Vol.2020-ARC-242 No.3
2020/10/12

情報量に着目したインフォメーションフロー追跡による情報漏洩防止手法の検討

小川 愛理[1,2,a)]　塩谷 亮太[2]

概要：情報機器の発展に伴い，様々な個人情報や機密情報がコンピュータ上で扱われるようになると共に，それらが外部に漏洩する情報漏洩による被害が増加している．そのような情報漏洩を防ぐために動的インフォメーションフロー追跡 (DIFT: Dynamic Information Flow Tracking) と呼ばれる手法を応用した技術が提案されている．DIFT はランタイム環境やハードウェアの上に実装され，それらの上で情報のフローを追跡する手法である．DIFT を使用した情報漏洩防止手法では，守りたい重要なデータに taint と呼ばれる情報を付加し，依存関係にある演算間で taint をもつデータのフローを追跡して taint を伝搬させる．フロー追跡の結果，もし伝搬された taint を持つデータが外部に出力されようとしている場合には，それを重要な情報の漏洩として検出する．しかし既存手法では直接の依存関係にない条件分岐などを介した暗黙的フローを適切に追跡できず，情報漏洩を正しく検出できないか，誤検出を起こしてしまうという問題があった．本論文では，taint 情報をその有無の二値ではなく，有限の範囲を持つ情報量として扱う手法を提案する．提案手法ではフローごとに入力から出力に伝わる情報量を動的に算出する．これにより，情報量の大小に基づく危険度の判定や，情報量の蓄積による漏洩の防止を実現し，より精度の高い情報漏洩の検出を目指す．

1. はじめに

コンピューターが扱う個人情報や機密情報は年々増加しており，その漏洩による被害が深刻な問題となっている．このような情報漏洩の多くはスパイウェアやトロイの木馬のような攻撃プログラムの実行によって引き起こされることが一般的である．攻撃プログラムは様々な方法で攻撃対象のコンピューターに侵入し，そのコンピューターにある個人情報などの秘密データを外部に流出させようとする．そのような秘密データの漏洩を防ぐため，攻撃プログラムをパターンマッチングやヒューリスティックを用いて検出することが広く行われている．しかしそれらの手法ではあらかじめ観察された攻撃プログラムからパターンなどを学習する必要があるため，未知の攻撃パターンの検出が難しいという問題がある．

そのような事前の学習を必要としないアプローチとして，**動的インフォメーションフロー追跡 (DIFT: Dynamic Information Flow Tracking)**[1] を用いた情報漏洩防止手法が研究されている，DIFT はランタイム環境やハードウェアの上に実装される手法であり，攻撃手法の個々のパターンによらず網羅的に情報漏洩を防ぐことができる．DIFT を用いた情報漏洩防止手法では，守りたい秘密データに **taint** と呼ばれるフラグを付加し，演算時に入力から出力へ taint を伝搬させることにより，taint をもつデータのフローを追跡する．フロー追跡の結果，もし taint を持つデータが外部に出力されようとしている場合には，それを重要な情報の漏洩として検出する．

この taint の伝搬で特に問題となるのが**暗黙的フロー**と呼ばれるデータフローである．暗黙的フローとは，複数のデータ間で明示的な演算や代入を介した依存関係がないにも関わらず，入力に依存して出力が決定される場合のデータフローである．暗黙的フローの代表的な例として条件分岐を介した制御フローがある．条件分岐の条件式に秘密データが使用された場合，条件分岐の結果を観測することにより秘密データの情報を間接的に知ることができる．たとえば `if(x==0){y=1}` のような文では，y が 1 か否かを観測することにより，x の値が 0 であるか否かを知ることが

[1] IBM 東京基礎研究所
[2] 東京大学 大学院情報理工学系研究科
[a)] ogawa@rsg.ci.i.u-tokyo.ac.jp

できる．そのため，暗黙的フローにおいても taint の伝搬を行う手法が提案されている [2], [3], [4].

しかし，それらの暗黙的フローを対象とした既存手法では，プログラムの正常な動作を妨げてしまうか，あるいは情報が漏洩してしまう問題がある．単純に全ての暗黙的フローにおいて taint の伝搬を行うと，本来不必要な部分にまで taint が伝搬してしまい，多くの場合にプログラムの正常な実行を妨げてしまう [2]．これに対し，いくつかの既存手法では情報漏洩につながらないと判断した条件分岐では taint 伝搬を行わないことで解決を図っている [3][4]．しかし，そのような伝搬を逃れた分岐を繰り返し実行することによって情報を蓄積させ，結果として情報を漏洩させることができる場合がある．

我々は上記の問題を，入力から出力へ伝搬する**秘密データの情報量**（以下では情報量と呼ぶ）と呼ぶ概念を導入し説明する．ここで，あるデータの持つ情報量とは，そのデータから伝搬元の秘密データをどの程度知ることができるかを意味する．暗黙的フローにおいて入力から出力へ伝搬する情報量は，条件分岐の条件式の内容によって変化する．たとえば if(x==0){y=1} のような文では y が 1 の場合に x を確定することができるため，大きな情報量が伝搬していると考える．暗黙的フローには極めて小さな情報量しか伝搬しないものもあれば，秘密データを一意に確定できるだけの大きな情報量を伝搬するものもある．

我々は，上記で述べた既存手法の問題は，本来問題とならないようなごく小さな情報量しかもたないデータを危険と判定することや，あるいは小さな情報量を蓄積させることで漏洩させているのだと考えた．既存の taint 伝搬手法は taint をその有無の二値でしか表現できないため，上記のような情報量に大小に基づく危険度の判定や，情報量の蓄積を捉えることが本質的にできない．

以上の考察から，我々は taint 情報をその有無の二値ではなく有限の範囲を持つ情報量として扱うインフォメーションフロー追跡手法を提案する．提案手法ではフローごとに入力から出力に伝わる情報量を動的に算出する．これにより，情報量の大小に基づく危険度の判定や，情報量の蓄積による漏洩の防止を実現し，既存手法の持つ問題を解決する．本論文では提案手法を Lua 言語上に簡易的に実装し，単純な評価プログラムを用いて初期評価を行った．評価の結果，既存手法では正常なプログラムの実行を妨げたり情報を漏洩している場合であっても，提案手法ではそれらを適切に処理できていることを確かめた．

本論文の構成は以下の通りである．まず 2 章で本論文が対象とする脅威モデルについて述べ，3 章で動的インフォメーションフロー追跡による情報漏洩防止手法について詳しく記述する．4 章では既存手法の持つ問題に関する我々の洞察について述べる．5 章ではこの洞察に基づいた提案手法について説明し，6 章で提案手法の評価結果とその考察を述べる．7 章で関連研究について説明し，その後まとめと今後の課題について論じる．

2. 脅威モデル

本章では，本論文で想定する攻撃モデルについて述べる．まず攻撃者は，個人情報などの秘密データを盗み出すことを目的として攻撃プログラムを作成する．そして攻撃者はその攻撃プログラムを攻撃対象のコンピューターに様々な方法で送り込み，ユーザーに実行させる．実行された攻撃プログラムは密かに取得した秘密データを外部に送信し，それを受信することによって攻撃者は秘密データを得る．

このようなモデルの典型的な例としては，たとえば攻撃者が一見有益なプログラムを作成して配布する場合があげられる．配布したプログラムを攻撃対象者のコンピュータ上で実行させ，その際に密かに取得した秘密データを外部に送信する．また別の例としては，メールなどに攻撃プログラムを添付する形で攻撃対象者に送信し，それを実行させる場合があげられる．いずれの場合でも，攻撃者は任意のコードを攻撃対象のコンピュータ上で実行でき，コンピューター上の秘密データを自由に取得できる．

Code 1: 攻撃プログラムの例

```
1 String imei = getDeviceId();
2 sms.sendTextMessage(..., imei, ...);
```

Code1 は識別情報を漏洩させる攻撃プログラムの例である．一般にスマートフォンなどのモバイル端末では，それぞれの端末に固有の IMEI (International Mobile Equipment Identity) と呼ばれる製造番号が存在する．WEB サイトなどにアクセスした際に IMEI を送信させ，それらの情報を解析することでユーザーの行動を追跡することができる．このため，IMEI は近年では重要な個人情報とみなされている．このコードは $getDeviceId()$ によって製造番号を取得し，それを $sms.sendTextMessage(...)$ によって外部に送信している．すなわちこのコードが攻撃対象の端末で実行されることで，その端末の製造番号が外部に送信され，攻撃者に漏洩する．

本論文では，正常なプログラムの実行を妨げずに上記のような攻撃プログラムによる情報漏洩を防ぐことを目的とする．より具体的には，上記のような攻撃プログラムが外部へ情報漏洩を行う際にそれを検知し，秘密データが外部に流出する前に実行を停止する．またこの際，端末内に閉じて正しく秘密情報を扱うプログラムについては，その実行を妨げないようにする．

3. 動的インフォメーションフロー追跡による情報漏洩防止手法

DIFT はランタイム環境やハードウェアの上に実装され，幅広い攻撃を検知するために利用される手法である．

DIFT を用いた個人情報等の秘密データの漏洩防止手法が広く研究されてきた [2][3][4]．DIFT では，守りたい秘密データに taint と呼ばれるフラグを付加し，依存関係にあるデータ間で taint を伝搬させることで taint をもつデータのフローを追跡する．フロー追跡の結果，もし taint を持つデータが外部に出力されようとしている場合には，それを重要な情報の漏洩として検出する．

DIFT では，データフローにおいて taint をどのように伝搬させるかを定める伝搬規則が重要になる．データフローには明示的フローと暗黙的フローが存在する．以下では明示的及び暗黙的フローについてそれぞれ説明し，その後暗黙的フローにおける taint 伝搬の問題について説明する

3.1 明示的フローの追跡

明示的フローとは，あるデータ同士が代入や演算などで直接的な依存関係にあるデータフローである．例えば $y = x + 1$ という演算を考える．この演算では，入力値 x から出力値 y に情報が伝わっている．これは，出力値 y を観測することによって入力値 x の値を完全に知ることができるためである．

このように入力と出力の間で直接的な依存関係がある明示的フローの追跡は比較的単純である．ある値に taint がついていた場合，その値は秘密データの情報を保持することを意味する．明示的フローでは入力値から出力値に情報が伝わるため，入力値が秘密データの情報を保持する場合には出力値も秘密データの情報を保持することになる．そのため，DIFT は出力値に入力値の taint を伝搬する．上記の例において x に taint が付いていた場合，その taint が y に伝搬される．

3.2 暗黙的フローの追跡

Code 2: 暗黙的フローのコード例

```
1 int y;
2 if (x == 0)
3     y = 0;
4 else
5     y = 1;
```

暗黙的フローとは，条件分岐を介して片方のデータからもう片方のデータへ情報が伝搬している場合のデータフローである．暗黙的フローでは，データ間に演算や代入を介した明示的な依存関係が存在しない．暗黙的フローは，条件分岐の条件式に含まれる入力変数に対し，その条件の成立や不成立に応じて異なる方法で出力変数を書き換える場合に生じる．このような場合，条件式と出力より入力の情報を得ることができる．Code2 に暗黙的フローの例を示す．この例では，入力値 x から出力値 y に一部情報が伝搬している．なぜなら，このコードを最後まで実行した際の出力値 y の値を観測することにより，入力値 x が 0 か否かを知ることができるためである．

データ間に暗黙的フローがあり条件式の入力に秘密データが使用された場合，その出力から秘密データの情報を得ることができる．このため，暗黙的フローにおいても taint を伝搬する DIFT の手法が提案されている [2], [3], [4]．暗黙的フローにおける taint 伝搬の最も簡単な実装では，条件式に taint の付いたデータが使用された場合に条件節内で書き換えられる可能性がある全ての変数に taint の伝搬を行う．たとえば上記の Code2 の例では，x に taint が付いていた場合，その taint を y に伝搬する．

3.3 暗黙的フローにおける taint 伝搬の問題

上記で述べたような単純な伝搬方法では不必要な部分にまで taint が伝搬してしまい，多くの場合にプログラムの正常な実行を妨げてしまうことが知られている．これに対し，多くの既存手法では各分岐ごとに選択的に taint 伝搬を行うことで解決を図っているが，しかし不適切な選択により情報が漏洩してしまう場合がある．以下ではそれらの手法とその問題について順に説明する．

3.3.1 全ての分岐で taint 伝搬を行う方法

Dytan[2] は暗黙的フローにおいて全ての分岐で taint を伝搬する手法であり，3.2 節で述べた最も単純な伝搬方法に相当する．Dytan は taint の過剰伝搬により，攻撃を意図しないプログラムの正常な実行を妨げてしまう場合がある [3], [4]．一般に，攻撃を意図しないプログラムにおいても秘密データを条件式に含む条件分岐は頻繁に現れる．たとえば，入力された文字列を新しいパスワードとして更新する際，その文字数が規定を満たしているか否かのチェックを行うことがあげられる．このようなプログラムにおいて全ての分岐で taint を伝搬すると，秘密データであるパスワードを条件式に利用した条件分岐内の処理すべてに taint が伝搬する．この結果，パスワードの文字数が規定を満たしているか否かを表示することや，パスワードが正常に更新された旨を表示することもブロックされてしまう．

3.3.2 分岐ごとに選択的に taint 伝搬を行う方法

上記の問題を解決するため，既存研究 [3] や [4] は暗黙的フローにおいて分岐ごとに選択的に taint を伝搬する手法を提案している．これらの手法では，秘密データを入力として使用する分岐でも情報漏洩に繋がりにくいものを見分け，それらに関しては taint 伝搬を行わない．これにより，攻撃を意図しないプログラムの正常な動作を妨げることなく暗黙的フローによる情報漏洩を防ぐことが理想的にはできる．

これらの手法ではたとえば，条件分岐の条件式がどの程度入力の範囲を限定しているかに着目する．その分岐の条件式が成立する場合，成立する際に取りうる値が限定されているほど，その出力から入力を決定することが容易にな

る．選択的に伝搬を行う既存手法では，そのような限定を強く行う条件分岐に限り taint を伝搬する．たとえば前述したように Code2 において 2 行目の $(x == 0)$ が真である場合，出力値 y を観測することで入力値 x が 0 であると一意に決定できる．このような一致比較は入力の取りうる範囲を強く限定しており，y を観測することで x が容易に決定できるため，このような場合は taint の伝搬を行う．

しかしこれらの選択的に伝搬を行う手法では暗黙的フローによる情報漏洩を防ぐことができない場合がある．攻撃者が，taint の伝搬規則を定めた特定の条件式に当てはまらない条件式を意図的に使用して攻撃プログラムを書くことにより，秘密データを漏洩させることができる．

Code 3: 先行研究で漏洩を防ぐことができない攻撃プログラムの例

```
1 int y;
2 if (x >= 0)
3     y = 0;
4 if (x <= 10)
5     y = y + 1;
```

具体的には，条件式による入力の限定が弱い分岐を繰り返すことにより，情報漏洩を行うことができる．Code3 は既存研究 [3][4] においてそのような情報漏洩を許してしまう攻撃プログラムの例である．2 行目と 4 行目の条件分岐では，どちらの条件分岐も条件式を真にするために取りうる値の範囲が広い．このため既存手法ではこれらの条件分岐では出力から入力をあまり限定できないと想定して，taint を伝搬しない．しかしこのコードを最後まで実行すると，y を観測することで x の取りうる範囲を強く限定することができる場合がある．たとえば y の値が 1 の場合，x の取りうる値の範囲は $0 <= x <= 10$ の非常に狭い範囲に限定できる．これは，$x >= 0$ と $x <= 10$ の個別の条件式ではそれぞれが取りうる値の範囲が広いが，複数の条件式を合わせた $x >= 0 \&\& x <= 10$ では，その結果としてとりうる値の範囲は限定されているためである．

4. 既存手法の問題に対する洞察

本節では上記で述べた既存手法の問題に対する我々の洞察を述べる．上記で述べたように，全ての分岐で taint を伝搬する方法と選択的に伝搬を行う方法では，情報漏洩の防止と正常なプログラムの動作の妨害の点において相反する問題がある．この問題に対し，以下では秘密データの情報量の概念を導入し，その情報量の伝播の観点から説明する．

4.1 秘密データの情報量

本節では変数間で伝搬する**秘密データの情報量**の概念を導入する（以下では単純に情報量と呼ぶ）．ここで，あるデータの持つ情報量とは，そのデータ（とプログラム内の定数や条件式）から，伝搬元の秘密データをどの程度確定できるかを意味するものとする．秘密データが N ビットの情報であり，伝搬先のデータから秘密データを M ビットの精度で確定できる場合，その伝播先のデータ持つ情報量を M とする．あるデータを観測することで秘密データを完全に確定することができる場合，そのデータは秘密データと等しい情報量を持ち，$N = M$ となる．逆に全く秘密データを復元できない場合は 0 となる．より形式的な定義については，次の 5 節で詳しく述べる．

この情報量は，明示的と暗黙的フローの双方において下記のように伝搬する．明示的フローでは，伝搬する情報量が演算の種類と入力ごとに異なる．たとえば $y = x + 1$ のような場合，基本的に出力値 y を観察することで入力値 x を一意に決定できる．したがって，この場合は入力から出力に全ての情報量が伝搬していると言える．一方，$y = x\&1$ の場合，y を観察しても x の最下位ビットしか決定できないため，伝搬している情報量は少ないといえる．

暗黙的フローでは，その伝搬元と伝搬先の間にある条件分岐の条件式が伝搬する情報量を決定する．ここで，条件分岐においてその分岐が成立した場合に伝搬する情報量について考える．その分岐の条件式が真になるために取りうる値の範囲が広いほど伝搬する情報は小さく，逆にその条件式を真になるために取りうる値が限定されているほど伝搬する情報量は大きくなる．

たとえば Code2 において 2 行目の $(x == 0)$ が真である場合，出力値 y を観測することで入力値 x が 0 であると一意に決定できる．この場合，出力の y に入力の x と等しい情報量が伝搬していると言える．一方，$(x == 0)$ が偽である，すなわち $(x \neq 0)$ である場合，出力値 y を観測すると x が非 0 であることがわかる．ここで x が 32 ビットの整数である場合，0 以外の取りうる値の数は $2^{32} - 1 = 4294967295$ 通りである．したがって，x は”4294967295 通りの整数の中のどれか”ということしか確定できず，その情報量は極めて小さい．

4.2 既存手法の問題と情報量

前述した既存研究の問題を，我々は情報量の観点から以下のように考える．

- 全ての分岐で taint を伝搬させるとプログラムの正常な実行を阻害してしまう．この問題は，伝搬される情報量が極めて小さい暗黙的フローにおいても taint 伝搬させ，それを情報漏洩として検出しているために生じる．前述したパスワードの長さのチェックの例では「パスワードが一定以上長いか否か」という情報については実際に漏洩しているのであるが，その情報量は小さく，そこからパスワードを復元することは事実上不可能であるため一般に問題はないとされている．

- 限定が弱い分岐においては taint を伝搬させない場合，分岐を繰り返すことにより情報が漏洩してしまう場合がある．これは個々の条件分岐において伝搬される情報量が小さい場合でも，複数の条件分岐を通ることで情報量を蓄積させることができるためである．

すなわち，プログラムの正常な実行を許すためにはごくわずかな情報量の流出を許容する必要がある一方で，情報量の蓄積による情報漏洩は適切に検出する必要がある．しかし，そもそも既存手法では taint がその有無の二値しか取り得ないために，情報量の大きさによる流出の可否の判定や，情報量の蓄積を実現することができない．

この洞察に基づき，我々は暗黙的フローにおいては秘密データが伝搬したかどうかという二値ではなく，どれくらいの情報が伝搬したかという情報量が重要であると考えた．次章ではこれを踏まえた提案手法について述べる．

5. 提案手法

前章までの考察を踏まえて，taint をその有無の二値ではなく有限の範囲を持つ情報量として扱うインフォメーションフロー追跡手法を提案する．提案手法では，フローごとに入力から出力に伝わる情報量を特定し，蓄積することにより暗黙的フローを適切に追跡することができる．これにより既存手法より正確に情報漏洩を検出することができる．以下ではまず有限範囲の情報量を持つ taint について定義し，次に情報量を用いたフロー追跡手法について述べ，最後に暗黙的フローにおける情報量の管理手法について述べる．

5.1 情報量を持つ taint の定義

提案手法では，データフローによって入力から出力に伝搬される taint の情報として有限の範囲の情報量を用いる．この情報量を **taint 量**と呼ぶ．taint 量を以下に定義する．

- taint 量は，その taint を持つ変数が秘密データの情報量をどれくらい含むかを示す．より具体的には変数が持つ秘密データのビット幅で表現される．
- taint 量の最大値は秘密データのビット幅である．taint 量が最大値のとき，その taint を持つ変数は秘密データと等しい情報量を持つことを意味する．すなわち，その変数を観測することによって秘密データの情報を完全に復元できる．
- taint 量の最小値は 0 である．taint 量が最小値のとき，その taint をもつ変数は秘密データの情報量をもたないことを意味する．
- 通常の taint と同様に，taint 量は明示的及び暗黙的フローによって伝搬されるが，その際に taint 量は最小値から最大値の範囲内で増減する．例えばあるフローによって情報量が蓄積する場合には taint 値が加算される．

定義に示す通り，taint 量は秘密データのビット幅で表現される．例えばある秘密データが 8 ビット文字を 4 文字含む文字列であった場合，その秘密データの taint 量は $8 * 4 = 32$ である．taint 量が大きいほど秘密データの情報を多く持つことを意味する．したがって大きな taint 量を持つ変数の外部流出を検出するが情報漏洩防止の観点から重要となる．

5.2 明示的フローの追跡手法

Code 4: taint 量を用いた明示的フロー追跡の例

```
1 void outputSecret(uint8_t secret) { // secretのtaint量: 8.0
2   int x = secret+1; // xのtaint量: 8.0
3   int y = x >> 1; // yのtaint量: 7.0
4   int z = x + y; // zのtaint量: 8.0
5   output(z); // 情報漏洩
6 }
```

明示的なフローにおいては，taint 量は依存関係を持つ演算の入力から出力へ伝搬させる．明示的フローにおける taint の伝搬規則は基本的には二値の taint の伝搬と同様で，入力値の taint 量は単純に出力に伝搬する．ただし特定の演算は情報量を削減するため，その場合には入力値の taint 量から適切に計算した taint 量を出力に伝搬する．また taint 量を持つ入力値が複数ある場合には，入力の taint 量の最大値を出力に伝搬する．

Code4 の例を用いて taint 量による明示的フローの追跡を具体的に説明する．このコードでは *outputSecret* という関数に対し秘密データである 8 ビット符号なし整数の *secret* という変数が入力値として与えられている．*secret* は 8 ビットの秘密データであるので，その taint 量は 8.0 となる．

コードの 2 行目では，32 ビット符号付き整数の x に対し *secret* に 1 を足した数を代入している．x を観測することで *secret* の値を一意に決定することができることから，*secret* の情報量が x にそのまま伝搬させる．したがって x には *secret* の taint 量である 8.0 が伝搬される．

コードの 3 行目では，同じく 32 ビット符号付き整数の y に対し，x を 1 ビット右に論理シフトした値を代入している．これも直接の依存関係を持つ明示的フローの一種であるが，入力値の taint 量を減らす効果を持つ特殊なケースである．x は 8 ビットの秘密データの情報量を持つが，1 ビット右にシフト演算を行うことによって最下位ビットの情報が失われる．つまり y を観測することによって x を完全に知ることはできず，最下位 1 ビット以外の 7 ビット分の情報のみ知ることができる．したがって y は x の taint 量 8.0 から，1 ビット分少ない taint 値 7.0 が伝搬される．

コードの 4 行目では，2 行目と 3 行目で定義された x と y を入力値とし，それらを足したものを出力値 z に代入し

ている．taint 量を持つ入力値が複数ある場合には，その最大値を出力値に伝搬する．x は 8.0, y は 7.0 の taint 量をそれぞれ持つため，z に伝搬される taint 量は 8.0 となる．

3 行目の *output* 関数において，z の値を外部に出力すると仮定する．z の taint 量は 8.0 であり，これは秘密データ 8 ビット全ての情報量をもつことを意味する．z を外部に出力することは秘密データの情報をそのまま漏らすことと等しい．そのため，提案手法ではこれを情報漏洩として検出する．

5.3 暗黙的フローの追跡手法

前節で説明した明示的フローに比べ，暗黙的フローにおいて taint 量を適切に伝搬することは単純ではない．その理由は 3.3 節で示したように，そのフローに到達するための条件式の内容によって入力から出力に伝わる情報量が変化するためである．

暗黙的フローを介して伝搬される情報量を計算するためには以下の二つの情報が必要である．

入力値の taint 量条件式の入力値の taint 量

真となる場合の数条件式を真にするために入力値が取りうる値の数

入力値の taint 量は条件式への入力値が持つ秘密データの情報量を示す．真となる場合の数が小さいほど，その条件式を真にするという事象が起こりにくい，すなわち情報量が大きいことを示す．

自己エントロピー の定義式: P この二つの情報を用いて，暗黙的フローを介して伝搬される情報量 I は自己エントロピーの定義から以下の式で計算される．

$$I = -\log_2(P) = -\log_2(\frac{\text{真となる場合の数}}{2^{(\text{入力値の taint 量})}}) \qquad (1)$$

P : その条件式が真である確率

Code 5: taint 量を用いた暗黙的フロー追跡の例

```
1 void outputSecret(uint8_t secret) { // secretのtaint量: 8.0
2     int x;
3     if (secret == 0)
4         x = 0; // xのtaint量: 8.0
5     else
6         x = 1; // xのtaint量: 0.0
7     output(x); // 情報漏洩
```

Code5 を用いて暗黙的フローの追跡手法を具体的に説明する．1 行目は明示的フローの例と同様である．3 行目の条件分岐が真である場合には 4 行目の $x = 0$ が実行される．このとき 3 行目の条件式の入力値 *secret* から 4 行目の代入文の出力値 x に伝搬する情報量を考える．

3 行目の条件式 $(secret == 0)$ において，入力値の taint 量は *secret* の taint 量である 8.0 である．真となる場合の数は $(secret == 0)$ を真にする *secret* の値の数であり，このとき *secret* は 0 でなければならないため，1 通りである．前述の式を用いて情報量を計算すると $-\log_2(\frac{1}{2^{8.0}}) = 8.0$ となり，この条件式が真であることによって伝搬される秘密データの情報量は 8.0 となる．よって 4 行目の出力値である x には 8.0 の taint 量が伝搬される．3 行目の条件式が真である場合，x の値が 0 であることを観測することによって *secret* の値が 0 であることが間接的にわかる．すなわち x は *secret* の 8 ビットの情報を完全に確定できる．このことは x に *secret* と同じ taint 量 8.0 が伝搬されるという結果に矛盾しない．

条件式 $(secret == 0)$ が偽であった場合，6 行目の else 節が実行される．つまり 6 行目の実行のための条件式は $(secret! = 0)$ である．この条件式の入力値の taint 量は 3 行目と同じ 8.0 である．真となる場合の数は $(secret! = 0)$ を真にする *secret* の場合の数であり，*secret* が 0 以外の 8 ビットの整数となるため，$2^8 - 1 = 255$ 通りである．前述の式を用いて情報量を計算すると前述の式を用いて情報量を計算すると $-\log_2(\frac{255}{2^{8.0}}) = 0.056...$ というとても小さい値になる．6 行目の出力値である x にはほぼ 0 に近い taint 量が伝搬される．else 節が実行される場合，x の値を観測することによってわかる情報は *secret* が非 0 であるということであり，その情報量は極めて小さい．このことは x にほぼ 0 の taint 量が伝搬されるという結果に矛盾しない．

5.4 複数の暗黙的フローの追跡手法

前節では暗黙的フローにおける taint 量の計算とその伝搬について説明した．しかしプログラムの実行にあたって複数の異なる条件分岐を通る場合，個々の条件分岐に対する情報量だけを計算しても，正しい情報量が得られない場合がある．

たとえば，$(secret > 1)$ という条件式と $(secret < 10)$ という条件式を順番に通り，その結果が共に真であった場合を考える．このとき，プログラムの結果的には $(secret > 1)\&\&(secret < 10)$ という条件式を通り，真であったことと等しい．$(secret > 1)\&\&(secret < 10)$ を通ることで伝搬する情報量を仮に I とする．$(secret > 1)$ を通ることで伝搬する情報量を I_1，$(secret < 10)$ を通ることで伝搬する情報量を I_2 としたときに $I \neq I_1 + I_2$ である．なぜならば，I_1 と I_2 はどちらも *secret* に対する情報量であり，条件の範囲が被っているため，違いに独立ではないからである．この場合にプログラムを通して正しく情報量を得るためには，個々の条件式で情報量を計算するのではなく，今まで通った全ての条件式の結果から情報量を計算する必要がある．

そこで我々は，taint 量をもつ各変数に対し taint 量に加えて二つのメタデータを持たせ動的なシンボリック実行を行うことによって，複数の異なる条件分岐を通る場合においても正しく情報量を計算するための手法を提案する．

シンボリック実行とは，プログラム上の変数をシンボルとして扱い、シンボルに対する一連の操作を分析することで条件を満たす入力値を特定するプログラム解析手法である．これは通常静的に解析が行われるが，提案手法ではこれを動的に行う．これにより，各変数が通る全てのパスに対するパス制約を得ることで，その変数がもつ秘密データの情報量を正しく解析する．

通常シンボリック実行によって得られるパス制約は，様々な変数のシンボルを含む．しかしこの手法では，あくまである変数が持つ秘密データの情報量だけ知ることができれば良いため，パス制約に必要なシンボルは伝搬元の秘密データだけである．そのため，動的にその他のシンボルの数値を代入し，伝搬元の秘密データのシンボルみを含むパス制約を得る．こうして得られたパス制約を解くことで，最終的に取りうる秘密データの値の範囲を求めることができる．

この手法において各変数が持つ taint 情報は以下の 3 つである．

- 自身の taint 量
- 秘密データの taint 量
- パス制約

自身の taint 量は前節までで説明した各変数が持つ taint 量である．この値は明示的フローにおいては 5.2 節の例のように代入や演算により伝搬される値であるが，暗黙的フローにおいては秘密データの taint 量と，パス制約によって決定される真となる場合の数から式 1 によって計算される値となる．秘密データの taint 量は，伝搬元の秘密データの taint 量を示す．たとえば，ある 8 ビットの秘密データの部分情報をもつ変数において，秘密データの taint 量は 8 である．パス制約は，動的シンボリック実行によって得られたパス制約を示す．前述の通り，これは伝搬元の秘密データのシンボルのみを含むように動的にその他のシンボルを解決される．

Code6 の例を用いて複数の暗黙的フローの追跡を説明する．このコードの入力である *secret* の設定は前節の Code4 と同様である．*secret* は 8 ビットの秘密データであるので，その taint 値は 8.0 になる．このコードでは 4 行目の条件分岐によって 5 行目の then 節もしくは 7 行目の else 節が実行される．加えて，8 行目の条件分岐によって 9 行目の then 節もしくは else 節が実行される．このコードを実行した際に通る各パスについて，動的シンボリック実行を行うことで得られるパス制約を図 1 に示す．

以下では 4 行目及び 8 行目の条件分岐が共に真であったと仮定して説明する．図の (a) は Code6 の 2 行目に該当する．x はこの時点では秘密データの taint 量及びパス制約をもたない．4 行目の条件分岐の実行で条件式が真であった場合，5 行目の $x = 0$ が実行される．これは図 1 の (b) に該当する．5 行目が実行されるためには $(secret >= 0)$ の条件式が真である必要があるため，(b) のパス制約に $(secret >= 0)$ が追加される．同時に伝搬元の秘密データである *secret* の taint 量 8.0 が秘密データの taint 量として代入される．5 行目の実行後，8 行目の条件分岐の実行でも条件式が真であった場合，9 行目の $x = x+1$ が実行される．これは図 1 の (c) に該当する．このとき $(secret <= 10)$ の条件式が真であるため，(c) のパス制約に $(secret <= 10)$ が追加される．

Code 6: taint 量と動的シンボリック実行を用いた複数の暗黙的フロー追跡の例

```
1   void outputSecret(uint8_t secret) {
2       int x;
3       if (secret >= 0)
4           x = 0;
5       else
6           x = 1;
7       if (secret <= 10)
8           x = x + 1;
9       else
10          x = x + 3;
11      output(x)
```

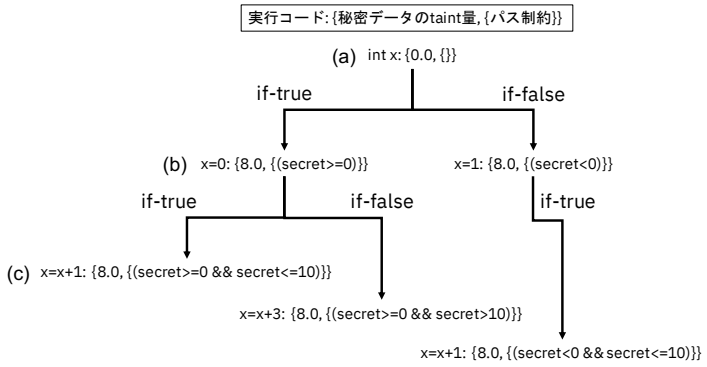

図 1: Code6 を実行した場合の各フローにおけるパス制約

結果として，コードが 12 行目の出力関数まできた際，出力値の x は $(secret >= 0 \&\& secret <= 10)$ というパス制約をもつ．このパス制約から *secret* の取りうる場合の数（0 以上 10 以下の整数なので 11 通り）を求めることができる．秘密データの taint 量と，パス制約によって決定される真となる場合の数から式 1 を用いて x の情報量は 4.54... と計算できる．この場合 x は秘密データを 4.5 ビットほど確定できる大きな情報量を持つため，外部流出は危険であると判断できる．そのため提案手法ではこれを情報漏洩と検出して実行を停止させる．

6. 評価

6.1 評価環境

スクリプト言語 Lua の実行環境を用いて提案手法と既存手法の Dytan[2] と DTA++[3] の評価を行った．Lua の各

変数は自身の値と共に taint を持つように変更し，taint の伝搬は評価手法の taint 伝搬規則に従って Lua VM によるランタイム中に行われる．変数が持つ taint は，Dytan と DTA++においては二値のフラグであり，提案手法においては前章で説明した taint 量となっている．

提案手法の実装は，初期検討として複数の暗黙のデータフローを検出するための動的シンボリックを簡易的にしたものを実装した．具体的には，前章で説明したパス制約に相当する条件分岐の条件文を taint 量と一緒に各変数が持てるようにし，条件文からもとまる秘密データの限定範囲から taint 量を計算する．この簡易実装では，動的シンボリック実行を行なっているわけではないため，秘密データの限定範囲を完全に追跡できるわけではない．しかし単純に複数の条件分岐を通るだけの簡単なプログラムであればこの実装でも複数の暗黙的フローを追跡し，情報量を得ることができる．この評価では Lua スクリプトの入力引数に対し taint をつけるよう設定し，外部出力される変数の taint を観察することで taint 伝搬が適切に行われているかを確認する．

6.2 評価プログラム

Code 7: 入力パスワードのチェックを行うプログラム

```
1  local str
2  if #secret >= 8 then
3      str = "valid␣password!"
4  else
5      str = "invalid␣password"
6  end
7  print (str)
```

Code 8: 情報漏洩を意図した攻撃プログラム

```
1  local x = 0
2  if secret >= 'a' then
3      x = x + 1
4  else
5      x = 0
6  end
7  if secret <= 'z' then
8      x = x + 1
9  else
10     x = 0
11 end
12 print (x)
```

評価プログラムとして，攻撃を意図しない入力パスワードのチェックを行うプログラム（プログラム 1）と情報漏洩を意図した攻撃プログラム（プログラム 2）の二つを作成した．二つのプログラムは評価環境で実行するため，Lua で記述される．

プログラム 1 はユーザーの入力した文字列 *secret* が 8 文字以上かどうかを確認し，その結果をユーザーに文字列として表示するプログラムである．パスワードが文字数制限を満たしているかどうかを確認する処理は一般的に行われることであり，このプログラムは秘密データである *secret* を外部に流出させることを意図しない．したがって文字列 *str* の外部出力を情報漏洩と認識せず，正常実行させることが求められる．

一方プログラム 2 は情報漏洩を意図した攻撃プログラムである．外部出力される x の値を確認することで，ユーザーが入力した文字 *secret* が小文字のアルファベットであるかどうかという情報が漏洩する．秘密データである *secret* の内容がある程度特定できるプログラムになっており，整数値 x の外部出力が情報漏洩を行っていると認識して実行を停止することが求められる．

6.3 評価結果

表 1: 情報漏洩の検出結果

DIFT 手法	プログラム 1	プログラム 2
Dytan	yes (NG)	yes (OK)
DTA++	no (OK)	no (NG)
提案手法	no (OK)	yes (OK)

評価対象の三つのインフォメーション追跡手法を実装した Lua の実行環境で二つの評価プログラムを実行し，情報漏洩を検出したかどうかの結果を表 1 に示す．

Dytan はプログラム 1，2 共に情報漏洩であると検出した．Dytan では条件式に taint のついたデータが使用された場合に条件節内で書き換えられる可能性のある全ての変数に taint の伝搬を行う．したがってプログラム 1，2 共に節内の変数に taint が伝搬し，その外部流出を情報漏洩であると検知する．しかしプログラム 1 は情報漏洩を意図するプログラムではなく，これは誤検出である．このように Dytan では，taint の過剰伝搬により多くのプログラムで用いられる一般的なコードに対しても情報漏洩を検出してしまい，正常な実行を妨げてしまう．

DTA++はプログラム 1，2 共に情報漏洩を検出しなかった．DTA++では条件分岐ごとに選択的に taint 伝搬を行うことで taint の過剰伝搬を防ぐ手法である．具体的には $(secret == 0)$ という式のように，分岐の条件式を真にするために取りうる値が 1 つに決定される場合にのみ taint 伝搬を行う．しかしプログラム 1，2 共に条件分岐の条件式は *secret* が取りうる値の範囲を 1 つに決定しない．例えばプログラム 2 の 2 行目の条件式は $(secret >=' a')$ であり，ASCII コード a の数値 97 より大きい値は全てこの式を真として通過する．このとき DTA++では taint の伝搬を行わない．7 行目の条件分岐においても同様であり，結果として出力値の x に taint は伝搬されず，プログラム 2 は情

報漏洩として検出されない．しかしながら，前節で説明したようにプログラム 2 は情報漏洩を意図する攻撃プログラムであり，DTA++は情報漏洩を正確に検出できていない．

提案手法ではプログラム 2 だけが情報漏洩であると検出した．プログラム 1 は暗黙のフローによって情報の伝搬が起こるが，伝搬される情報量はごくわずかである．外部出力される *str* が持つ taint 量がこくわずかであるため，提案手法はこれを情報漏洩であるとはみなさない．プログラム 2 では，2 つの条件分岐を通ることによって結果的に多くの情報が x に伝搬している．*secret* が小文字のアルファベットであるとき，2 つの条件分岐を通ることで x が得る taint 量は.. である．このとき提案手法は x の外部流出を情報漏洩であるとみなす．

提案手法は各分岐により伝搬される情報量を taint 量として持ち，伝搬する．各条件分岐によって伝わる情報量が小さい場合でも，その分岐を複数回通ることによって結果的に多くの情報が伝搬する場合に，これを検知できる．提案手法では最終的に外部流出される変数の持つ taint 量が十分大きい場合に情報漏洩であると判断する．どれくらいの taint 量を持つ時にこれを情報漏洩とみなすかは重要な閾値であり，今後の実験や考察によって明らかにする必要がある．

7. 関連研究

DIFT は元々，SQL インジェクションやクロスサイトスクリプティングといったプログラムの脆弱性を突く攻撃を防止する手法である [1][5][6]．プログラムの未知の脆弱性を検知するために，DIFT では信頼できない外部入力に対し taint を付与し，データフローにしたがって taint 伝搬を行う．伝搬した taint が入力データに付加されているかを，システムに影響を与えるような重要な操作を行う部分（SQL エンジンやファイル API の入力部分など）で確認する．もし taint が付いていた場合は外部から攻撃が挿入されたと見なしてプログラムを停止することにより攻撃を防ぐことができる．

情報漏洩防止のための DIFT 手法は広く研究されているが，その多くは暗黙的フローを考慮していない [7][8]．暗黙的フローにおける taint 伝搬を考慮した手法としては，前述の Dytan[2] や DTA++[3] の他に PIFT[9] がある．PIFT はモバイル環境での情報漏洩を防ぐための軽量な DIFT 手法を提案している．ハードウェアで DIFT を行う際には，一般的に全ての CPU 命令におけるデータフローを追跡し taint の伝搬を行う必要があり，オーバーヘッドが課題となる．PIFT ではあるデータがメモリから LOAD されてから STORE されるまでの時間的局所性に着目し，全ての CPU 命令ではなく直近の LOAD と STORE 命令でのみ taint の伝搬を行うことで効率的にフローを追跡する手法を提案している．条件分岐にかかわらず LOAD と STORE の時間距離でのみ taint の伝搬が起こることから暗黙的フローを介した taint の伝搬を行うことができるが，追跡精度の点で課題がある．攻撃プログラムの製作者が PIFT の taint 伝搬規則を知っている場合，依存のある LOAD と STORE の間に余計な命令を複数挿入することで taint 伝搬を妨害することができる．そのため情報漏洩を起こす攻撃プログラムを検出できない恐れがある．

8. おわりに

これまでに情報漏洩を防ぐために DIFT を応用したさまざまな手法が提案されてきたが，既存手法では暗黙的フローを適切に扱うことができず，正常なプログラムの実行を妨げたり情報漏洩を引き起こしてしまっていた．これに対し，本論文では taint 情報をその有無の二値ではなく，有限の範囲を持つ情報量として扱う手法を提案した．本論文では提案手法を Lua 言語上に簡易的に実装し，単純な評価プログラムを用いて初期評価を行った．評価の結果，既存手法では正常なプログラムの実行を妨げたり情報を漏洩している場合でも，提案手法ではそれらを適切に処理できていることを確かめた．今後は，伝搬アルゴリズムの詳細化とその実装，およびさまざまな複雑な実用プログラムを用いた評価を行う予定である．

参考文献

[1] Suh, G. E., Lee, J. W., Zhang, D. and Devadas, S.: Secure Program Execution via Dynamic Information Flow Tracking, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI (2004).

[2] Clause, J., Li, W. and Orso, A.: Dytan: A Generic Dynamic Taint Analysis Framework, *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07 (2007).

[3] Kang, M., McCamant, S., Poosankam, P. and Song, D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation., *Proceedings of the Network and Distributed System Security Symposium*, NDSS '11 (2011).

[4] de Araújo, L. S., Marzulo, L. A. J., Alves, T. A. O., França, F. M. G., Koren, I. and Kundu, S.: Building a Portable Deeply-Nested Implicit Information Flow Tracking, *Proceedings of the 17th ACM International Conference on Computing Frontiers*, CF '20 (2020).

[5] Dalton, M., Kannan, H. and Kozyrakis, C.: Raksha: A Flexible Information Flow Architecture for Software Security, *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07 (2007).

[6] Li, K., Shioya, R., Goshima, M. and Sakai, S.: String-Wise Information Flow Tracking against Script Injection Attacks, *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing* (2009).

[7] Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y. and Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks, *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39.

[8] Slowinska, A. and Bos, H.: Pointless Tainting? Evalu-

ating the Practicality of Pointer Tainting, *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09 (2009).

[9] Yoon, M., Salajegheh, N., Chen, Y. and Christodorescu, M.: PIFT: Predictive Information-Flow Tracking, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16 (2016).