した代入が直接的な代入に変換されやすい．具体的には，図 14 のコードのような最適化が行われる．内側ループは条件判定への簡略化と直接的な代入への置き換えにより，ループ構造が除去された形に変換される．

ただし，Ascii 表を用いたパターン 20 に関しては，ソースコードの暗黙的フローのまま，最適化後も Ascii 表を介した代入をするコードとなり明示的フローとはならない．ループは実行速度にも大きく影響するため，LLVM はできる限りループを展開するような最適化を施す．文字列の場合に明示的フローへの変換が多くなる理由は，二重ループの内側ループの展開に伴い，間接的な代入が直接的な代入に変換されたためである．このため，ループが二重ではなく，また，暗黙的フローの構造も複雑な Ascii 表を介したパターンは変換されなかったと考えられる．

4.4.5　まとめ

最適化の目的は暗黙的フローを取り除くことではないが，暗黙的フローは基本的に非効率な代入であるため，効率化の過程で明示的フローに変換される例があることが判明した．しかし，一般的な最適化のみですべての暗黙的フローを取り除けるわけではないこともまた明らかになった．最適化を暗黙的フローの除去に利用するためには，残った暗黙的フローを除去する手法が別途必要である．

5.　提案手法

本章では，提案手法の詳細，及び提案手法で対象とした暗黙的フローについて説明する．

5.1　提案内容

4.4 節の調査結果で残ったバイナリレベルでの暗黙的フローを明示的フローに変換する．これにより，暗黙的フローをコンパイル段階で完全に除去する．残った暗黙的フローが対象であるため，-O3 最適化で呼び出されるすべての最適化 Pass 適用後の LLVM IR を対象として実装を行った．

5.2　対象とする暗黙的フロー

提案手法のプロトタイプとして，最適化後に残った暗黙

```
mov    %edi,         %eax # secret => %eax
add    $0xffffffff,  %eax # secret - 1 => %eax
xor    %ecx,         %ecx # 0 => %ecx
sub    $0x9,         %eax # [1 <= secret <= 9] CF => 1
cmovb  %edi,         %ecx # [CF = 1] secret => %ecx
(中略)
mov    %ecx,         %esi # %ecx => public
```

図 15　最適化後の暗黙的フローを含むアセンブリ命令列.

Fig. 15　Assembly instruction sequence including the implicit flow of pattern 3.

```
%2 = add i32 %0, -1 ; secret + (-1)
%3 = icmp ult i32 %2, 9 ; 比較: (secret - 1) <= 9
%4 = select i1 %3, i32 %0, i32 0 ; 真: secret, 偽: 0
```

図 16　最適化後の暗黙的フローを含む LLVM IR 命令列.

Fig. 16　LLVM IR instruction sequence including the implicit flow of pattern 3.

的フローの一部に対する実装と評価を行った．実装の対象とした暗黙的フローはパターン 3 である．対応するソースコードは図 5 の `secret` が未確定の変数の場合であり，最適化後のコードは図 11 となる．また，最適化後のアセンブリ命令列は図 15 である．本稿では，変数 `public` を `printf()` に渡すことにより，表示という形で公開データを表現した．したがって，アセンブリコードでは，変数 `secret` の値が `printf()` の引数となる `%ecx` に暗黙的に渡されている場合が暗黙的フローとなる．

4.4 節で述べたように，パターン 3 は `secret` が 0 の場合に限って暗黙的フローとなる例である．これは図 15 のアセンブリからも分かるように，条件判定の前に `secret` から 1 引くことで 0 を比較条件から外すことに起因する．データフローに直接的に関わる `cmovb` 命令は CF がセットされている場合にのみ実行される命令であり，条件となる CF のセットは直前の `sub` 命令の結果次第である．この命令で CF がセットされる条件は，`0 <= %eax <= 8`，即ち，`1 <= secret <= 9` であるため 0 が除かれる．なお，0 の場合の `public` への値の代入は，`public` の初期化値 0 を用いることで間接的に実行される．以上を踏まえると，最初に 1 引くこと，変数 `public` の初期値が 0 であることの 2 つの条件が揃うコードが対象とする暗黙的フローの主な特徴である．

6. 実装

本章では，提案手法の実現のため，対象とする暗黙的フローを明示的フローに変換する方法について述べる．また，実装内容の適用方法についても述べる．

6.1 変換対象のコードの検出

まず，変換対象となる LLVM IR を検出する．図 15 のアセンブリ命令に対応する LLVM IR を図 16 に示す．また，各形式間での変数またはレジスタの対応関係を表 4 に示す．暗黙的フローの原因は，変数 `secret` の値を 1 引いてから条件判定をしている点にある．この操作のため，変数 `secret` が 0 の時は，条件判定の段階で負値となっており範囲外となる．これを踏まえ，各命令のオペランドに着目すると，図 16 のような命令の並び，即ち，`add icmp select` の並びがあり，`add` 命令と `select` 命令に同一の変数が使われていること，`add` 命令の結果が `select` 命令の条件に使われていること，さらにその条件が `ult` (unsigned less than) であることが主な特徴となる．これらの特徴を起点に変換対象の命令列を検出する．

表 4　主な変数またはレジスタの対応関係.

Table 4　Correspondence between main variables or registers.

形式	変数・レジスタでの表現	
	機密データ	公開データ
ソースコード (図 5)	`secret`	`public`
アセンブリ (図 15)	`%edi`	`%esi`
LLVM IR (図 16)	`%0`	`%4`

```
%2 = icmp ule i32 %0, 9 ; 比較: secret <= 9
%3 = select i1 %2, i32 %0, i32 0 ; 真: secret, 偽: 0
```

図 17　変換後の LLVM IR 命令列.

Fig. 17　LLVM IR instruction sequence after conversion.

6.2 コードの変換と手法の適用

変換の方針は，“1 引いてから条件判定” と結果が等価，かつ 1 引く処理が入らないようなコードに変換することである．そこで，図 17 に示す命令列に変換することを考える．図 17 では減算処理を除去し，代わりに `ult` を `ule` (unsigned less equal) に変更している．これにより，0 が条件範囲内に含まれるようになる．

以上の処理を LLVM の Pass として実装した．Pass は共有ライブラリとして提供可能である．即ち，コンパイル時に最適化オプションに加えて共有ライブラリの読み込み指定を追加するのみで，暗黙的フローを除去した実行ファイルを生成できる．

7. 評価

本章では，提案手法の評価について述べる．まず，実装の正しさの評価について述べ，次に，一般のプログラムを用いた評価について述べる．評価環境は表 1 と同様である．

7.1 Triton を用いた評価

表 2 の全項目に対し，テストプログラムを作成して検証を行った．まず，パターン 3 に対して正しくコードの変換が行われること，及び実行結果に問題がないことを確認した．次に，その他のパターンに対して予期しない書き換え

表 5 GitHub 上で公開されているプログラムを用いた評価の結果.
Table 5 Results of evaluation using programs published on GitHub.

	リポジトリ数
調査総数	887
Make の実行が開始	357
Make の実行に成功	161
`add icmp select` の命令列が存在	81
対象となる命令列が存在	0

等の副作用が起きないことを確認した．最後に，対象としたすべての暗黙的フローが明示的フローに変換されることを Triton を用いて確認した．以上より，実装は正当であると判断した．

7.2 公開されているプログラムを用いた評価

GitHub [13] 上で公開されている一般のプログラムのソースコードを取得し，提案手法を適用してビルドした結果を評価した．今回は，コンパイルの容易な CMake を用いたコンパイル手段を提供しているリポジトリに対象を限定した．評価結果を表 5 に示す．

ビルド工程を自動実行したため，依存パッケージが足りない等の理由で CMake や Make の実行段階で失敗したリポジトリがあった．81 件のリポジトリで `add icmp select` の命令列が検出されたが，その内で暗黙的フローとなる命令列は 1 件もなかった．原因として，暗黙的フローは基本的には悪意をもって組み込むものであるため，一般的なソースコードには含まれないこと，また，今回対象とした暗黙的フローの種類が限定的で少なかったことが考えられる．調査範囲を拡大してより広く検証すること，より多くの暗黙的フローのパターンに対応することは今後の課題である．

8. おわりに

調査により，コンパイラ最適化によって，暗黙的フローの一部が明示的フローに変換されることを明らかにした．また，最適化後のコードに残った暗黙的フローに対して提案手法を実装し，実際に暗黙的フローを削減できることを示した．以上より，既存のコンパイラ最適化と提案手法を組み合わせることで，コンパイル段階での暗黙的フローの削減が可能であると言える．

今後は，本稿で対応していない残りの暗黙的フローへの対応と，対象外とした分岐による暗黙的フローへの対応が課題となる．また，一般のプログラムにおいて，提案手法が適用される例を引き続き調査する．

参考文献

[1] 横田侑樹，塩谷亮太，五島正裕，坂井修一：情報漏洩防止プラットフォーム，全国大会講演論文集，Vol. 72, No. セキュリティ，pp. 629–630 (2010).

[2] 川古谷裕平，塩治榮太朗，岩村　誠，針生剛男：テイント伝搬に基づく解析対象コードの追跡方法，コンピュータセキュリティシンポジウム 2012 論文集，Vol. 2012, No. 3, pp. 1–8 (2012).

[3] 栗田弘之，塩谷亮太，入江英嗣，五島正裕，坂井修一：動的なインフォメーションフロー制御による情報漏洩防止手法，技術報告 17(2007-HPC-109) (2007).

[4] Sabelfeld, A. and Myers, A. C.: Language-based Information-flow Security, *IEEE J.Sel. A. Commun.*, Vol. 21, No. 1, pp. 5–19 (online), DOI: 10.1109/JSAC.2002.806121 (2006).

[5] Min, Gyung, K., Stephen, M., Pongsin, P. and Dawn, S.: DTA++: Dynamic taint analysis with targeted control-flow propagation, *the Network and Distributed System Security Symposium (NDSS)* (2011).

[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*, CGO '04, Washington, DC, USA, IEEE Computer Society, pp. 75– (online), available from ⟨http://dl.acm.org/citation.cfm?id=977395.977673⟩ (2004).

[7] Chris Lattner and Vikram Adve: The LLVM Instruction Set and Compilation Strategy, Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign (2002).

[8] Lattner, C.: LLVM: An Infrastructure for Multi-Stage Optimization, Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL (2002).

[9] 柏木餅子，　風薬，矢上栄一：きつねさんでもわかる LLVM -コンパイラを自作するためのガイドブック-，株式会社インプレスジャパン (2013).

[10] The Clang Team: DataFlowSanitizer, https://clang.llvm.org/docs/DataFlowSanitizer.html (Accessed 2019-08-02).

[11] Saudel, F. and Salwan, J.: Triton: A Dynamic Symbolic Execution Framework, *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, SSTIC, pp. 31–54 (2015).

[12] Newsome, J. and Xiaodong Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. (2005).

[13] GitHub, Inc.: GitHub, https://github.com/search (Accessed 2019-08-10).